Advanced Distributed systems
Professor Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology Delhi
Lecture 07
Logical Clocks and Physical Clocks

(Refer Slide Time: 0:18)

Synchronous Systems
Asynchronous Systems

Synchronization
Physical Clocks, Logical Clocks

Smruti R. Sarangi

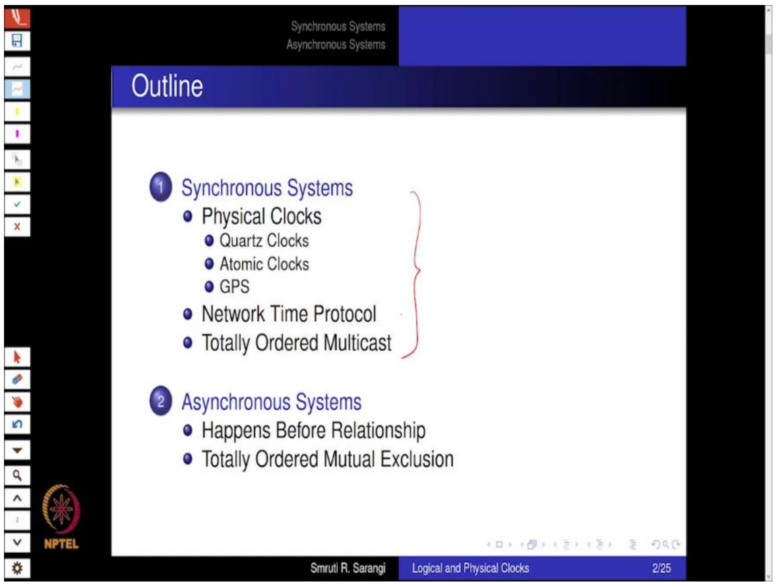
Department of Computer Science
Indian Institute of Technology
New Delhi, India

Smruti R. Sarangi | Logical and Physical Clocks | 1/25

Welcome to the chapter on synchronization. So, in this chapter we will talk about clocks, so we will talk about logical and physical clocks, because DHTs are not the only things that can be built using distributed systems and distributed algorithms, so for all other kinds of algorithms which we shall see in the second part of this course, we will see that there is a need for different kinds of clocking mechanisms to maintain time.

So, we can either have physical time or we can have what is called logical time. In a distributed algorithm, logical time is clearly more important but physical time is also used, so we will see when what is used?

(Refer Slide Time: 01:07)


Synchronous Systems
Asynchronous Systems

Outline

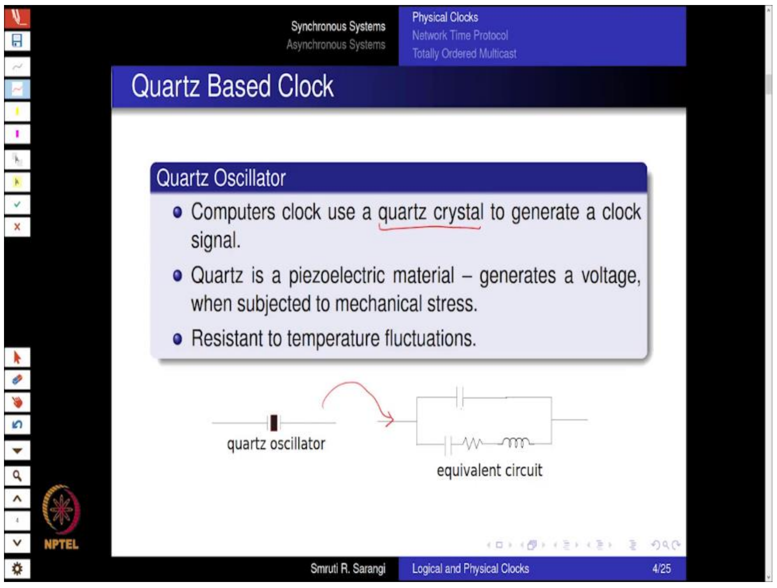
1. Synchronous Systems
   - Physical Clocks
     - Quartz Clocks
     - Atomic Clocks
     - GPS
   - Network Time Protocol
   - Totally Ordered Multicast
2. Asynchronous Systems
   - Happens Before Relationship
   - Totally Ordered Mutual Exclusion

Smruti R. Sarangi | Logical and Physical Clocks | 2/25

So, we will discuss what is called a synchronous system? This relies on physical clocks or it relies on systems where there is some notion of timing, and then we will discuss asynchronous systems that do not rely on any notion of timing. So, it basically means that the delay of a message can be very large.

So, whether it is finite or infinite that again is a theoretical issue, but essentially you cannot assume anything about the time it takes to process a message or send a message, or the message transmission time that cannot be assumed.

(Refer Slide Time: 01:50)


Synchronous Systems
Asynchronous Systems
Physical Clocks
Network Time Protocol
Totally Ordered Multicast

Quartz Based Clock

Quartz Oscillator

- Computers clock use a quartz crystal to generate a clock signal.
- Quartz is a piezoelectric material – generates a voltage, when subjected to mechanical stress.
- Resistant to temperature fluctuations.

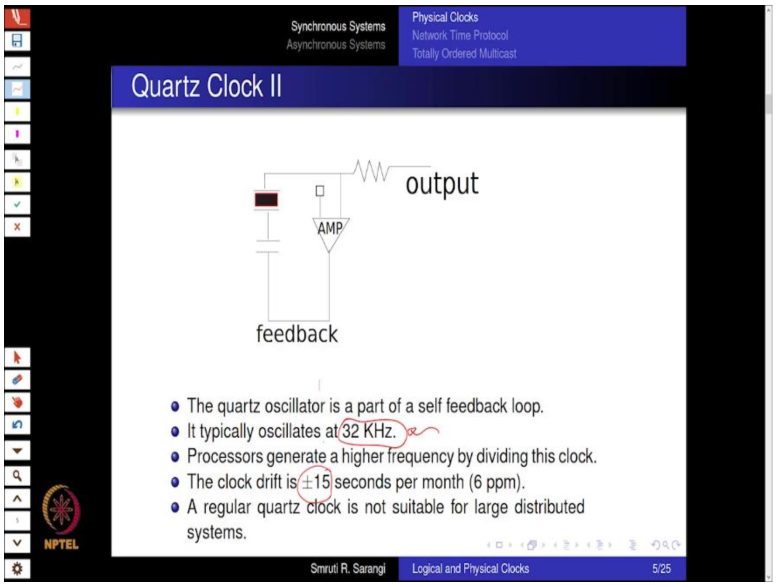
Smruti R. Sarangi | Logical and Physical Clocks | 4/25

So, we will discuss physical clocks first, so we will understand the way that you know the physical clocks work in our watches and within computers, so inside a computer also there is a clock, so the basic idea is the same. So, the idea is that a quartz crystal is used to generate a clock signal. What is quartz?

Quartz is a piezoelectric material which generates a voltage when subjected to mechanical stress, and when it is subjected to a voltage stress when in the sense when you change the potential across it, then also mechanical stress develops on it, vice versa. So, the basic quartz oscillator that you have over here, the basic oscillator can be represented with an equivalent circuit which is shown over here.

And every quartz oscillator has a natural frequency, a natural frequency of oscillation, which is essentially the resonant frequency of this circuit. So, basically what happens is that there is a feedback mechanism; via the feedback mechanism every quartz oscillator would oscillate at a certain frequency which for it is the natural frequency.

(Refer Slide Time: 03:10)


Synchronous Systems
Asynchronous Systems
Physical Clocks
Network Time Protocol
Totally Ordered Multicast

Quartz Clock II

- The quartz oscillator is a part of a self feedback loop.
- It typically oscillates at 32 KHz.
- Processors generate a higher frequency by dividing this clock.
- The clock drift is ±15 seconds per month (6 ppm).
- A regular quartz clock is not suitable for large distributed systems.

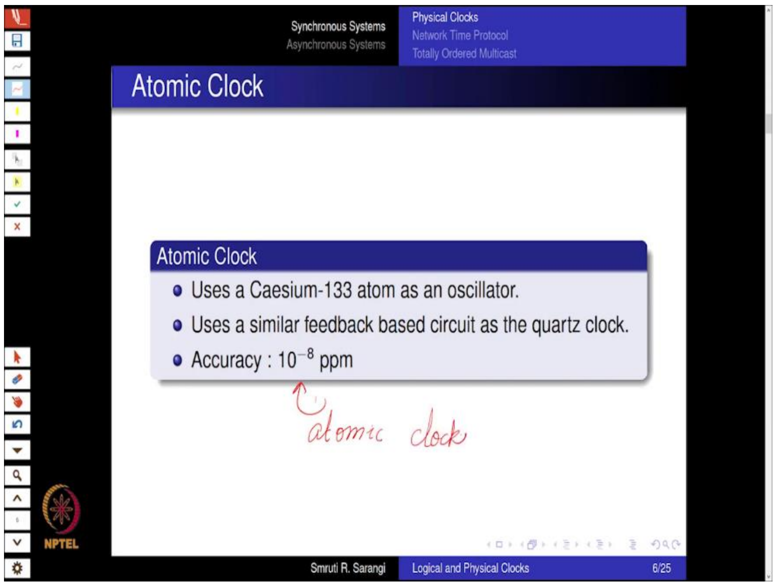
Smruti R. Sarangi   Logical and Physical Clocks   5/25

So, what happens, as I just showed that there is a quartz oscillator that is part of a cell feedback loop, and there is, of course, an amplifier here to amplify the signal to properly embellish the signal, so the quartz oscillator here typically oscillates at 32 kilohertz, so that is the base frequency of the oscillator.

Furthermore, this clock can be divided using a standard clock divider circuit and that can be used to create a much higher frequency. So essentially this is can be thought of as the time base and then this time base can be used to construct a clock with a much much higher frequency. The clock drift is very little. It is ±15 seconds per month, which is fine, so most of our watches do not maintain time to that extent.

And the clock drift per se is not an issue it is only six parts per million, but nowadays of course quartz technology has improved by leaps and bounds, so it is getting much better by the day. But nevertheless, for large distributed systems, a regular quartz clock is not suitable, because even the 6-ppm error might not be either, might be on the higher side, so you would want a lower error.

(Refer Slide Time: 04:37)



Atomic Clock

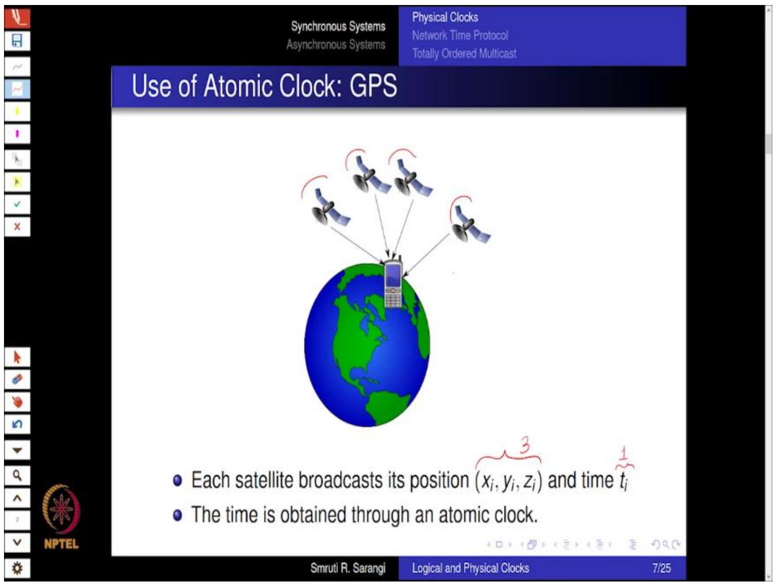
Atomic Clock

- Uses a Caesium-133 atom as an oscillator.
- Uses a similar feedback based circuit as the quartz clock.
- Accuracy : 10^{-8} ppm

atomic clock

So, we use a Caesium-133 atom as an oscillator, so this is very accurate. So, this also uses a similar feedback, mechanism feedback-based circuit as the basic, what is clock is accuracy is 10^{-8} ppm, 10^{-8} paths per million, which means it is extremely accurate, extremely-extremely-extremely accurate and this is used and this is what is typically there in what is called an atomic clock.

(Refer Slide Time: 05:11)



Use of Atomic Clock: GPS

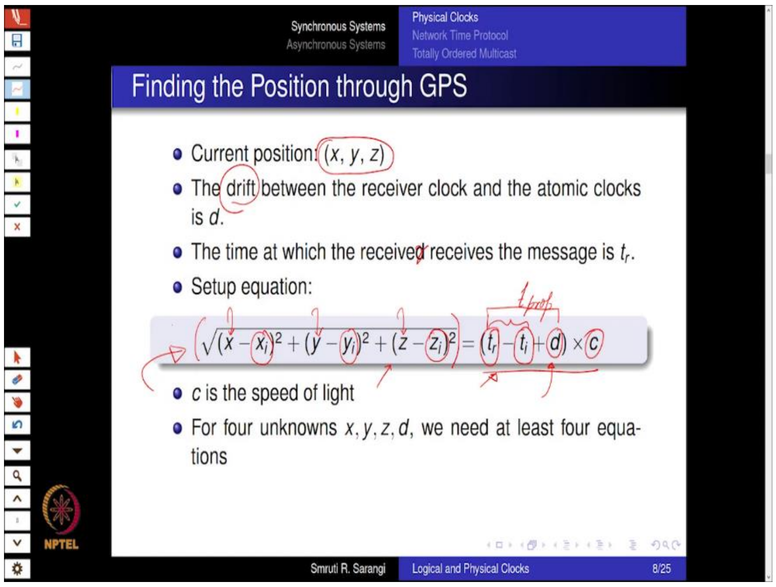
- Each satellite broadcasts its position $\overbrace{(x_i, y_i, z_i)}^{3}$ and time $\overbrace{t_i}^{1}$
- The time is obtained through an atomic clock.

So, now let us see that how the atomic clock is actually used, so we will come to the use of the GPS system. So, what happens is, that let us consider what is called satellite GPS? So what we actually use in our mobile phones is what is called a GPS? or assisted GPS? which means it triangulates based on the positions of multiple mobile towers; that is not exactly what we are looking at? we are talking about regular satellite GPS.

So, let us consider a simple version of the problem, where there are four unknowns, so one unknown, of course, is the current time such, so, we do not know the current time very accurately, even the mobile phone does not know because it does not have an atomic clock. And of course, it is x, y, and z coordinates.

So given the fact that the three unknowns here and one unknown over here, a total of four unknowns, we would need four equations, and the four equations will come when we have four independent sets of data, and these four sets of data would require four satellites. So, this is basically required for satellite GPS, and this is how time is obtained via an atomic block.

