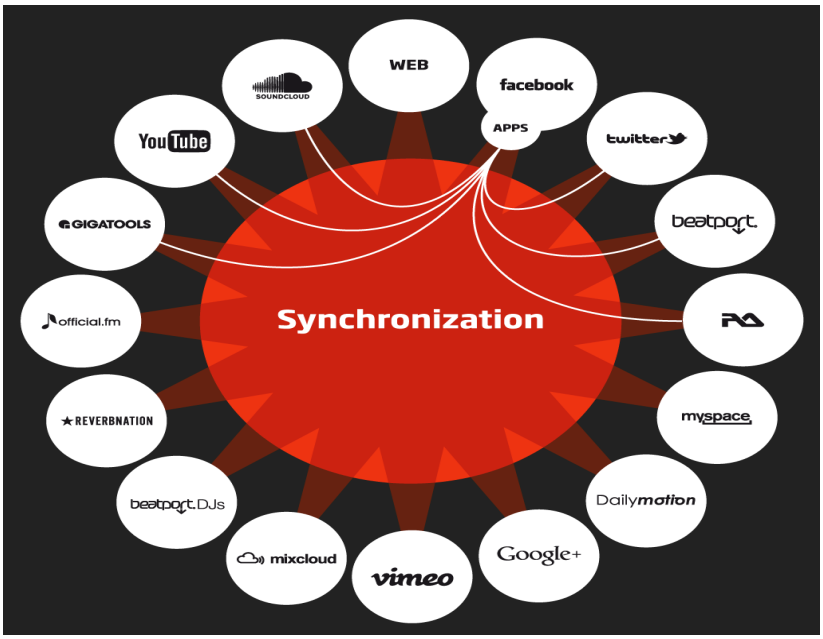
MODULE-3: Synchronization

VIT | Vidyalankar Institute of Technology
Accredited A+ by NAAC



Prepared by Prof. Amit K. Nerurkar

Certificate

This is to certify that the e-book titled "SYNCHRONIZATION" comprises all elementary learning tools for a better understating of the relevant concepts. This e-book is comprehensively compiled as per the predefined eight parameters and guidelines.

Signature

Date: 20-03-2020

Prof. Amit K. Nerurkar

Assistant Professor

Department of Computer Engineering

⚠ ***DISCLAIMER:*** *The information contained in this e-book is compiled and distributed for educational purposes only. This e-book has been designed to help learners understand relevant concepts with a more dynamic interface. The compiler of this e-book and Vidyalankar Institute of Technology give full and due credit to the authors of the contents, developers and all websites from wherever information has been sourced. We acknowledge our gratitude towards the websites YouTube, Wikipedia, and Google search engine. No commercial benefits are being drawn from this project.*

Module 3 Synchronization

Clock Synchronization, Logical Clocks, Election Algorithms, Mutual Exclusion, Distributed Mutual Exclusion-Classification of mutual Exclusion Algorithm, Requirements of Mutual Exclusion Algorithms, Performance measure, Non Token based Algorithms: Lamport Algorithm, Ricart–Agrawala's Algorithm, Maekawa's Algorithm

CLOCK SYNCHRONIZATION

Q.1 Define Clock Synchronization.

(A) In a centralized system, time is unambiguous. When a process wants to know the time, it makes a system call and the kernel tells it. If process A asks for the time, and then a little later process B asks for the time, the value that B gets will be higher than (or possibly equal to) the value A got. It will certainly not be lower. In a distributed system, achieving agreement on time is not trivial.

Just think, for a moment, about the implications of the lack of global time on the UNIX make program, as a single example. Normally, in UNIX, large programs are split up into multiple source files, so that a change to one source file only requires one file to be recompiled, not all the files. If a program consists of 100 files, not having to recompile everything because one file has been changed greatly increases the speed at which programmers can work.

The way make normally works is simple. When the programmer has finished changing all the source files, he starts make, which examines the times at which all the source and object files were last modified. If the source file input, c has time 2151 and the corresponding object file input.o has time 2150, make knows that input.c has been changed since input.o was created, and thus input.c must be recompiled. On the other hand, if output.c has time 2144 and output.o has time 2145, no compilation is needed here. Thus make goes through all the source files to find out which ones need to be recompiled and calls the compiler to recompile them.

Now imagine what could happen in a distributed system in which there was no global agreement on time. Suppose that **output.o** has time 2144 as above, and shortly thereafter **output.c** is modified but is assigned time 2143 because the clock on its machine is slightly behind, as shown in Figure 1 below. Make will not call the compiler. The resulting executable binary program will then contain a mixture of object files from the old sources and the new sources.

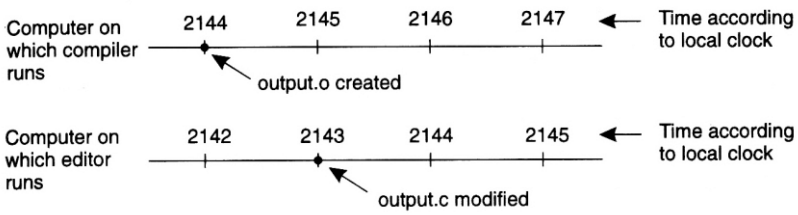


Fig. 1 : When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

Physical Clocks

Nearly all computers have a circuit for keeping track of time. Despite the widespread use of the word "clock" to refer to these devices, they are not actually clocks in the usual sense. **Timer** is perhaps a better word. A computer timer is usually a precisely machined quartz crystal. When kept under tension, quartz crystals oscillate at a well-defined frequency that depends on the kind of crystal, how it is cut, and the amount of tension. Associated with each crystal are two registers, a **counter** and a **holding register**. Each oscillation of the crystal decrements the counter by one. When the counter gets to zero, an interrupt is generated and the counter is reloaded from the holding register. In this way, it is possible to program a timer to generate an interrupt 60 times a second, or at any other desired frequency. Each interrupt is called one **clock tick**.

As soon as multiple CPUs are introduced, each with its own clock, the situation changes. Although the frequency at which a crystal oscillator runs is usually fairly stable, it is impossible to guarantee that the crystals in different computers all run at exactly the same frequency. In practice, when a system has n computers, all n crystals will run at slightly different rates, causing the (software) clocks gradually to get out of synch and give different values when read out. This difference in time values is called clock skew. As a consequence of this clock skew, programs that expect the time associated with a file, object, process, or message to be correct and independent of the machine on which it was generated (i.e., which clock it used) can fail, as we saw in the make example above.

In some systems (e.g., real-time systems), the actual clock time is important. For these systems external physical clocks are required. For reasons of efficiency and redundancy, multiple physical clocks are generally considered desirable, which yields two problems: (1) How do we synchronize them with real-world clocks, and (2) How do we synchronize the clocks with each other ?

Before answering these questions, let us digress slightly to see how time is actually measured. It is not nearly as simple as one might think, especially when high accuracy is required. Since the invention of mechanical clocks in the 17th century, time has been measured astronomically. Every day, the sun appears to. rise on the eastern horizon, climbs to a maximum height in the sky, and sinks in the west. The event of the sun's reaching its highest apparent point in the sky is called the transit of the sun. This event occurs at about noon each day. The interval between two consecutive transits of the sun is called the solar day. Since there are 24 hours in a day, each containing 3600 seconds, the solar second is defined as exactly 1786400th of a solar day. The geometry of the mean solar day calculation is shown in Figure 2 below.