(Refer Slide Time: 06:26)


Synchronous Systems
Asynchronous Systems
Physical Clocks
Network Time Protocol
Totally Ordered Multicast

Finding the Position through GPS

- Current position: (x, y, z)
- The drift between the receiver clock and the atomic clocks is d.
- The time at which the receiver receives the message is t_r.
- Setup equation:

$$\sqrt{(x-x_i)^2+(y-y_i)^2+(z-z_i)^2} = \overbrace{(t_r - t_i}^{t_{prop}} + d) \times c$$

- c is the speed of light
- For four unknowns x, y, z, d, we need at least four equations

Smruti R. Sarangi | Logical and Physical Clocks | 8/25

So, the basic physics is very simple, it is simply based on the propagation of light and the Euclidean geometry. So, of course there are some relativistic effects that also do come into this and that is of course required but I am talking of a much simpler formulation. So, if the current position is x, y, z relative to some reference, relative to some inertial reference frame, the drift between let us say the receiver clock and atomic blocks is d.

So we are assuming that all the atomic clocks on the satellites, there is no drift between them, but there is some drift between the clock of the receiver which in this case is the mobile phone, and the atomic clocks. So, let us say the time at which the receiver receives the message, let it be “tr”, so we can set up a small equation.

So, the small equation that we will set up over here is basically this is the distance, which as you can see this is this simple Euclidean distance, where Xi, Yi, and Zi are basically the coordinates in this case of the satellite. So, tr - ti is the time that we are reporting, so what happens is that whenever the satellite sends a message it affixes its timestamp along with the message, and when the GPS receiver receives the message, it looks at the current time, so it then subtracts them it is tr - ti.

$$\sqrt{(x-x_i)^2+(y-y_i)^2+(z-z_i)^2} = (tr - ti + d)\ X\ C$$

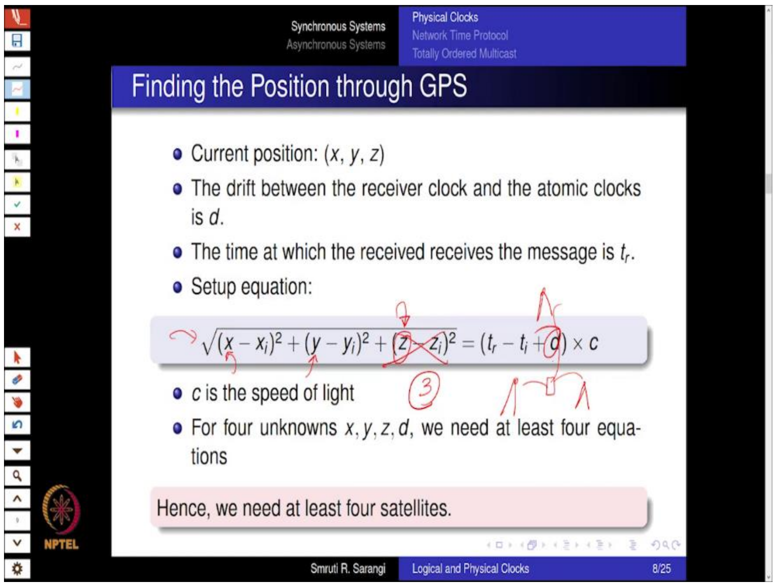
And so, this is roughly the time it took for the signal to come from the satellite to the receiver, but of course since there is a drift in this, we add that, so this becomes the time. So, the drift in this case is an unknown, we do not know that, and so if you look at the previous slide, we were assuming that the time the exact time is an unknown, but this has been changed to make drift the unknown.

Mathematically, they are the same thing just a rearrangement of terms. So, given the fact that this is actually the time of propagation, we multiply that with the speed of light. And then after multiplying that with the speed of light, what do we get? So we have distance on the right-hand side, and distance on the left-hand side, so we have an equation in front of us. This equation has four unknowns, so what are the four unknowns?

The four unknowns over here are x, y, z, x y, z in this case are the position of the receiver x, y, and z coordinates and of course the drift, so which is the drift in time. Xi, Yi, and Zi is essentially the position of the ith satellite, and "ti" is its time, which has been affixed with the message. And given that all the satellites haven't have an atomic clock, we are assuming that there is no drift between them.

Now, given that we have four unknowns, we need four equations. So, since we have four satellites, each satellite contributes an equation of this type, so four equations that are quickly solved.

(Refer Slide Time: 09:47)



Finding the Position through GPS

- Current position: (x, y, z)
- The drift between the receiver clock and the atomic clocks is d.
- The time at which the received receives the message is t_r.
- Setup equation:

$$\sqrt{(x-x_i)^2+(y-y_i)^2+(z-z_i)^2} = (t_r - t_i + d) \times c$$

- c is the speed of light
- For four unknowns x, y, z, d, we need at least four equations

Hence, we need at least four satellites.

So, What we need to do is, we just set up these four equations and then all that your GPS receiver does is that it solves these four equations. There are many techniques on how to solve them, I am not getting into that but let us put it this way, it is easy to solve.

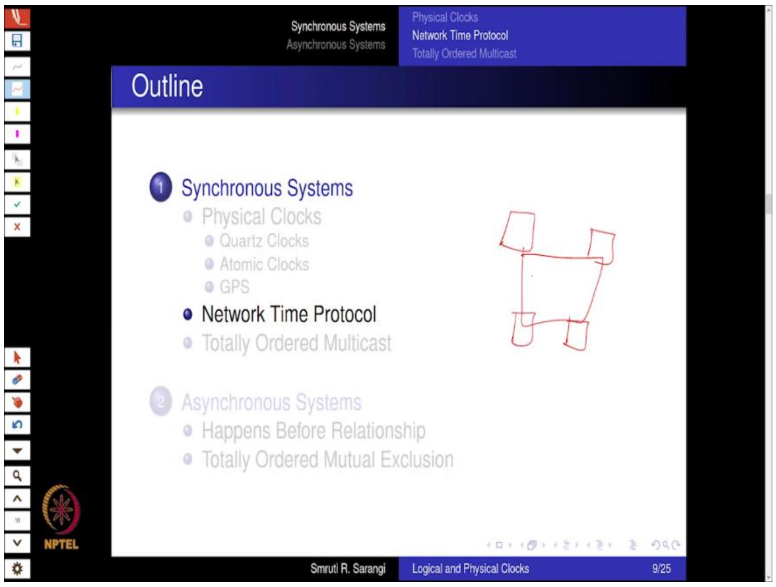
And, once it is solved we can also find the drift meaning, we can find the exact time, we can compensate for the drift and find the exact time on the receiver as well as we can find its coordinate x, y, and z. So normally, we do not use the z coordinate, so we can make a fixed assumption. If we know what is the elevation of a certain region.

Then of course we can just have x, y, and the drift, so then we would need three satellites, which is most of the time done incidentally or if we really want the z coordinate let us say for an aircraft or something, we would need four satellites. So, this is of course a very exact mechanism of doing it, but let us say that if the z coordinate is not an issue, what we would need is that we would need to triangulate.

So, then let us say that if this is not an unknown, we would need three satellites or three other places that can give us the exact time. So, normally what we have in assisted GPS is that we triangulate, so when we triangulate what we actually get is that we find the distance from three mobile phone towers and that gives us an estimate of the current location.

So, this is normally what we refer to as GPS, and this as you can see is much cheaper, much cheaper than actually getting the coordinates from the satellite. And since most mobile phones are in a region with a signal, this is often easy to do it is easy to achieve.

(Refer Slide Time: 11:50)


Synchronous Systems
Asynchronous Systems
Physical Clocks
Network Time Protocol
Totally Ordered Multicast

Outline

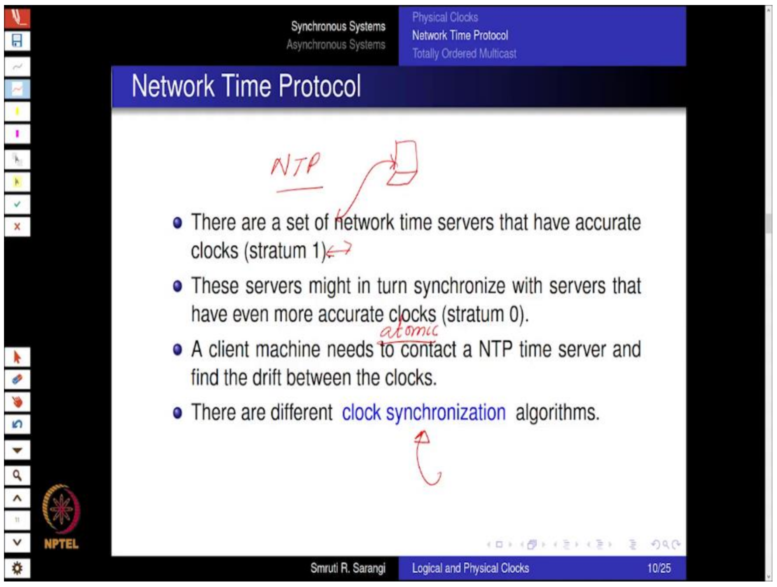
1. Synchronous Systems
   - Physical Clocks
     - Quartz Clocks
     - Atomic Clocks
     - GPS
   - Network Time Protocol
   - Totally Ordered Multicast
2. Asynchronous Systems
   - Happens Before Relationship
   - Totally Ordered Mutual Exclusion

Smruti R. Sarangi | Logical and Physical Clocks | 9/25

Now, let us consider a slightly different protocol. So, in this protocol what happens is that there is no external time base like there is no satellite or there is no mobile phone tower, there are just a set of machines and they want to ensure that between themselves they have a synchronized time.

So, given that there is an external provider, they will have to somehow ensure that all of them are following the same time base, and this time base might not match with the real-world clock time that is okay, insofar as running a distributed algorithm is concerned, as long as all of them refer to the same time it is okay, use the same time base it is all right.

(Refer Slide Time: 12:36)


Synchronous Systems
Asynchronous Systems
Physical Clocks
Network Time Protocol
Totally Ordered Multicast

Network Time Protocol

NTP

- There are a set of network time servers that have accurate clocks (stratum 1)
- These servers might in turn synchronize with servers that have even more accurate clocks (stratum 0).
- A client machine needs to contact a NTP time server and find the drift between the clocks.
- There are different clock synchronization algorithms.

atomic

Smruti R. Sarangi | Logical and Physical Clocks | 10/25

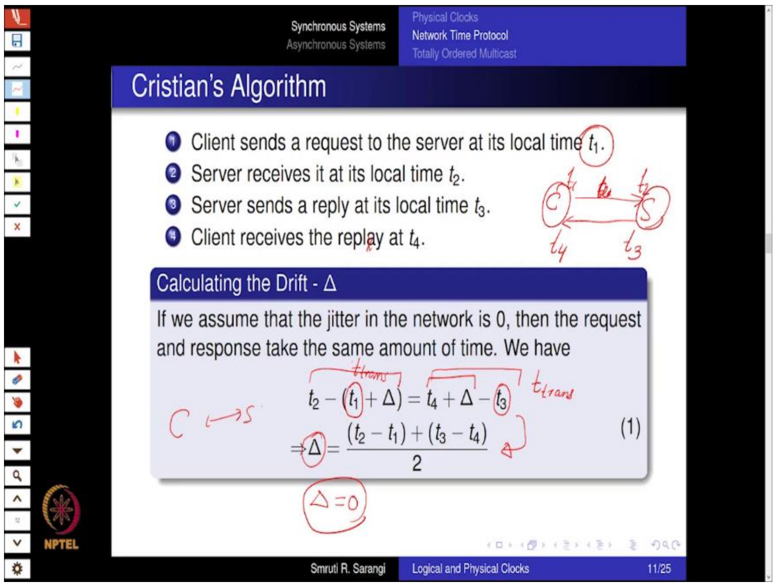
So, this is where we use the network time protocol, you would have heard about this in many scenarios so this is called NTP, normally. So, what happens is that we have a set of network time servers that have accurate clocks, let us call it the stratum ones. These servers synchronize themselves with an even more accurate clocks which are mostly atomic clocks, which are stratum 0, but these atomic clocks are typically not accessible to common users like us, so we would basically synchronize our times.

If you go to the windows time setting or Linux’s time setting, it will give you an option to synchronize the time with a network time server, and so these network time servers are run by numerous agencies, and most universities also run them, major vendors also run them, does not matter, they are all in stratum 1, all that my machine needs to do is it needs to connect to a network time server and synchronize the time with it.

Mobile phones also do that; you would have often seen that. Let us say you take a flight and land up in the region with a different time zone or let us say your mobile phone is off, you charge it on the plane, and then when you land, what you would actually see is that the moment it gets its signal, the time changes.

The time changes mainly because it synchronizes the time with the clock of the tower, so this is a stratum 1 synchronization. And client machines contact the NTP time servers, find the drift between the clocks, and update their clock. So, this is of course a simple example, but we look at slightly more complicated examples where we discuss advanced forms of clock synchronization.

(Refer Slide Time: 14:26)



Synchronous Systems | Asynchronous Systems — Physical Clocks | Network Time Protocol | Totally Ordered Multicast

Cristian's Algorithm

1. Client sends a request to the server at its local time t_1.
2. Server receives it at its local time t_2.
3. Server sends a reply at its local time t_3.
4. Client receives the replay at t_4.

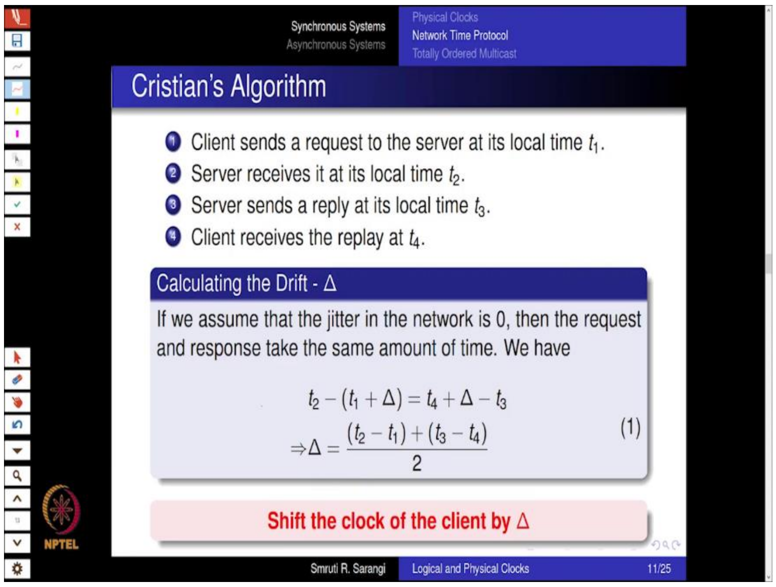
Calculating the Drift - Δ

If we assume that the jitter in the network is 0, then the request and response take the same amount of time. We have

$$\overbrace{t_2 - (t_1 + \Delta)}^{t_{trans}} = \overbrace{t_4 + \Delta - t_3}^{t_{trans}}$$
$$\Rightarrow \Delta = \frac{(t_2 - t_1) + (t_3 - t_4)}{2} \qquad (1)$$

$\boxed{\Delta = 0}$

Smruti R. Sarangi | Logical and Physical Clocks | 11/25



Synchronous Systems | Asynchronous Systems — Physical Clocks | Network Time Protocol | Totally Ordered Multicast

Cristian's Algorithm

1. Client sends a request to the server at its local time t_1.
2. Server receives it at its local time t_2.
3. Server sends a reply at its local time t_3.
4. Client receives the replay at t_4.

Calculating the Drift - Δ

If we assume that the jitter in the network is 0, then the request and response take the same amount of time. We have

$$t_2 - (t_1 + \Delta) = t_4 + \Delta - t_3$$
$$\Rightarrow \Delta = \frac{(t_2 - t_1) + (t_3 - t_4)}{2} \qquad (1)$$

Shift the clock of the client by Δ

Smruti R. Sarangi | Logical and Physical Clocks | 11/25

So, what happens is that we will discuss the simplest algorithm first it is called the Cristian's Algorithm. So, here the client sends a request to the server at its local time ‘t1’. So, let us consider the client and the server so it sends a request at its local time ‘t1’. The server receives it at time ‘t2’, which is its local time.

Server then sends a reply at its local time ‘t3’, which the client receives the reply at ‘t4’. So, if we assume that the jitter in the network is 0, which means that this it took the same time to send the message from the client to the server, and it took exactly the same time for the reply to come back. If we assume that this jitter is 0, then the request and response take the same amount of time, what do we have? What we have is t2 – (t1 + Δ).

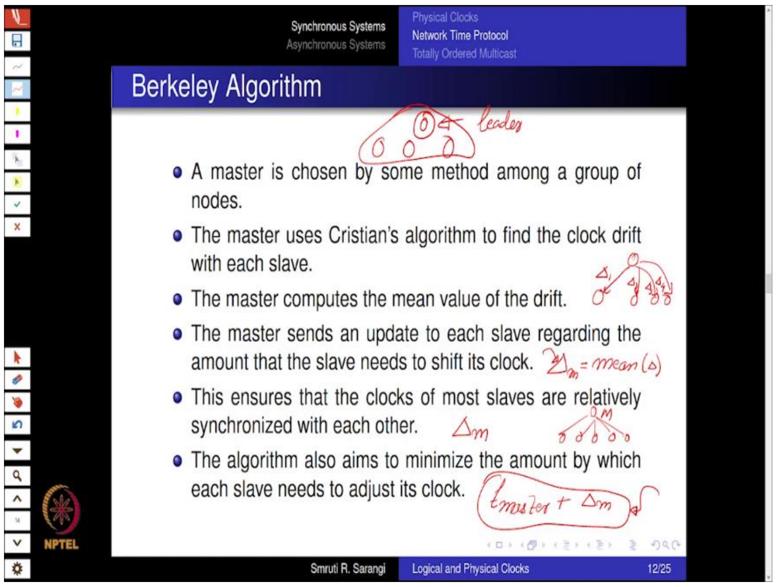
So let us say that when the time was t1 at the client, the actual time was (t1 + Δ) with respect to the server. So, the delta here is a drift, so this is the time, this is essentially the message transmission time, so let us call it t-trans. Similarly, when the client received the message, it received the message at ‘t4’, which is the client's local time.

So the server's local time would be (t1 + Δ) – t3, where delta is the drift between the clocks, and this (- t3), where t3 is the time at which the server sent relative to its own clock, so this is the transmission time again t trans. And given the fact that we are assuming that there is no jitter in the network, which means it takes the same time to send and same time to receive, we have an equation over here.,

And the value of Δ = $\frac{(t2-t1)+(t3-t4)}{2}$ which is just a simple algebraic reduction from here. What this essentially teaches us is that, if we just consider the client a client and a server regardless of any external time base, it is possible for the client to figure out the clock drift between it, between its clock and the server's clock, and then the client can adjust its local time such that the $\Delta = 0$.

So, in this case, even in isolation, a client and a server can synchronize their clocks using this simple algorithm which is called the Cristian’s Algorithm. Of course, the key assumption that is being made over here is that the jitter is 0, which means that the message transmission time regardless of the sender or receiver as long as the link is the same, remains the same, regardless of the direction. So then, at the end what happens the client shifts the clock by delta, and that sets their clocks to be the same.

(Refer Slide Time: 17:47)


Synchronous Systems
Asynchronous Systems
Physical Clocks
Network Time Protocol
Totally Ordered Multicast

Berkeley Algorithm

- A master is chosen by some method among a group of nodes.
- The master uses Cristian's algorithm to find the clock drift with each slave.
- The master computes the mean value of the drift.
- The master sends an update to each slave regarding the amount that the slave needs to shift its clock.
- This ensures that the clocks of most slaves are relatively synchronized with each other.
- The algorithm also aims to minimize the amount by which each slave needs to adjust its clock.

Smruti R. Sarangi Logical and Physical Clocks 12/25

Now, we will discuss the Berkeley Algorithm. So, the Berkeley Algorithm is an extension over and above the Cristian's Algorithm. So, in this case, a master is chosen by some method among a group of nodes. So, let us say we have a group of nodes, one among them is chosen as the master, so the algorithm calls in master, but let us call it a leader, so they choose a leader. Well, how will they choose a leader?

Remember, this we will discuss leader election algorithms, maybe one or two lectures later, but let us assume that there is some way for the nodes to select a leader which is one distinguished node among them, let us not look at failures for the timeline. The leader slash master will use the Cristian's algorithm that we just studied to find the clock drift with each slave. So, what does the leader do?

It basically finds the clock drift with each slave. We call them master and slave over here, not leader and follower. Then what the master does is that it computes the mean value of the drift. So, master then sends an update to each slave regarding the amount that the slave needs to shift its clock. So, what this basically means is that, once let us say you compute all the drifts, you compute the mean, write the mean of the drifts.

So then, let us say this value be delta m, so then what happens is that all the nodes assume, that they would like to synchronize to a clock, which is delta m shifted from the master, so the master will make an appropriate calculation and send the amounts that they need to shift their clock by to each of the slaves including itself.

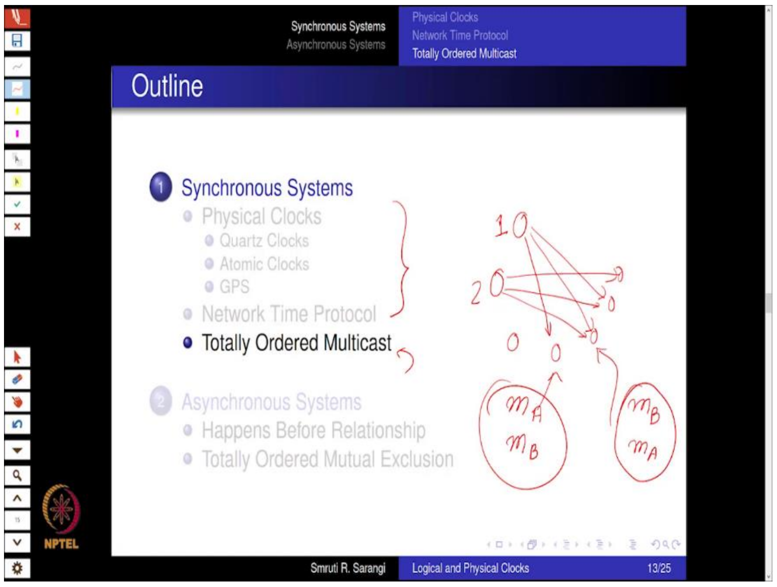
All the slaves including the master will shift their clock, so this will ensure that the clocks of most slaves and the master are relatively synchronized with each other. Furthermore, given the fact that we consider the mean, this will minimize the amount by which each slave needs to adjust its clock. So, basically major adjustments are avoided.

Let me summarize this algorithm in a different manner, what will happen is that, if we have the master and we have a set of slave nodes, different slave notes will have different values of the drift. One option could be that, we say that look the master's clock is correct, all the slaves look this is your drift, you correct your clocks, such that you are totally in sync with the master.

The other is that we say that look none of the clocks are correct, let us instead find the mean value of the drift, then let us assume that the correct value of the clock is essentially t master, which is the current $t_{master} + \Delta_m$, let this be the correct value of the clock. Given the fact that we know that this is the correct value, the master as well as the slaves need to change their clocks, such that all of them, all of their clocks are roughly synchronized to this hypothetical value.

And furthermore, the amounts that they need to actually change their clock is not much because the fluctuations across the mean are not expected to be great, are not expected to be substantial. So, this is one this is like a higher level algorithm, over the Cristian’s algorithm, that uses the Cristian’s algorithm as one of its components to find the drift between each pair, each master slave pair.

(Refer Slide Time: 21:45)



Outline

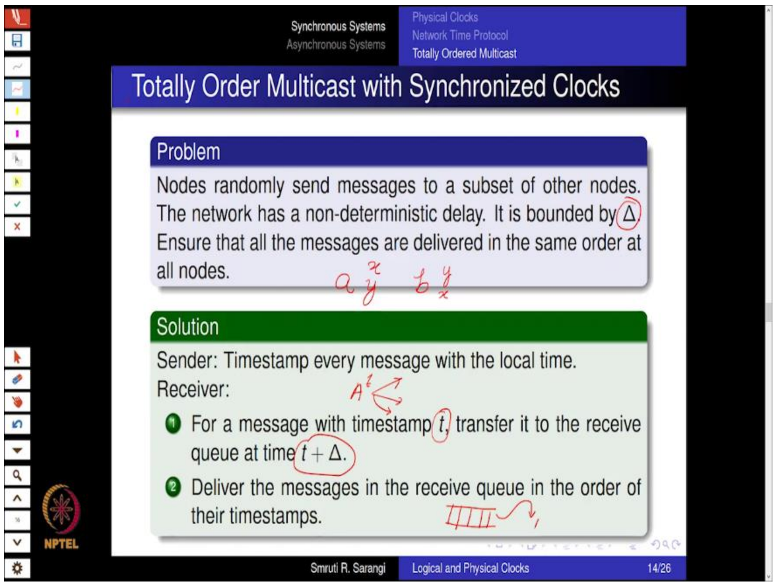
1. Synchronous Systems
   - Physical Clocks
     - Quartz Clocks
     - Atomic Clocks
     - GPS
   - Network Time Protocol
   - Totally Ordered Multicast
2. Asynchronous Systems
   - Happens Before Relationship
   - Totally Ordered Mutual Exclusion

So, now that we have discussed these physical clocks, it is time to discuss a small algorithm with physical clocks or synchronous clocks, we will call it Totally Ordered Multicast. So, what does this mean? What this means is that, if there are a set of nodes, and let us say one node is sending a message to a subset of the other nodes.

Let us say this is sending this message and this is sending, this other node one is sending these messages node, two is sending these messages, so then the order of in which the messages are received should be the same across all the nodes, in the sense it should never be the case that message A from let us say node 1 was received before message B in one of these nodes.

And the other node records different order, which means message B was received first, and message A was received later, that should not happen. So, sending to multiple nodes is called multicast, but ensuring that there is an order for every message is known as total orderly. So, the algorithm here that we are discussing is totally ordered multicast.

(Refer Slide Time: 23:01)



Synchronous Systems
Asynchronous Systems

Physical Clocks
Network Time Protocol
Totally Ordered Multicast

Totally Order Multicast with Synchronized Clocks

Problem

Nodes randomly send messages to a subset of other nodes. The network has a non-deterministic delay. It is bounded by Δ. Ensure that all the messages are delivered in the same order at all nodes.

Solution

Sender: Timestamp every message with the local time.

Receiver:

1. For a message with timestamp t, transfer it to the receive queue at time $t + \Delta$.
2. Deliver the messages in the receive queue in the order of their timestamps.

Smruti R. Sarangi | Logical and Physical Clocks | 14/26

So, the problem is as follows. So, the problem is that we have a system with multiple nodes, they randomly send messages to a subset of other nodes. So, it is important to note that they are not sending a message to a single node, but they are sending a message to a subset of other nodes. Furthermore, the network has a non-deterministic delay, however the delay is bounded by capital delta, and we need to ensure.

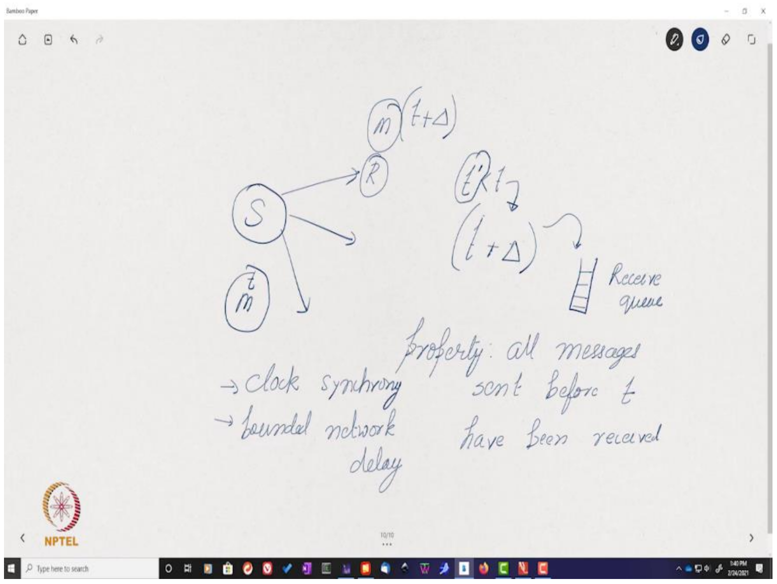
So, there is also one more assumption that we are making, which I am not stating because it has kind of been flowing from the last few slides, which is that we are assuming that their clocks are synchronized. In such a scenario, we want to ensure that all the messages are delivered in the same order at all nodes.