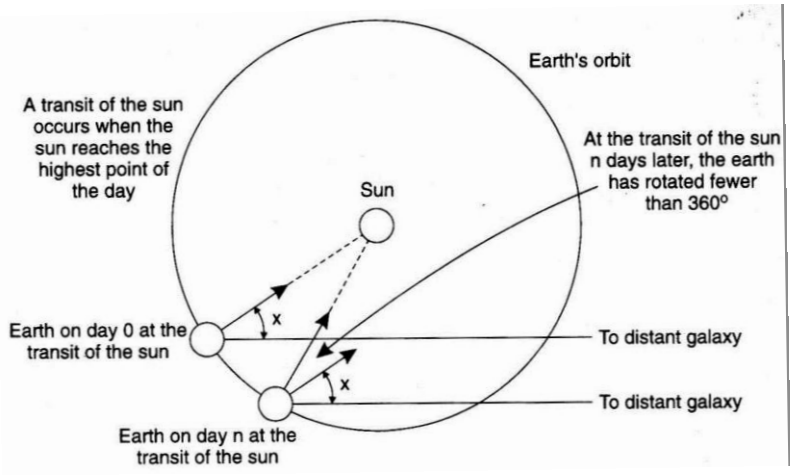


Fig. 2 : Computation of the mean solar day

Clock synchronization algorithms

Q.2 Discuss Clock synchronization algorithms.

(A) Cristian's algorithm let us start with an algorithm that is well suited to systems in which one machine has a receiver and the goal is to have all the other machines stay synchronized with it. Let us call the machine with the WWV receiver a time server. Our algorithm is based on the work of Cristian and prior work. Periodically, certainly no more than every $\delta/2p$ seconds, each machine sends a message to the time server asking for the current time. That machine responds as fast as it can with a message containing its current time, C_{UTC}, as shown in Figure 1.

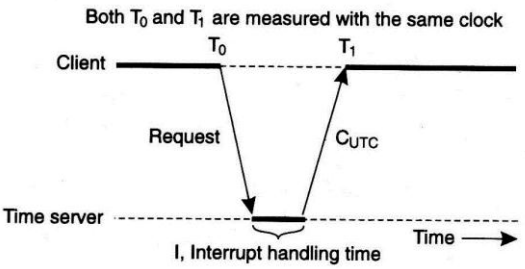


Fig. 1 : Getting the current time from a time server

As a first approximation, when the sender gets the reply, it can just set its clock to C_{UTC}. However, this algorithm has two problems, one major and one minor. The major problem is that time must never run backward. If the sender's clock is fast, C_{UTC} will be smaller than the sender's current value of C. Just taking over C_{UTC} could cause serious problems such as an object file compiled just after the clock change having a time earlier than the source which was modified just before the clock change.

Such a change must be introduced gradually. One way is as follows. Suppose that the timer is set to generate 100 interrupts per second. Normally, each interrupt would add 10 msec to the time.

When slowing down, the interrupt routine adds only 9 msec each time until the correction has been made. Similarly, the clock can be advanced gradually by adding 11 msec at each interrupt instead of jumping it forward all at once.

The minor problem is that it takes a nonzero amount of time for the time server's reply to get back to the sender. Worse yet, this delay may be large and vary with the network load. Cristian's way of dealing with it is to attempt to measure it. It is simple enough for the sender to record accurately the interval between sending the request to the time server and the arrival of the reply. Both the starting time, T_0, and the ending time, T_1, are measured using the same clock, so the interval will be relatively accurate even if the sender's clock is off from UTC by a substantial amount.

In the absence of any other information, the best estimate of the message propagation time is $(T_1 - T_0)/2$. When the reply comes in, the value in the message can be increased by this amount to give an estimate of the server's current time. If the theoretical minimum propagation time is known, other properties of the time estimate can be calculated.

This estimate can be improved if it is known, approximately how long it takes the time server to handle the interrupt and process the incoming message. Let us call the interrupt handling time I. Then the amount of the interval from T_0 to T_1 that was devoted to message propagation is $T_1 - T_0 - 1$, so the best estimate of the one-way propagation time is half this. Systems do exist in which messages from A to B systematically take a different route than messages from B to A, and thus have a different propagation time, but we will not consider such systems here.

To improve the accuracy, Cristian suggested making not one measurement but a series of them. Any measurements in which $T_1 - T_0$ exceeds some threshold value are discarded as being victims of network congestion and thus are unreliable. The estimates derived from the remaining probes can then be averaged to get a better value. Alternatively, the message that came back fastest can be taken to be the most accurate since it presumably encountered the least traffic underway and therefore is the most representative of the pure propagation time.