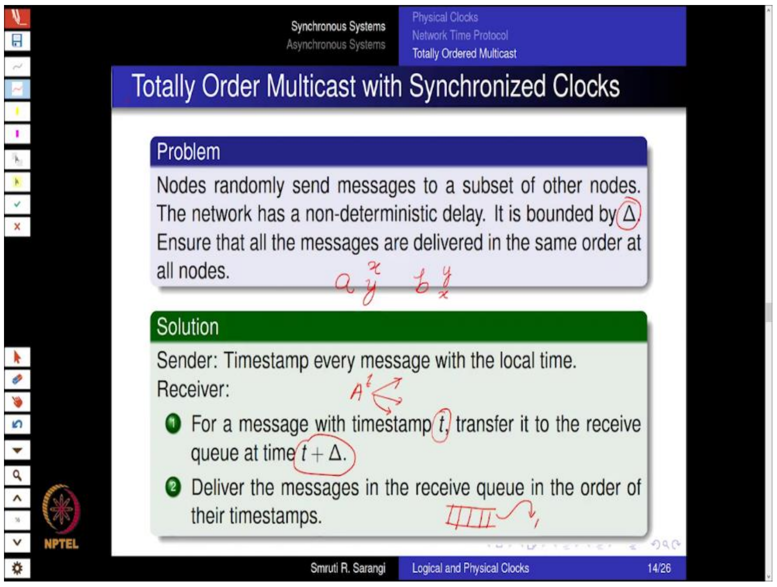
So, as we have just discussed it should never be the case that node A says that message X was delivered before message Y, and node B says that message Y was delivered before message X, so this should never be the case. So, this is not totally ordered, but let us see if all of them see the same order of delivery, it will become totally ordered.

So, what we do is that, we timestamp every message with the local time. So, here of course, the assumption is that the local time is clock synchronized. For the receiver, for a message with timestamp t, all that it does is that it transfers it to the receive queue at time (t + Δ), so why (t + Δ)? well the answer is very simple, so we are assuming that the network has a non-deterministic delay.

This is bounded by Δ or alternatively we can say that the network has a fixed delay but the clocks are not fully synchronized they have a skew of delta, so we can play it in different ways. But the important point is that, if let us say I am receiving the message or let us say I am sending the message at time t, so every message is being time stamped, and let us say the sender is sending the message to a multitude of receivers, it is transmitting the message with its own timestamp which is t.

(Refer Slide Time: 25:50)




Synchronous Systems
Asynchronous Systems
Physical Clocks
Network Time Protocol
Totally Ordered Multicast

Totally Order Multicast with Synchronized Clocks

Problem

Nodes randomly send messages to a subset of other nodes. The network has a non-deterministic delay. It is bounded by Δ. Ensure that all the messages are delivered in the same order at all nodes.

Solution

Sender: Timestamp every message with the local time.
Receiver:

1. For a message with timestamp t, transfer it to the receive queue at time $t+\Delta$.
2. Deliver the messages in the receive queue in the order of their timestamps.

Smruti R. Sarangi   Logical and Physical Clocks   14/26

So, let me show this in a better way. If this is the sender, it is sending the message to a bunch of receivers. The message has a timestamp which is the sender's time which is t, so let us for the time being assume that all the clocks are synchronized, the only jitter is there in the network. Subsequently, at a later point of time, the receiver receives the message.

So, if let us say it waits for all messages, as you can see over here that if let us say it receives, it waits for all messages till the time, $t + \Delta$. So, this will essentially guarantee that all messages sent before time t have reached it. So, what am I saying? What I am saying is that at a time $t + \Delta$, the message is transferred to the receive queue.

The property that I am claiming that all messages sent before have been received, correct, not only have they been received they have also been put in the receiving queue, that is exactly what I am saying? So, why am I saying that, well, the reason I am saying is that let us say consider another message that was sent at time t', so let us assume, let us make a simplistic assumption if you want, I can break it later, where let us assume that the network transmission delay the minimum delay 0 and the maximum is delta.

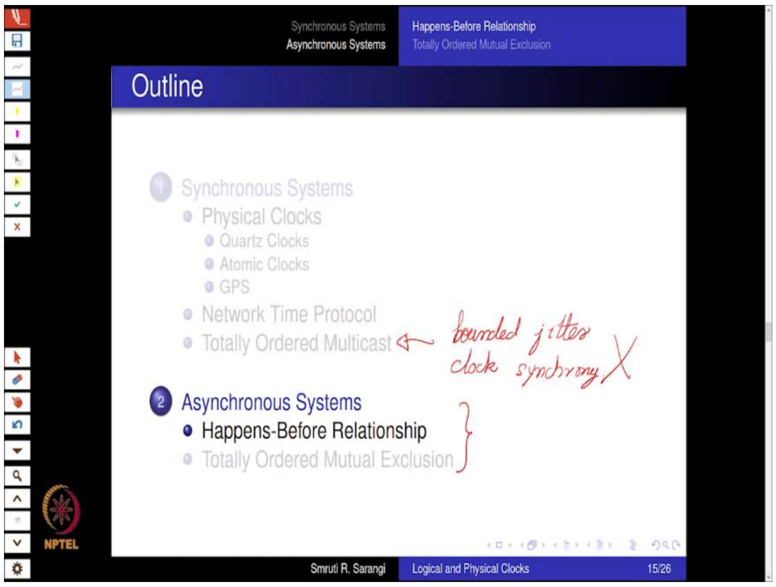
So, in this case what would happen is that let us say I transmit another message at time t' and this is just before t, that is okay, so after delta time units which is the maximum, this message would have been received by all the receivers, and this would automatically guarantee that because of this maximum time which is delta, any message sent before it would also have been received.

And then, so this is why when we add this message with time t to the receive queue at time $t + \Delta$, we are sure that all messages sent to the current receiver with a timestamp less than t are present in the received queue, so that is important. This property, that all messages sent before t to the current receiver have been received and also have been placed in the receive queue, this is important, and this is directly a consequence of two things.

One is clock synchrony, and the other is bounded network delay. So, it is a consequence of these two properties that this is happening. So, given that we have this, what we can see is subsequently we deliver the messages in the receive queue, based on the order of the timestamps.

So, what we do is we take a look at the receive queue and given the fact that every message is time stamped to the timestamp, we deliver the messages from the receive queue to the process, so this is not a first come first serve queue, but it is rather a priority queue, and the priority in this case is the timestamp, so based on the order of the timestamps they are delivered to the receiver process.

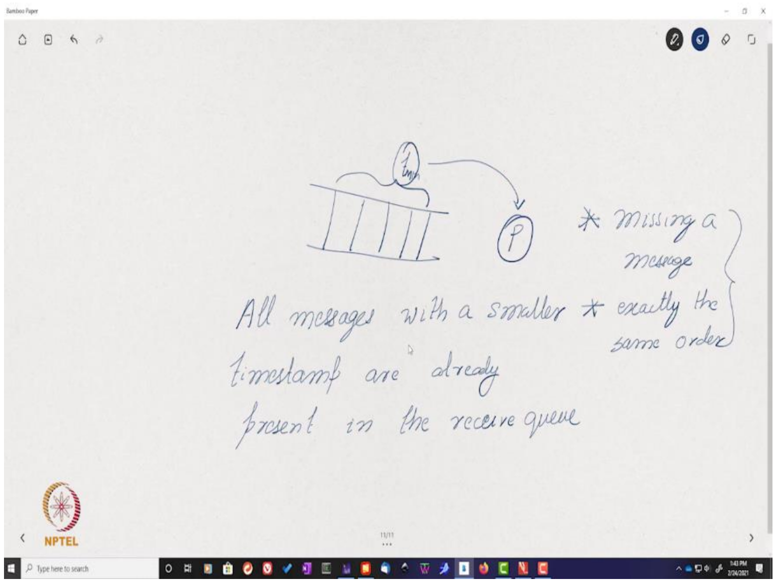
(Refer Slide Time: 30:02)


Synchronous Systems
Asynchronous Systems
Happens-Before Relationship
Totally Ordered Mutual Exclusion

Outline

1. Synchronous Systems
   - Physical Clocks
     - Quartz Clocks
     - Atomic Clocks
     - GPS
   - Network Time Protocol
   - Totally Ordered Multicast ← bounded jitter clock synchrony ✗
2. Asynchronous Systems
   - Happens-Before Relationship
   - Totally Ordered Mutual Exclusion

Smruti R. Sarangi Logical and Physical Clocks 15/26



All messages with a smaller timestamp are already present in the receive queue

* missing a message
* exactly the same order

So, the claim is that this is a totally ordered multicast, so we will see, why? So, the key idea is that let us go to the next. So, the key idea is that if you consider the receive cube, and if you just take a look at the time stamps, what we are saying is that you take the minimum time stamp over here, and let this minimum be let us say tm, tm in, and you deliver this message to the receiving process.

So, let us assume that this is wrong in the sense that there is one more message which should be delivered, and this message has a lower timestamp, it has an earlier timestamp, but that is not possible given the fact that we just proved in the previous sheet, that all the messages with a lower timestamp are already present in the received queue, so it will never be the case that any message with a lower timestamp will not be present in the received queue.

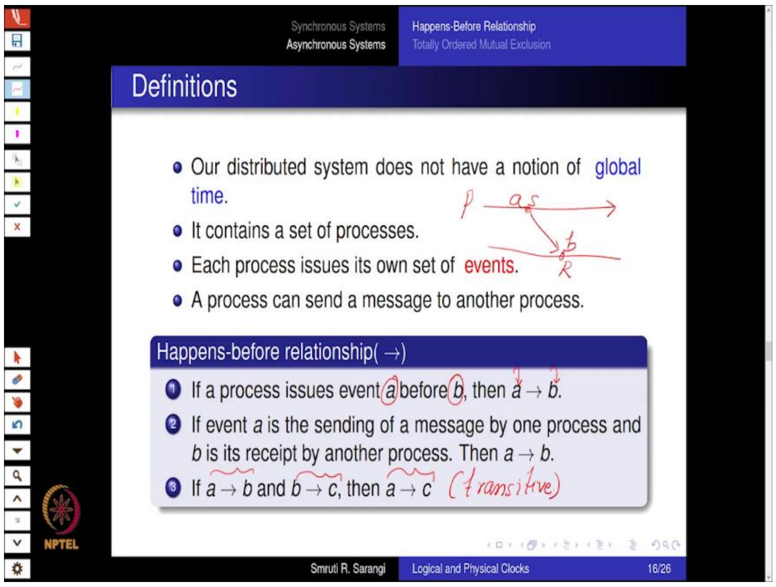
So, let me write that, all messages with let us call it a smaller timestamp are already present in the receive queue. So, given that we have this property, we will essentially be delivering messages to the process in an ascending order of timestamps. So, this is clearly globally ordered, because we are clearly not missing any message.

So we are not dropping in a message, here we are waiting for the message to come, not only the message to come all of his predecessors to come because of the delta property. So, we are not missing a message, this is not happening. Given the fact that we are not missing a message, we are looking at all the relevant messages that should be considered, and we are delivering in exactly the same order, which is the order across nodes which is exactly the same order across nodes, which is essentially the order of increasing timestamps.

So, we are delivering in exactly the same order, hence given these two properties that no message is being missed, and we are delivering in exactly the same order which is an ascending order of timestamps. This proves that we have achieved a totally ordered multicast. Now, the problem is that this did make assumptions on bounded network delay, or let me call it bounded jitter, this did make assumptions about clock synchrony.

So, these assumptions do not hold in practice for a practical distributed system, the reason being that we can have reasonably large indefinite delays in the network as well and along with that that we do not have clock synchrony. So, we need to create some method of a logical time and try to achieve the same thing, maybe not multicast but a slightly simpler problem, and we can of course extend this to multicast, and see how to do it with a logical clock instead of a physical clock use a logical clock, that does not make these assumptions.

(Refer Slide Time: 34:14)


Synchronous Systems
Asynchronous Systems
Happens-Before Relationship
Totally Ordered Mutual Exclusion

Definitions

- Our distributed system does not have a notion of global time.
- It contains a set of processes.
- Each process issues its own set of events.
- A process can send a message to another process.

Happens-before relationship($\rightarrow$)

1. If a process issues event a before b, then $a \rightarrow b$.
2. If event a is the sending of a message by one process and b is its receipt by another process. Then $a \rightarrow b$.
3. If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$ (transitive)

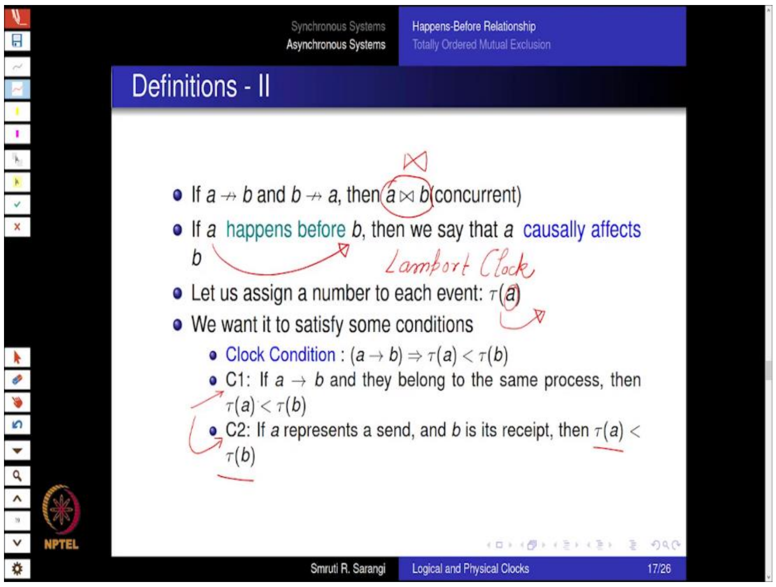
Smruti R. Sarangi Logical and Physical Clocks 16/26

So, given that we do not want to have a notion of global time, what we can do is that we can think of processes, so every process over a timeline has its own events. So, events can either be internal events or event can be as send and the other process can receive, so it can be a send and they receive.

So, let us define a happens before relationship, if a process issues event a before event b in the, within the same process, then we can clearly say that a has happened before b. If event a is the sending of a message by one process and b is its receipt, receiving by another process, then also we can say that look event a → b, because it is not possible to receive event b without a being sent.

So, these are our two happens before relationships, one holds within the same process, and the other holds across processes, and essentially same process we keep on incrementing our events and a send receive, the receive needs to happen after the send. Furthermore, transitivity holds in the sense if a → b, and b → c, we can say that a → c. So, the transitive property does indeed hold. So, we are trying to develop some notion of a logical time which is different from physical time.