The Berkeley Algorithm

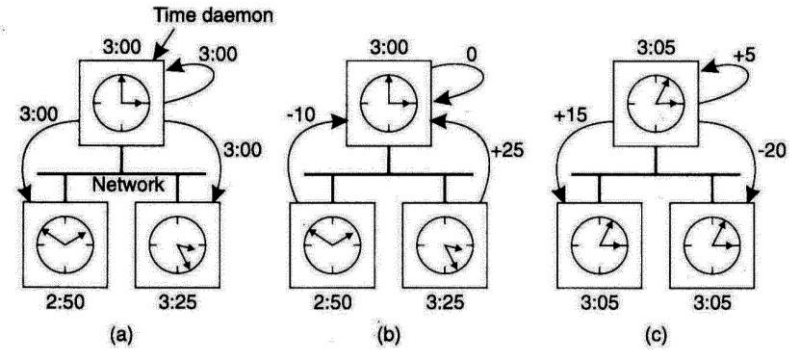


Fig. 2: (a) The time daemon asks all the other machines for their clock values. (b) The machines answer. (c) The time daemon tells everyone how to adjust their clock.

In Cristian's algorithm, the time server is passive. Other machines periodically ask it for the time. All it does is respond to their queries. In Berkeley UNIX, exactly the opposite approach is taken. Here the time server is active, polling every machine from time to time to ask what time it is there. Based on the answers, it computes an average time and tells all the other machines to advance their clocks to the new time or slow their clocks down until some specified reduction has been achieved. This method is suitable for a system in which no machine has a WWV receiver. The time daemon's time must be set manually by the operator periodically. The method is illustrated in figure 2.

In Figure 2 above, at 3:00, the time daemon tells the other machines its time and asks for theirs. In Figure 4 above, they respond with how far ahead or behind the time daemon they are. Armed with these numbers, the time daemon computes the average and tells each machine how to adjust its clock.

Averaging Algorithms

Both of the methods described above are highly centralized with the usual disadvantages. Decentralized algorithms also exist. One class of decentralized clock synchronization algorithms works by dividing time into fixed-length resynchronization intervals. The ith interval starts at $T_0 + iR$ and runs until $T_0 + (i + 1)R$, where T_0 is an agreed-upon moment in the past, and R is a system parameter. At the beginning of each interval, every machine broadcasts the current time according to its clock. Because the clocks on different machines do not run at exactly the same speed, these broadcasts will not happen precisely simultaneously.

After a machine broadcasts its time, it starts a local timer to collect all other broadcasts that arrive during some interval S. When all the broadcasts arrive, an algorithm is run to compute a new time from them. The simplest algorithm is just to average the values from all the other machines. A slight variation on this theme is first to discard the m highest and m lowest values, and average the rest. Discarding the extreme values can be regarded as self defense against up to m faulty clocks sending out nonsense.

Additional clock synchronization algorithms are discussed in the literature One of the most widely used algorithms in the Internet is the **Network Time Protocol (NTP)**. NTP is known to achieve (worldwide) accuracy in the range of 1-50 msec. It achieves this accuracy through the use of advanced clock synchronization algorithms.

Use of Synchronized Clocks

In the past few years, the necessary hardware and software for synchronizing clocks on a wide scale (e.g., over the entire Internet) has become easily available. With this new technology, it is possible to keep millions of clocks synchronized to within a few milliseconds of UTC. New algorithms that utilize synchronized clocks are just starting to appear. One example concerns how to

enforce at-most-once message delivery to a server, even in the face of crashes. The traditional approach is for each message to bear a unique message number, and have each server store all the numbers of the messages it has seen so that it can detect new messages from retransmissions. The problem with this algorithm is that if a server crashes and reboots, it loses its table of message numbers. Also, for how long should message numbers be saved?

Using time, the algorithm can be modified as follows. Now, every message carries a connection identifier (chosen by the sender) and a timestamp. For each connection, the server records in a table the most recent timestamp it has seen. If any incoming message for a connection is lower than the timestamp stored for that connection, the message is rejected as a duplicate.