(Refer Slide Time: 35:49)


Synchronous Systems
Asynchronous Systems
Happens-Before Relationship
Totally Ordered Mutual Exclusion

Definitions - II

- If $a \nrightarrow b$ and $b \nrightarrow a$, then $a \bowtie b$ (concurrent)
- If a happens before b, then we say that a causally affects b
- Let us assign a number to each event: $\tau(a)$
- We want it to satisfy some conditions
  - Clock Condition : $(a \rightarrow b) \Rightarrow \tau(a) < \tau(b)$
  - C1: If $a \rightarrow b$ and they belong to the same process, then $\tau(a) < \tau(b)$
  - C2: If a represents a send, and b is its receipt, then $\tau(a) < \tau(b)$

Lamport Clock

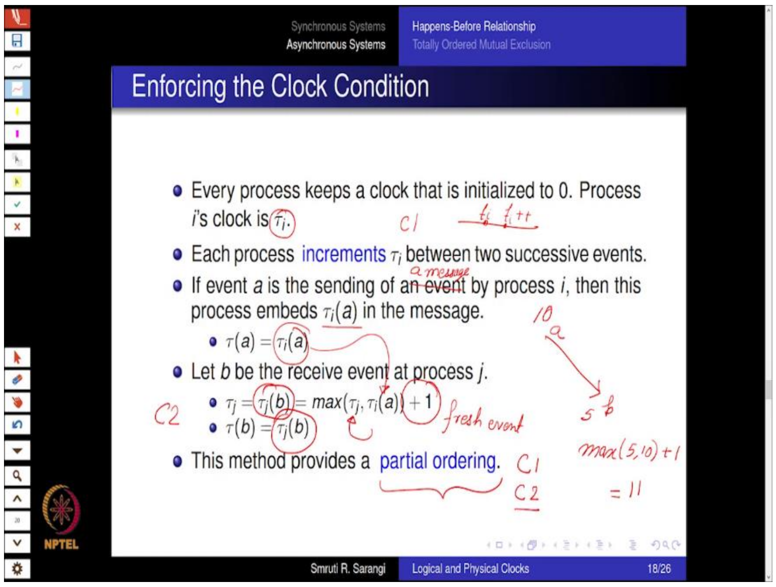
Smruti R. Sarangi Logical and Physical Clocks 17/26

So, now let us look at some definitions, so if it is the case that a has not happened before b, and b has not happened before a, so this of course can happen in a distributed system, it cannot happen in a physical system, but in a distributed system it can definitely happen, so let us say there are two events into on two different processes and though processes do not send any messages between each other, so then it will be the case that look a has not happened before b, and b has not happened before a.

Say a *and* b are said to be concurrent, so we would use this symbol for concurrence. If a happens before b, then we can say that a causally affects b in the sense that there is a causal dependence a happens first a is the cause and b is the effect. So, let us create what is called a lamport clock, so let us assign a number to each event, so let us say for event a, let us assign a number to it and let us call it the lamport clock, it is an integer number, it is a natural number.

So, we wanted to satisfy some conditions, the clock condition, that if $(a \rightarrow b) = \tau (a) < \tau (b)$. So, the clock condition can kind of be broken into two sub conditions, that if a happens before b, and a and b belong to the same process, then obviously $\tau (a) < \tau (b)$, so we need to ensure that. Second, if a represents a send and b is its receipt then also, we need to ensure that the timestamp of the sender is less than the timestamp of the receiver, so these are the two things we need to ensure c1 and c2.

(Refer Slide Time: 37:43)



Synchronous Systems
Asynchronous Systems

Happens-Before Relationship
Totally Ordered Mutual Exclusion

Enforcing the Clock Condition

- Every process keeps a clock that is initialized to 0. Process i's clock is τ_i. C1 t_i t_i++
- Each process increments τ_i between two successive events.
- If event a is the sending of a message ~~an event~~ by process i, then this process embeds $\tau_i(a)$ in the message.
  - $\tau(a) = \tau_i(a)$
- Let b be the receive event at process j.
  - $\tau_j = \tau_j(b) = max(\tau_j, \tau_i(a)) + 1$ fresh event
  - $\tau(b) = \tau_j(b)$
- This method provides a partial ordering. C1 C2

C2

10 a → 5 b

max(5,10)+1 = 11

Smruti R. Sarangi — Logical and Physical Clocks — 18/26

So, reinforcing the clock condition is easy, it is not hard at all. Every process keeps a clock, keeps a clock which is a counter that is initialized to 0. So, let us call process size clock or processor is counter as τi, ti. So then, so let us refer to τi as ti as it is, I will not use the term τ anymore I will call it ti. Each process increments ti between two successive events, so this ensures that clock condition c1, that whenever I have one event and then I have one event after this, if this is ti, I just increment it by 1.

If event a is the sending of an event with the sending of a message, let me just change, that by process I then this process embeds its own timestamp in the message, so what it does is that it embeds its timestamp in the message, and let b be the receive event at process j.

So, what we will do is, we need to ensure that the timestamp of b > timestamp of a, a is the send event and b is the receive event. So, what we do is that we compute a max operation between this time stamp which is a timestamp and b's existing timestamp. So, clearly the final timestamp of b has to be greater than both which is its existing timestamp as well as the timestamp of a, because it is a new event, so this needs to be the case, so we compute a maximum of them and then we add 1 because it is a fresh event.

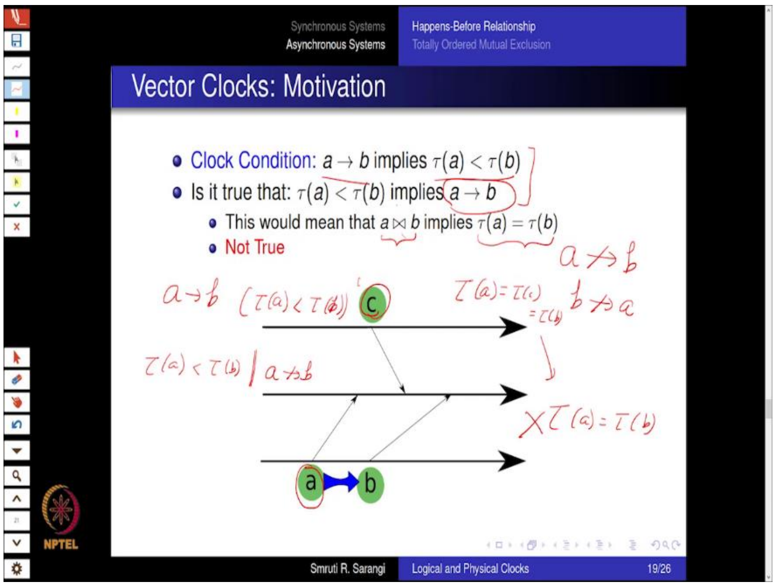
So, one of course comes because it is a fresh event, but the maximum comes because number 1, we do not want to go back in time. So, given the fact that we do not want to go back in time, the timestamp of b will only increase, but the other thing is that if a has a higher timestamp, so let us say a's timestamp = 10 and b = 5.

We first need to compute the maximum of 5 and 10 which of course is 10 and then add 1, because it is a fresh event, so this becomes 11, and 11 becomes the timestamp of event b as well as the timestamp of process j, as well as the time of the, current time of process j. So, this method, which is very simple essentially the two clock conditions c1 and c2, the way that they are being enforced is extremely simple.

So, for c1 what we are doing is that any new event of a process, we are just incrementing the counter by 1 and for c2, what we are doing is anytime a sends a message, anytime a is the send event and b is the receive event, we just ensure that the timestamp of the receiving process is greater than its current timestamp, and the timestamp of the sending process and we add one because it is a new event.

So, this does provide a partial ordering between the events but it does hold both of our clock conditions c1 and c2 and this is called a lamport clock, a simple lamport clock.

(Refer Slide Time: 41:20)



Synchronous Systems
Asynchronous Systems

Happens-Before Relationship
Totally Ordered Mutual Exclusion

Vector Clocks: Motivation

- Clock Condition: $a \rightarrow b$ implies $\tau(a) < \tau(b)$
- Is it true that: $\tau(a) < \tau(b)$ implies $a \rightarrow b$
  - This would mean that $a \bowtie b$ implies $\tau(a) = \tau(b)$
  - Not True

Smruti R. Sarangi · Logical and Physical Clocks · 19/26

$a \rightarrow b$ $(\tau(a) < \tau(b))$
c
$\tau(a) < \tau(b)$ | $a \not\rightarrow b$
$\tau(a) = \tau(c) = \tau(b)$
$a \not\rightarrow b$
$b \not\rightarrow a$
$\times\ \tau(a) = \tau(b)$
a
b
NPTEL

So, it is called a scalar clock, it is a lamport scalar clock. So, what we have seen is, that if a implies b, it implies that the time of a is less than the time of b, what about the converse? if the time of a is less than the time of b does it imply that a happened before b. If this were the case, if this were the case, bear in mind, then if a and b are concurrent events, it would imply that their timestamps are the same.

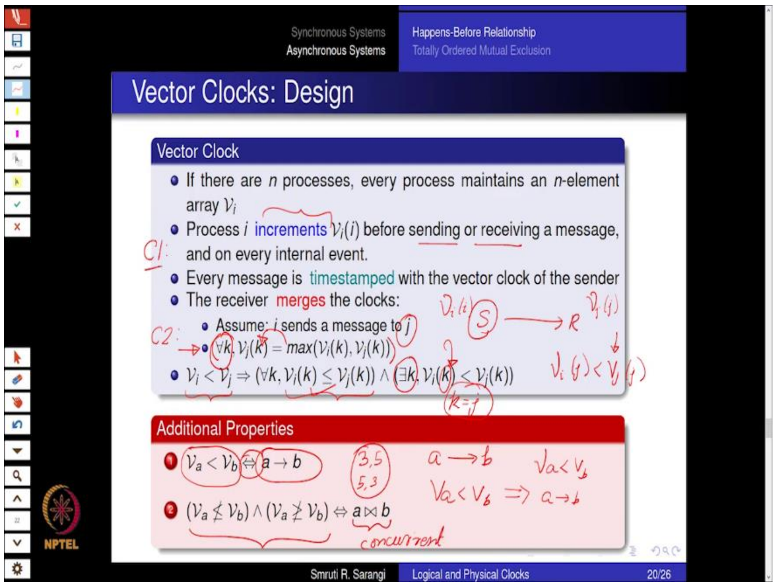
So, a and b are concurrent basically means, that a does not happen before b, b does not happen before a, given the fact that both of these relationships hold, you will clearly see that if a does not happened before b, which this basically means that the timestamp of b is not greater than the timestamp of a, similarly, the T(a) < T(b), and so then they must be equal, but we will see an example where this actually does not hold, in this case, in this case of a scalar clock, this actually does not hold.

So, consider event a and b in the same process, so what you can see is that event a and event c are concurrent, because they are in different processes, and there is no send receive interaction happening between them. Given that they are concurrent; by this assumption their time should be the same.

Again, given that b and c are concurrent by this assumption, the time should be the same which leads us to the fact, that this and this should be the same, this is clearly not the case there is a contradiction, because these are events issued by the same process. So, what we find that, so this essentially proves my contradiction that the converse is not true, which means if a happens before b then, it is correct that a scalar clock would ensure that the T(a) < T(b) but the converse is not true which means that if let us say this is the case.

Then it does not imply, that a happened before b this is clearly not the case, because if this were the case then this would hold and when, and we are clearly seeing that this relationship, for this example is not holding, given the fact that is not holding, we can say that in this case the converse is not true, it is not correct, so we cannot say that. If, let us say two scalar blocks, 1 is less than the other, it would automatically imply that there is a cause effect relationship it happens before relationship, this cannot be implied.

(Refer Slide Time: 44:11)



Synchronous Systems
Asynchronous Systems
Happens-Before Relationship
Totally Ordered Mutual Exclusion

Vector Clocks: Design

Vector Clock

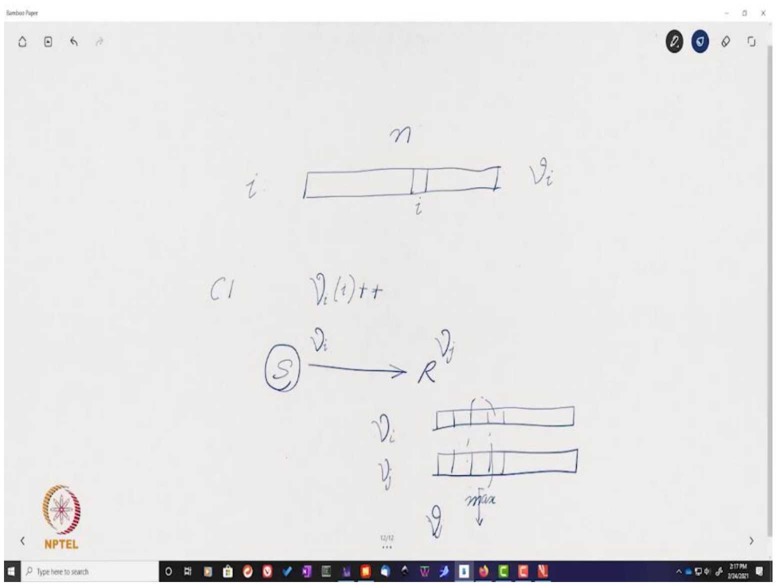
- If there are n processes, every process maintains an n-element array $\mathcal{V}_i$
- Process i increments $\mathcal{V}_i(i)$ before sending or receiving a message, and on every internal event.
- Every message is timestamped with the vector clock of the sender
- The receiver merges the clocks:
  - Assume: i sends a message to j
  - $(\forall k, \mathcal{V}_j(k) = max(\mathcal{V}_i(k), \mathcal{V}_j(k)))$
- $\mathcal{V}_i < \mathcal{V}_j \Rightarrow (\forall k, \mathcal{V}_i(k) \leq \mathcal{V}_j(k)) \wedge (\exists k, \mathcal{V}_i(k) < \mathcal{V}_j(k))$

Additional Properties

1. $\mathcal{V}_a < \mathcal{V}_b \Leftrightarrow a \rightarrow b$
2. $(\mathcal{V}_a \nleq \mathcal{V}_b) \wedge (\mathcal{V}_a \ngeq \mathcal{V}_b) \Leftrightarrow a \bowtie b$

C1:
C2:
$\mathcal{V}_i(i)$ S → R $\mathcal{V}_j(j)$
$\mathcal{V}_i(j) < \mathcal{V}_j(j)$
$k=j$
3,5
5,3
$a \rightarrow b$ $V_a < V_b$
$V_a < V_b \Rightarrow a \rightarrow b$
concurrent

Smruti R. Sarangi Logical and Physical Clocks 20/26



So, what we do is that instead of a scalar clock, we have a vector clock, this solves the converse problem for us. So, every process i has a vector clock, which is basically a vector of scalars, if there are n processes in the system, then we have n elements in this vector. So, within this the private clock of I is also embedded, it is essentially the ith element. So, to ensure condition c1, whenever process I has an internal event, that would include a send or a receive, it would first increment.

So, let us refer to this clock as vi, it would first increment the ith element, on any send receiver any internal event it will do that, and then it will send the message and in this case the message will be time stamped with the vector block, that is important, that is important to bear in mind that the message will be time stamped with the vector clock.