To make it possible to remove old timestamps, each server continuously maintains a global variable

G = CurrentTime - MaxLifetime - MaxClockSkew

where MaxLifetime is the maximum time a message can live and MaxClockSkew is how far from UTC the clock might be at worst. Any timestamp older than G can safely be removed from the table because all messages that old have already died out. If an incoming message has an unknown connection identifier, it is accepted if its timestamp is more recent than G and rejected if its timestamp is older than G because anything that old surely is a duplicate. In effect, G is a summary of the message numbers of all old messages. Every ΔT, the current time is written to disk.

When a server crashes and then reboots, it reloads G from the time stored on disk and increments it by the update period, ΔT. Any incoming message with a timestamp older than G is rejected as a duplicate. As a consequence, every message that might have been accepted before the crash is rejected. Some new messages may be incorrectly rejected, but under all conditions the algorithm maintains at-most-once semantics.

Video

Video

LOGICAL CLOCKS

Q.3 Explain Logical Clocks.

(A) For many purposes, it is sufficient that all machines agree on the same time. It is not essential that this time also agrees with the real time as announced on the radio every hour. For running make, for example, it is adequate that all machines agree that it is 10:00, even if it is really 10:02. Thus for a certain class of algorithms, it is the internal consistency of the blocks that matters, it is conventional to speak of the clock to the real time. For these algorithms, it is conventional to speak of the clocks as **logical clocks.**

Lamport timestamps

To synchronize logical clocks, Lamport defined a relation called happened before. The expression $a \rightarrow b$ is read "a happens before b" and means that all processes agree that first event a occurs, then afterwards event b occurs. The happened before relation can be observed directly in two situations:

i) If a and b are events in the same process, and a occurs before b, then $a \rightarrow b$ is true.

ii) If a is the event of a message being sent by one process, and b is the event of the message being received by another process, then $a \rightarrow b$ is also true. A message cannot be received before it is sent, or even at the same time it is sent, since it takes a finite, nonzero amount of time to arrive.

Happened before is a transitive relation, so if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$. If two events, x and y, happen in different processes that do not exchange messages (not even indirectly via third parties), then $x \rightarrow y$ is not true, but neither is $y \rightarrow x$. These events are said to be concurrent, which simply means that nothing can be said (or need be said) about when the events happened or which event happened first.

What we need is a way of measuring time such that for every event, a, we can assign it a time value C(a) on which all processes agree. These time values must have the property that if $a \rightarrow b$, then $C(a) < C(b)$. To rephrase the conditions we stated earlier, if a and b are two events within the same process and a occurs before b, then $C(a) < C(b)$. Similarly, if a is the sending of a message by one process and b is the reception of that message by another process, then C(a) and C(b) must be assigned in such a way that everyone agrees on the values of C(a) and C(b) with $C(a) < C(b)$. In addition, the clock time, C, must always go forward (increasing), never backward (decreasing). Corrections to time can be made by adding a positive value, never by subtracting one.

Now let us look at the algorithm Lamport proposed for assigning times to events. Consider the three processes depicted in Fig. below. The processes run on different machines, each with its own clock, running at its own speed. As can be seen from the figure, when the clock has ticked 6 times in process 0, it has ticked 8 times in process 1 and 10 times in process 2. Each clock runs at a constant rate, but the rates are different due to differences in the crystals.

At time 6, process 0 sends message A to process 1. How long this message takes to arrive depends on whose clock you believe. In any event, the clock in process 1 reads 16 when it arrives. If the message carries the starting time, 6, in it, process 1 will conclude that it took 10 ticks to make the journey. This value is certainly possible. According to this reasoning, message B from 1 to 2 takes 16 ticks, again a plausible value.

Now comes the fun part. Message C from 2 to 1 leaves at 60 and arrives at 56. Similarly, message D from 1 to 0 leaves at 64 and arrives at 54. These values are clearly impossible. It is this situation that must be prevented.