So, what we see here is that, if there are n processes every process maintains an n element array Vi. Process i increments the i-th element before sending or receiving a message or any internal event and every message is time stamped with the vector clock of the sender. Now, the sender sends a message to the receiver, so needless to say before sending the sender would increment its own private clock, so if it is the sender's id is I.

It would increment the ith element of the ith clock, and after process j receives, it would first increment the jth element of its clock, which means its own private clock it will increment, within the array the jth element it will increment. But here is the key operation, the key operation is and I will maybe take you over there.

So, the key operation here is that let us say if the sender is sending something to the receiver, it is attaching its vector clock along with it. Subsequently, the receiver is receiving it, it is updating its own internal vector clock and finally we have a sender's clock, and we have a receiver's clock, appropriately internal values, appropriately incremented. So, we are looking at two vectors.

So, what we do is that we compare them element by element, for every element pair something like this for every element pair we compute the maximum this is similar to what we were doing with the scalar clock we do exactly the same with the vector clock as well where for every element pair we compute the maximum and this maximum is used to compute the final value of the jth processes vector clock.

So, this is shown mathematically over here that, what we do is that for all elements, we compute the maximum of process I's clock and process j's clock element by element maximum and we set it to the kth element or the final value of process j's vector clock. So, instead of the maximum of two numbers, we just compute the pairwise maxima across the vectors similar corresponding elements across the vectors and the final value of the vector clock is just the result of the maximum.

So, then how do we say that one vector clock is less than the other, so by the way before we say that the internal events this is ensuring clock condition 1, this is ensuring clock condition 2, we will see why so we will need to see how because we have still not talked about how 2 vector clocks are less than or more but we are essentially looking at clock condition 2, which is simple, which is straightforward it follows from the definition of maximum, that for all k, if let us say $Vi < Vj \Rightarrow (\forall k, vi(k) \leq vj(k)) \wedge (\exists\ k, vi(k) < vj(k))$.

So, it means that for all values it is never the case, that any value at any cell is more than the corresponding value, in the corresponding cell of Vj(k), that is never the case, it is always less than equal to. But they are not strictly equal, they are not the same, which means there exists some k for which there, exists some k for which Vi(k) < Vj(k).

So, there exists some index for which the kth element, Vi, kth element of Vi is less than the kth element of Vj. For the rest, it is less than equal to for at, but for at least one element it is strictly less than, or in other words I can say that it is less than equal to for all elements but the clocks are not equal. This would translate to the same thing which means there is at least one element where they are unequal and ith element < jth, i’s element, process i's element < j's element.

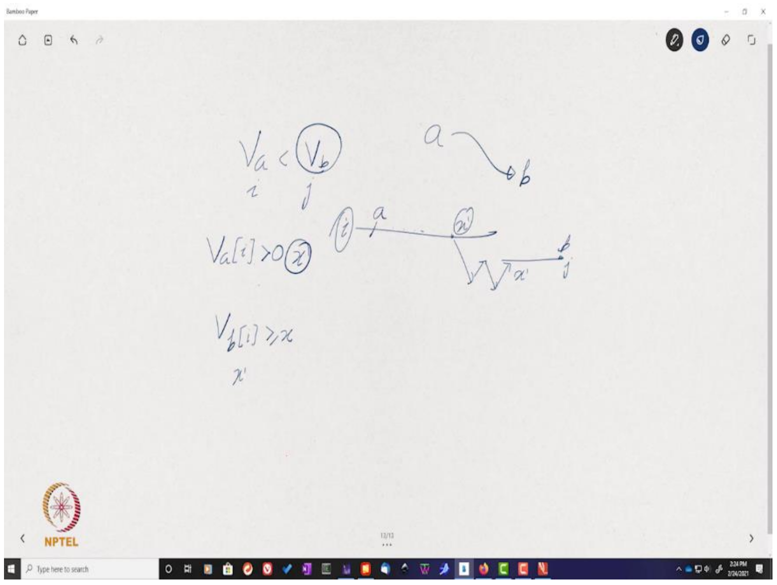
So, this as you can see automatically guarantees clock condition c2, because the mere fact that we are computing a maximum will enforce this condition. And the mere fact that we are incrementing the receiver's clock, that would enforce this condition that you know, so receiver is expected to have the latest version of its own clock of its own counter, and given that that is being incremented, that would automatically enforce this condition, because there is no way, the sender would have the same value for the receiver, for the case when (k = j).

So, there is no way that the sender would have the most recent value of this because the receiver just incremented it just before getting this message. So, at least for this element Vi(k) < Vj(k) or let us put it this way Vi(j) < Vj(j), which is exactly what we ensured by the increment. So, if we just look before every send or receive, we increment and this precisely ensures this condition.

So, given that our clock conditions are satisfied, this does follow the definition of a logical clock, and now we can say that let us say if there is a chain of causality, let us say a happens before b, it can happen before b in two ways one is they are events in the same process, then automatically by clock condition 1, we have Va < Vb, so this is like clock condition 1.

If a is a send and b is a receive, we also have Va < Vb by clock condition 2. Now the key question is, let us say, let us assume we have this does this imply that a happened before b. Well, so it does, so the thing is that it actually does, so the implication is both wise, so the original is true the clock condition is true as well as its converse is also true, and the reasons are not hard to actually think about, so let us prove this.

(Refer Slide Time: 52:24)



So, what we need to prove is that Va < Vb? so there are two events a and b and then we need to say that there is a path causality path of causality between a and b, so that is what we need to prove, because the other direction we have already proven. So, for this of course if a and b if there are no events in between, then it is easy to proof, so that is the trivial case, so let us consider the harder case where there can be any number of events in between.

So, let us now consider, so let us say that this is process i and this is process j. So, let us look for event a, let us look at, so it is essentially event a was issued by process i, so let us look at the ith component of this. See of course, if this is 0 means that this is not an event, so it cannot be 0. So, given the fact that it is an event it has to be associated with a non-zero-time stamp of at least its originating process, so this clearly cannot be 0, but let us say this is some value which is greater than 0, and so let us say that the value of this is equal to x.

So, given the fact that this property holds for process i at least which is the originating process of event a, its ith element is for sure greater than equal to x. So, now the question is how did it get to know? the only way that it would have actually gotten to know so this is a different process, this, so this is process j, and so this is a different process, so how would it how would you ever get to know something about another process.

Given the fact that you can only increment your own private timestamp within the vector clock which is the timestamp corresponding to your own process id, you cannot increment any other process id, the only way to get information is if somebody sends it to you. So, which basically means that the only way that V(b) knows that the id of this, is x or is some value greater than equal to x is only possible.

If let us say there is some event in process I, where a happened first and then maybe something there are some complex series of interaction does not matter but then there was some event that set the value to whatever. So let us say the value of Vb(i) is x', so it says, so it set it to x', and then this value was again communicated could be via a host of processes it does not matter, but finally it was communicated to process j that look the value is x' and clearly x' ≥ x.

So, given the fact that x' was communicated to it there has to be a causal chain of events within the same process and sends and receives that is the only way that event b that is happening in process j, actually knows about the value of the clock in process I, and in particular its ith component.

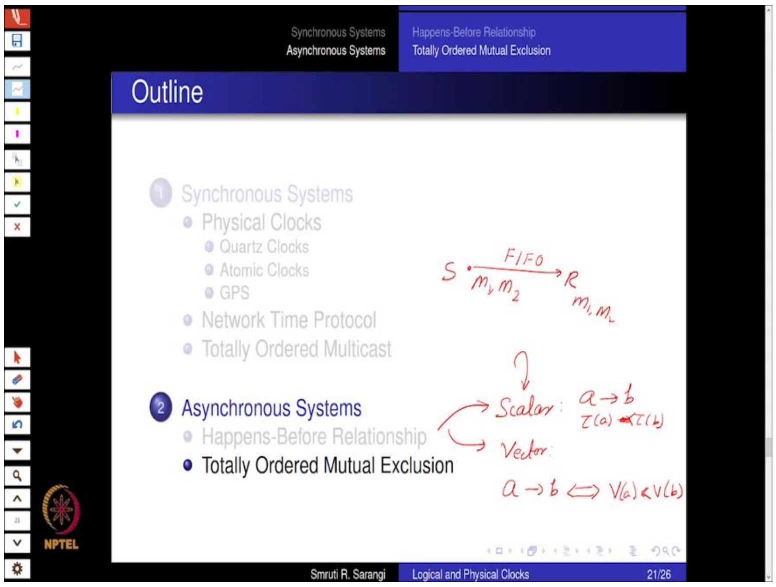
So, it at least has the value x which a is recording or some value which is more than that which can only be possible if it was explicitly communicated to j maybe directly or via other processes it does not matter, but nevertheless there has to be a causal chain otherwise j will simply not ever get to know, and given the fact that j knows that a causal chain does exist and this does prove that if one vector clock is less than another vector clock, you can establish a causal chain of happens before relationships.

So, given the fact that we have actually proven both directions, so we have proven that if let us say there is a causal chain the vector clocks one is lower than the other and if let us say Va < Vb, there is a causal chain between the events a and b, we have proven both directions so we can have an if and only if kind of implication, implication both sides means it is equivalent.

Now, given that it is a vector clock, so let us say you have two processes, so one vector clock could be 3,5 and the other can be 5,3 and this would basically indicate that Va is also not greater than equal to Vb. So, there is no less than equal to relationship and there is no greater than equal to relationship in the sense that these are not comparable.

Given the fact that these are vectors, in a sense these vector clocks are not comparable. So, since the vector clocks are not comparable, in this situation we will say that the events a and b are concurrent. So, this is the definition of concurrence in the world of vector clocks.

(Refer Slide Time: 57:52)


Synchronous Systems
Asynchronous Systems
Happens-Before Relationship
Totally Ordered Mutual Exclusion

Outline

1. Synchronous Systems
   - Physical Clocks
     - Quartz Clocks
     - Atomic Clocks
     - GPS
   - Network Time Protocol
   - Totally Ordered Multicast
2. Asynchronous Systems
   - Happens-Before Relationship
   - Totally Ordered Mutual Exclusion

$S \xrightarrow[m_1, m_2]{FIFO} R$
m_1, m_2
Scalar: $a \to b$
$\tau(a) < \tau(b)$
Vector:
$a \to b \iff V(a) < V(b)$

Smruti R. Sarangi
Logical and Physical Clocks
21/26

So, what vector clocks have actually given us is that as opposed to scalar clocks, if we just implied that look, if there is a causal relationship I can say something about the scalar time of a and the scalar time of b, it is less than vector clock said that look it does not matter, you can infer causality, you can infer causality from the values of the vector clocks themselves.

See, if let us say one vector clock is less than the other vector clock, you can infer causality and if there is causality then of course Va < Vb. So, it did solve the problem of scalar clocks to a certain extent but it is a, its overheads are more, we need to store more and we need to transmit more.

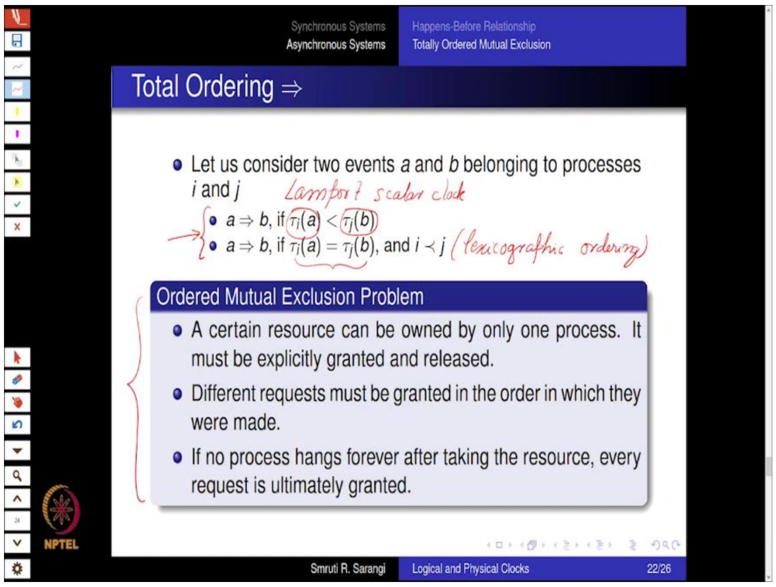
Now, given the fact that we have seen scalar clocks as well as vector clocks, let us look at an algorithm that uses the lamport scalar clocks, and we will call it totally ordered mutual exclusion, which is a different from totally ordered multicast even, though there is a total order over here across the processes.

So, we will solve this using a simpler variant of our clocks and we will show that in a world with non-deterministic network delays which can be very large. In a world without clock synchrony also we can achieve a lot just by using our notion of time which are scalar clocks and, in some cases, we will also end up using vector clocks.

But we will still make one key assumption, which is the fact that we have FIFO channels, which means that if a sender is sending data to a receiver and let us say it is sending two messages, let us call them m1 and m2, if let us say m2, m1 was sent first, and m2 was sent later, the messages will never get reordered in the network, in the sense that receiver will receive m first and m2 next.

And, so basically we will think of these as FIFO first in first out channels where the net network can delay messages and the delay can be in the indefinite, but it will never reorder messages, so this FIFO channel assumption, we will always make.

(Refer Slide Time: 60:20)



Synchronous Systems
Asynchronous Systems
Happens-Before Relationship
Totally Ordered Mutual Exclusion

Total Ordering ⇒

- Let us consider two events a and b belonging to processes i and j Lamport scalar clock
  - $a \Rightarrow b$, if $\tau_i(a) < \tau_j(b)$
  - $a \Rightarrow b$, if $\tau_i(a) = \tau_j(b)$, and $i \prec j$ (lexicographic ordering)

Ordered Mutual Exclusion Problem

- A certain resource can be owned by only one process. It must be explicitly granted and released.
- Different requests must be granted in the order in which they were made.
- If no process hangs forever after taking the resource, every request is ultimately granted.

Smruti R. Sarangi Logical and Physical Clocks 22/26

So, let us now look at the mutual exclusion problem. So, first we will consider what it means for two events a and b to be totally ordered. So, we will go back to our definition of lamport scalar clocks, so we will not consider the vector clock but we will consider the scalar clock. So, what we have said is that the scalar clock has limitations, but as far as we are concerned, we will try to derive some meaning out of it.

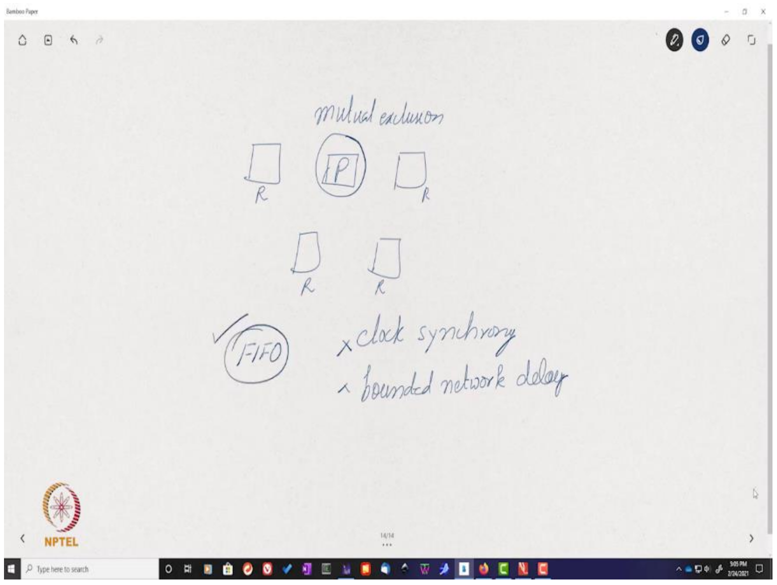
So, we will say, we will say the following of course, this does not this is not completely in consonance with the definition of a scalar clock, but here is what we will say. What we will say is that if let us say a is a send and b is a receive, then of course the clock conditions will hold, and we will further also say that look as far as we are concerned, the converse is true even though we know it not to be true.

But let us assume just for the sake of discussion, that we say that look, we order all the events by their scalar clock time, and if two events have the same scalar clock time, then we order them by the ids of the, we break the ties by the ids of the processes. So, this is kind of an alternative definition of a scalar clock and it does not, technically speaking that we are talking about the converse and technically speaking, it does not hold the way we have defined it.

But let us assume that we have a lamport scalar clock, and the for the purpose of, just for the purpose of ordering, just for the purpose of ordering events I repeat, we say that look even a appears to be before event b. If let us say the clock of a < b, and furthermore event a appears to be before b, if their clocks are the same but process i is less than process j.

So, this is by the way also called lexicographic ordering, so this is like imposing a total order on events when it actually does not exist, but let us assume that we do it and we still use the scalar clock mechanism that we have. So, what are we going to do with it? So we will solve the ordered mutual exclusion problem. So, mutual exclusion basically, I will give you a real world practical example first, and then I will talk about it.

(Refer Slide Time: 63:29)



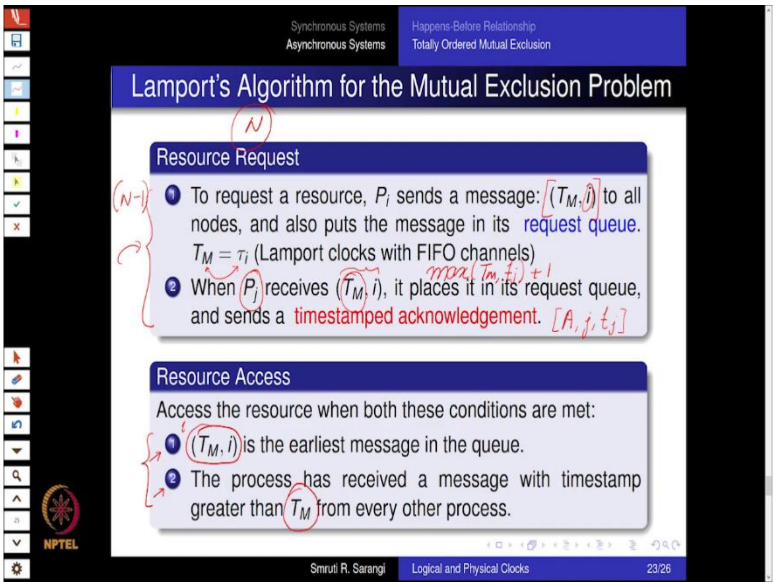
So, let us say that we have a system where we have a set of nodes, so the nodes can be desktops, laptops, does not matter and then there is a shared printer. So, as far as we are concerned only one of the machines in the network can access the shared printer and it can print a page, so this is a resource which cannot be acquired by two machines at the same point of time, it cannot be used concurrently, and furthermore once it is being used, the use has to finish, then only the next use will start.

So, we can say that different machines will have different usage requests they will somehow coordinate between themselves to ensure that the property or mutual exclusion as I just defined it, this holds in the sense only one request can use the printer at a given point of time, it is important that this holds. Furthermore, we will use the mechanism of lamports scalar clocks to create some protocol, where the scalar clocks are incremented the way that we have described.

So, the aim is to create a distributed protocol, we will not assume clock synchrony, so these are the things that we will not assume, we will not assume a bounded network delay. These two things will not assume, but of course we will assume FIFO channels, that is important between any two nodes, we will assume a FIFO channel.

So, with this in mind, let us go back to our mutual exclusion problem, it says a certain resource can be owned by only one process, and it has to be explicitly granted and released. Different requests must be granted in the order in which they were made, which basically means the order by which this total ordering that we have imposed, so this total ordering that we have imposed different requests are made in that, and right so they are consistent with this total ordering, and if no process hangs forever after taking the resource, then every request is ultimately granted, in the sense the protocol is fair.

(Refer Slide Time: 64:45)



Synchronous Systems
Asynchronous Systems
Happens-Before Relationship
Totally Ordered Mutual Exclusion

Lamport's Algorithm for the Mutual Exclusion Problem

N

Resource Request

$(N-1)$

1. To request a resource, P_i sends a message: (T_M, i) to all nodes, and also puts the message in its request queue. $T_M = \tau_i$ (Lamport clocks with FIFO channels)
2. When P_j receives (T_M, i), it places it in its request queue, and sends a timestamped acknowledgement. $[A, j, t_j]$

$\max(T_M, t_j) + 1$

Resource Access

Access the resource when both these conditions are met:

1. (T_M, i) is the earliest message in the queue.
2. The process has received a message with timestamp greater than T_M from every other process.

Smruti R. Sarangi — Logical and Physical Clocks — 23/26

See, if I were to consider this, so let us look at the lamport's algorithm for the mutual exclusion problem. So, the mutual exclusion problem is very generic, so in any distributed system ultimately if let us say 10 nodes are going to coordinate and do something, then they will have to, you know whenever there is a shared resource or they or they want to, let us say access something, so mutual explosion is mainly for access, in the sense they want to access something where only one machine or one process can access at a single point of time, then of course we need this.

And it is a generic mechanism, because let us say that you want to run some other algorithm or let us say you want to centralize the algorithm and you want one machine one process to kind of takeover, then also such kind of mechanisms are required. So, we will discuss mutual exclusion then these algorithms will naturally lead us towards what is called leader election, where out of all the processes we can elect a leader.

And the leader can do something on behalf of the rest. So, we are pretty much going in that direction but as I said mutual exclusion by itself is a very worthy problem to solve, because in many a time in a distributed environment, we do have shared resources, and hence the mutual exclusion property is required.

So, what is our model? our model is very simple, the model is that we have n processes, and okay. So, the model is that we have n processes with us and we have a single resource that needs to be requested for. So, the n processors compete for it and let us assume that a subset of them are interested. So, to request for a resource, this is what needs to be done. To request a resource Pi, process i sends a message and the message is (Tm, i).

So, what is tm? Tm is essentially the lamport scalar clock of the ith process, so it does whatever incrementing etcetera it needs to do and it sends Tm and also it sends its own process id which is i. So, this tuple over here is what is sent, so this is sent to the rest of the processors, every process will send this tuple to, the rest of the processes which is (n – 1) other processes.

So, when process Pj receives this tuple, well, so the first thing is that in any communication that involves scalar clocks, it increments, sees the value and it increments its clock. Using the operation that we have seen which is c2 in the sense it takes max first and then increments, it takes a max, it takes a maximum first the maximum of the clock of i max of Tm.

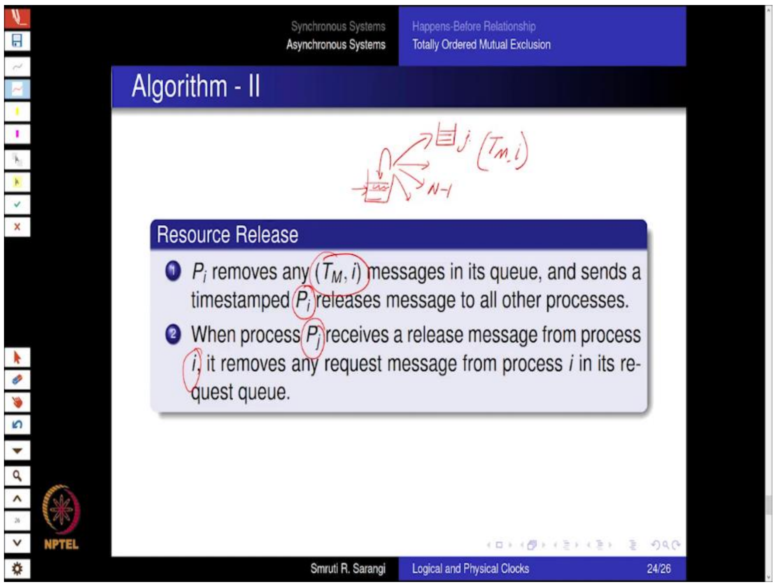
And let us say Tm is d message which is clock of i and then its own clock which is j and then it adds one which is what we have seen and after that after incrementing its local clock it just puts the message in its request queue. Furthermore, it sends a timestamp acknowledgement, and the acknowledgement basically says that look this is an acknowledgement, it is coming from process j and this is the internal time of process j, which has been incremented after receiving the message.

So, you receive the message as per your lamport clock, you take a maximum increment and you send a timestamp acknowledgement. So, this is the first round of the algorithm where a message is sent to n minus one other nodes. Now, to access the resource, what needs to be done? is that we will access the resource when these two conditions are met.

So, what we will do is that, when this message is the earliest message in the queue, so the earliest by the time, in the sense that we look at all the messages in the queue, and so who looks at it, ith process looks at all of the messages in its request cube and let us say Tm i is the earliest, which means either Tm is the lowest time stamp or if there are other ties for the lowest time stamp, then it will break the tie by the process id and if that if i is the smallest, then it becomes the earliest message in the queue, so this is condition 1.

Condition 2 is that the process has received a message with timestamp greater than Tm from every other process. Every other process has basically sent a reply to i, so it could be another message as well from every other process, so every other process has basically sent it a message with a timestamp which is greater than the timestamp of the message, which is Tm. See, if both of these conditions hold, then we claim that it is safe to access the resource.

(Refer Slide Time: 71:28)


Synchronous Systems
Asynchronous Systems
Happens-Before Relationship
Totally Ordered Mutual Exclusion

Algorithm - II

Resource Release

1. P_i removes any (T_M, i) messages in its queue, and sends a timestamped P_i releases message to all other processes.
2. When process P_j receives a release message from process i, it removes any request message from process i in its request queue.

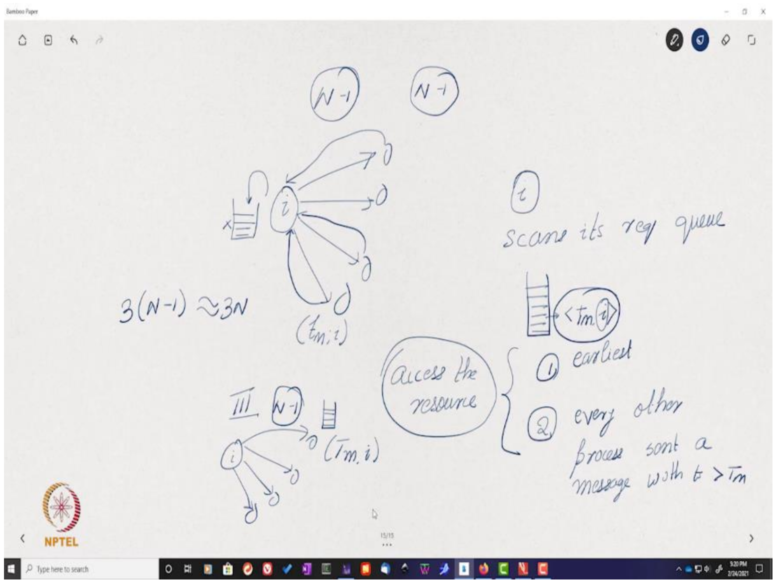
NPTEL
Smruti R. Sarangi
Logical and Physical Clocks
24/26

Then, what is the protocol for release? Well, the protocol for release is that after the resource has been accessed, then what happens is, that Pi scans its own request queue, removes the (Tm,i) the message that it sent, because recall that it sends a message to everybody on the network (n – 1) nodes, and also adds the message to its own request queue.

So, this own message over here is removed. Furthermore, it sends a stamped release message to all the other processes, to the rest of the processes. It sends a release message which basically says that I am releasing my current claim, so this request henceforth stands released. So, when process Pj receives a release message from process i, it removes any request from process i in its request queue.

So, this basically ensures that the original message that it had sent to let us say process j which was (Tm,i), so this message once the release message is sent the message is removed. So, the protocol is very simple, the protocol as such is not hard, so let us go back and discuss this protocol for a second.

(Refer Slide Time: 72:58)



So, the protocol as it sounds is very simple, we have let us say process I, if it wants to request then it sends a time stamped message. So, let us say it is its own personal timestamp (Ti,i), but given that Ti represents, we can use the term Tm for the message, so it sends it to everybody, the rest of the nodes. Along with that, it adds the message to its own request queue over here, and so these are (n – 1) messages.

Subsequently, each of them they increment their internal lamport clock and they send a reply back, they send a reply back. So, this is the second round where again (n – 1) messages are sent which are all the acknowledgements, all the acts. Subsequently, what happens is process i, scans its request queue. So, after scanning its request queue, here is what it does, so it takes a look at all of the messages that are there in its request.

So, it just keeps keeps on doing that, so within its queue it tracks the message that it had sent for getting access to the resource. The moment this becomes the earliest message, earliest message means this is the one with the least time stamp, and if there are multiple messages with the same Tm timestamp, i will be the smallest, so that is the way we break ties, so it waits for the time when this becomes the one with the least time stamp.

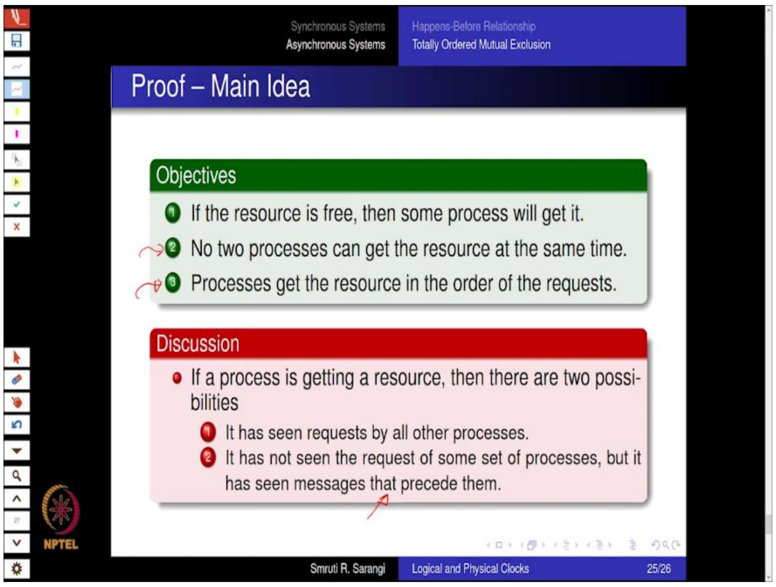
So, this is condition 1 and second is it should have received a reply, from every other node, from the rest of the (n – 1) nodes as you can see over here, it would have received a reply, a message with a timestamp greater than Tm from every other process. So, every other process would have sent it a message greater than Tm, and so once the we can say that every other process sent a message with timestamp greater than Tm, it, so it is not greater than equal to it is greater than Tm.

Once both these conditions hold, we claim that it is safe to access the resource, and we also claim that concurrent accesses to the resource are not possible, so this is the claim that we are making. Once this is done, nothing much needs to be done, you just throw out that original message from here and then again send another message called a release message.

So, the phase three is that i again sends a release message to the, all the node, saying that look you kindly remove the message that I had sent which was (Tm, i) from your internal queues, so from your internal request queue. You will have the (Tm, i) message, you kindly remove that, so this is again (n – 1) messages.

So, as I had said in a distributed algorithm the complexity is measured by the number of messages that needs to be that are sent. The reason being that internal computations are very fast and network messages are expensive. So, the total number of messages we need to send at 3(n – 1) ≈ 3N for large n of course. So, hopefully the protocol is clear now by 2 to all of you, the important thing is why does it work, so the proof is interesting, so that is the important thing, why does it work.

(Refer Slide Time: 77:37)



Synchronous Systems
Asynchronous Systems
Happens-Before Relationship
Totally Ordered Mutual Exclusion

Proof – Main Idea

Objectives

1. If the resource is free, then some process will get it.
2. No two processes can get the resource at the same time.
3. Processes get the resource in the order of the requests.

Discussion

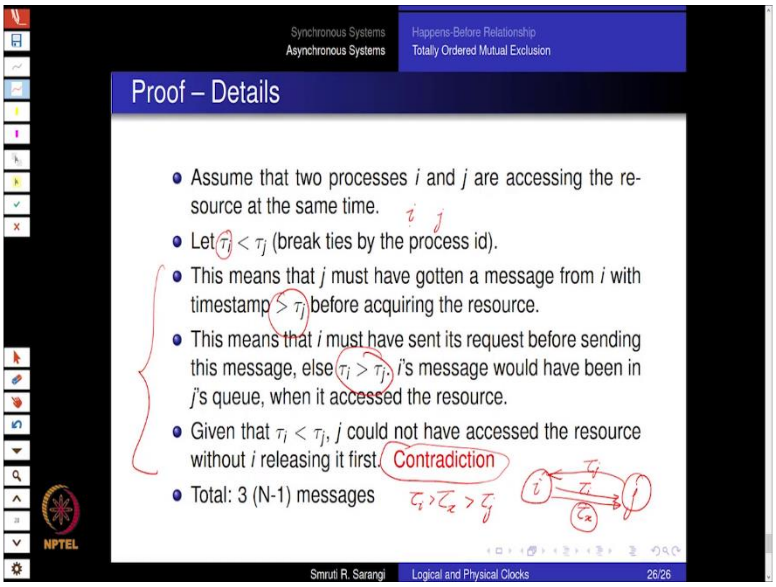
- If a process is getting a resource, then there are two possibilities
  1. It has seen requests by all other processes.
  2. It has not seen the request of some set of processes, but it has seen messages that precede them.

Smruti R. Sarangi   Logical and Physical Clocks   25/26

So, we will discuss the proof shortly, so the main will first, discuss the main idea, then I will take you into the details. So, the main idea is if the resource is free, so let us say nobody else is accessing it, then immediately everybody will get the request, they will send an acknowledgement, and then the original process will immediately start accessing the resource, so that is clear.

What we are claiming is no two processes can get the resource at the same time, and further more processes get requests in order of the request, the order is the one, the lexicographic order that we are imposing, the order that we are imposing on the scalar timestamps. So, the discussion is, if a process is getting a resource, then there are two possibilities. What is the possibility? it has seen requests by all other processes, it has not seen the request of some set of processes but it has seen messages that precede them. So, let us go into detail of what exactly these mean.

(Refer Slide Time: 78:50)



Synchronous Systems
Asynchronous Systems

Happens-Before Relationship
Totally Ordered Mutual Exclusion

Proof – Details

- Assume that two processes i and j are accessing the resource at the same time.
- Let $\tau_i < \tau_j$ (break ties by the process id).
- This means that j must have gotten a message from i with timestamp $> \tau_j$ before acquiring the resource.
- This means that i must have sent its request before sending this message, else $\tau_i > \tau_j$. i's message would have been in j's queue, when it accessed the resource.
- Given that $\tau_i < \tau_j$, j could not have accessed the resource without i releasing it first. Contradiction
- Total: 3 (N-1) messages

$\tau_i > \tau_x > \tau_j$

Smruti R. Sarangi — Logical and Physical Clocks — 26/26

So, we are now trying to get into the depths of the proof and this is the last side. So, let us now go into the details of the proof, so assume we have two processes i and j and they are accessing the resource at the same time, so this of course is not possible, but let us assume to the contrary that it has indeed happened. So, we are talking of two processes i and j and let the requests that led to this simultaneous access have time stamps Ti and Tj, respectively.

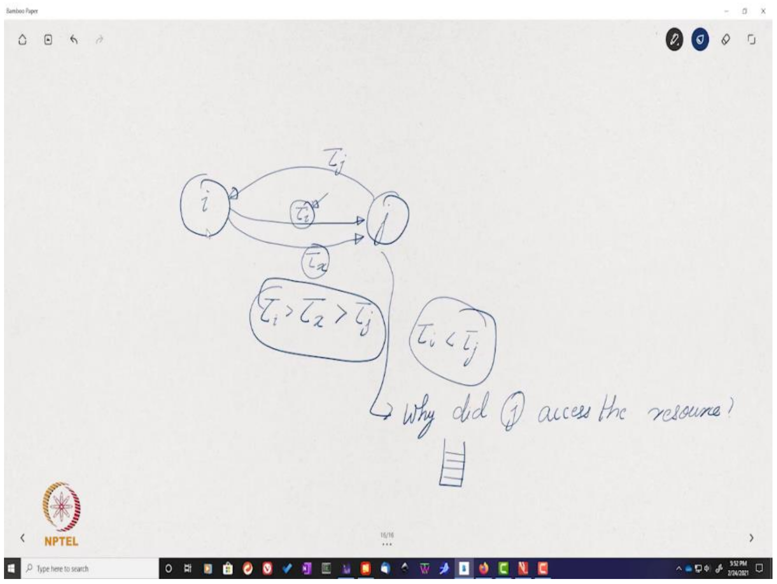
So, with no loss of generality, let us assume that Ti < Tj which means that we break the tie by the process i. So, even if the lamport clocks are the same, we break that tie by the process id but regardless of how we do it. The important point is we have a way of totally ordering the tuples which is the lamport clock timestamp, process id, we do that and it turns out that Ti < Tj with no loss of generality.

So, what would have happened, this means that j must have gotten a message from i with a timestamp greater than Tj before acquiring the resource, otherwise by the second condition, j could not have acquired the resource, so it must have sent the messages etcetera, all of that would have happened, but then I must have sent it a message with a timestamp, which is greater than Tj.

If that is the case, then what about the original request, that i actually sent. So, what is happening? j is sending a message with a timestamp Tj, i is sending a message back with timestamp let us say Tx, tau Tx and we know that this relationship holds, otherwise j could not have accessed the resource.

Now, what about Ti, when i sent its original message? so this could not have been sent after this event, otherwise Ti would have been greater than Tx which would have been greater than Tj, but that is not the case. So, then this basically means that i sent its message to j, before sending this message, so before that i actually sent Ti. If i did that, then let us look at the fun part, so let us go over here.

(Refer Slide Time: 81:25)



So, what we are saying is that we are considering the moment in which j actually entered the critical section, that is what we call in programming terminology or access the resource. This essentially means the following, it means that at some point of time j would have sent i a message Tj, i would have sent a message back at some point of time let us call it Tx, for j to have access the resource, Ti > Tx > Tj.

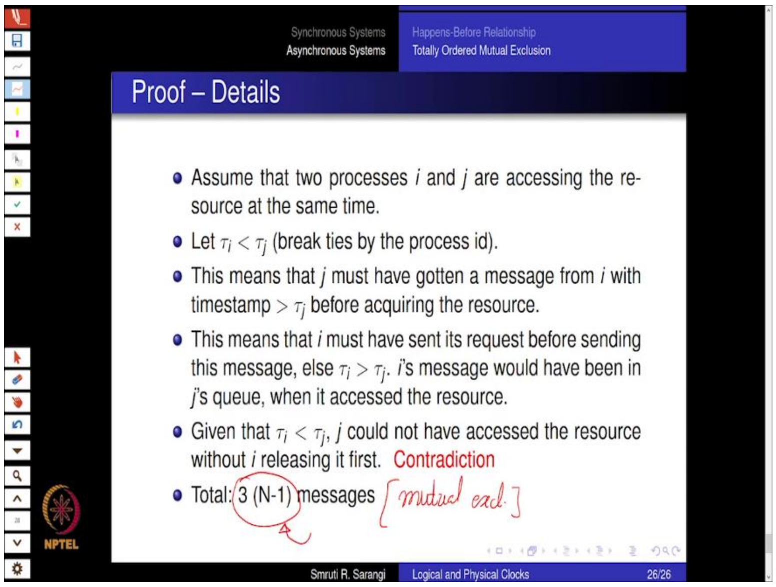
Now, what we are arguing is that the original message with i sent to j, could not have been after sending this message because otherwise i would have had a greater timestamp, which would have been greater than Tx > Tj, but we know that this relationship is not true, because we know that this holds by our assumption. Hence, it means that i would have sent a message to j prior to sending this message, which is Ti which contains its request.

This further means that when j took the decision to actually acquire the resource, it would have had the message from i. So, this is an important thing, let me repeat, bear this in mind, the moment that j took the decision to acquire the resource, it is guaranteed that it would have had the message from i, and given that i and j are accessing it currently, this message would not have been released.

Now, the question arises why did j access the resource it was not supposed to, because, so let me write it, it was not supposed to access the resource, the reason is that in its queue the request from itself was not the earliest request there was already a request from i and that was the earliest request. Consequently, j should not have accessed the resource, hence what we observe is there is a contradiction.

So, our assumption that two requests are actually, they have actually acquired the resource at the same point of time this is not correct, this has led to a contradiction, so this is not possible, and given the fact that a wrong thing is not possible the right thing would be holding all the time which means that mutual exclusion actually holds.

(Refer Slide Time: 84:00)



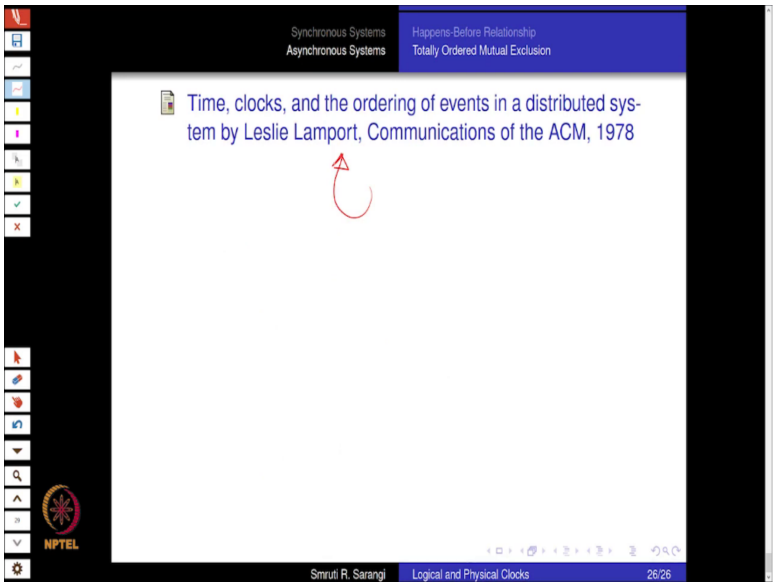
Proof – Details

- Assume that two processes *i* and *j* are accessing the resource at the same time.
- Let $\tau_i < \tau_j$ (break ties by the process id).
- This means that *j* must have gotten a message from *i* with timestamp $> \tau_j$ before acquiring the resource.
- This means that *i* must have sent its request before sending this message, else $\tau_i > \tau_j$. *i*'s message would have been in *j*'s queue, when it accessed the resource.
- Given that $\tau_i < \tau_j$, *j* could not have accessed the resource without *i* releasing it first. Contradiction
- Total: 3 (N-1) messages [mutual excl.]

So, this is what it means, that mutual exclusion holds and the algorithm is correct. So, what did we do in this lamport's algorithm, we sent 3 (n – 1) messages and for sure without assuming clock synchrony or bounded network delay mutual exclusion is guaranteed. And the reason it is guaranteed is because of the way that we constructed our lamport clocks, so this is the key point over here that I am trying to make is that given the fact that we said j has to wait for a message greater than its timestamp from i, i would have sent its request message prior to that.

Given the FIFO channel property that would be there in j's request queue, so you might want to hear this part of the video again. The message would be there in j's request queue, so there is no way on earth that j could have accessed the resource without i releasing it. So, concurrent access is not possible consequently mutual exclusion is guaranteed. So, what we will do in the next lecture is look at many more algorithms that really, that reduce the number of messages that are required to guarantee mutual exclusion.

(Refer Slide Time: 85:26)



Synchronous Systems
Asynchronous Systems

Happens-Before Relationship
Totally Ordered Mutual Exclusion

Time, clocks, and the ordering of events in a distributed system by Leslie Lamport, Communications of the ACM, 1978

Smruti R. Sarangi | Logical and Physical Clocks | 26/26

So, the original paper on time blocks and ordering of events was a seminal paper in distributed systems, pretty much something that started the field, this was published in 1978. You can read up this paper, so this will give you a lot of insights and understanding about the original ideas that came to define the field.