Lamport's solution follows directly from the happened before relation. Since C left at 60, it must arrive at 61 or later. Therefore, each message carries the sending time according to the sender's clock. When a message arrives and the receiver's clock shows a value prior to the time the message was sent, the receiver fast forwards its clock to be one more than the sending time. In figure below we see that C now arrives at 61. Similarly, D arrives at 7

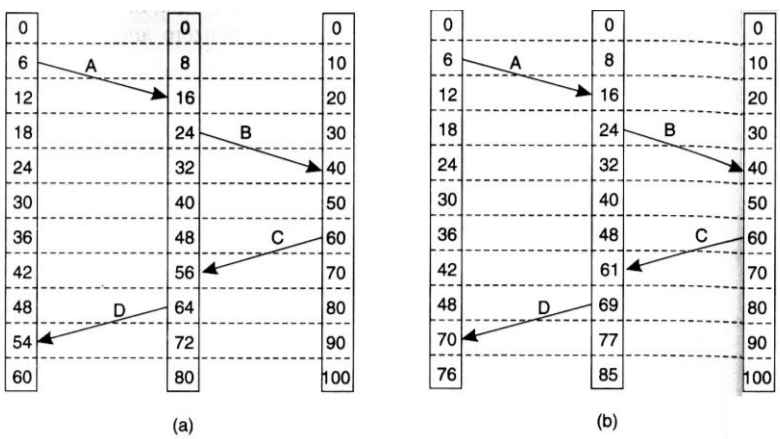


Fig. 1 : (a) Three process, each with its own clock. The clock run at different rates.
(b) Lamport's algorithm corrects the clocks

With one small addition, this algorithm meets our requirements for global time. The addition is that between every two events, the clock must tick at least once.

If a process sends or receives two messages in quick succession, it must advance its clock by (at least) one tick in between them.

Using this method, we now have a way to assign time to all events in a distributed system subject to the following conditions:

i) If a happens before b in the same process, C(a) < C(b).
ii) If a and b represent the sending and receiving of a message, respectively, C(a) < C(b).
iii) For all distinctive events a and b, C(a) and C(b) are different.

This algorithm gives us a way to provide a total ordering of all events in the system. Many other distributed algorithms need such an ordering to avoid ambiguities, so the algorithm is widely cited in the literature.

Example: Totally-Ordered Multicasting

As an application of Lamport timestamps, consider the situation in which a database has been replicated across several sites. For example, to improve query performance, a bank may place copies of an account database in two different cities, say New York and San Francisco. A query is always forwarded to the nearest copy. The price for a fast response to a query is partly paid in higher update costs, because each update operation must be carried out at each replica.

In fact, there is a more stringent requirement with respect to updates. Assume a customer in San Francisco wants to add $100 to his account, which currently contains $1,000. At the same time, a bank employee in New York initiates an update by which the customer's account is to be increased with 1 percent interest. Both updates should be carried out at both copies of the database. However, due to communication delays in the underlying network, the updates may arrive in the order as shown in Figure 2.

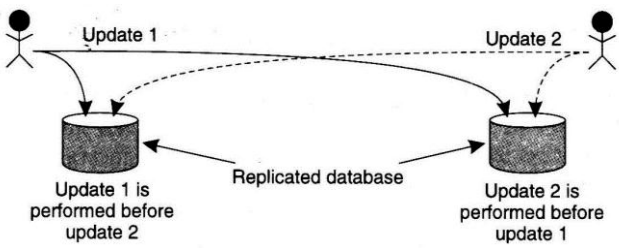


Fig. 2 : Updating a replicated database and leaving it in an inconsistent state.

The customer's update operation is performed in San Francisco before the interest update. In contrast, the copy of the account in the New York replica is first updated with the 1 percent interest, and after that with the $100 deposit. Consequently, the San Francisco database will record a total amount of $1,111, whereas the New York database records $1,110.

The problem that we are faced with is that the two update operations should have been performed in the same order at each copy. Although it makes a difference whether the deposit is processed before the interest update or the other

way around, which order is followed is not important from a consistency point of view. The important issue is that both copies should be exactly the same. In general, situations such as these require a totally-ordered multicast, that is, a multicast operation by which all messages are delivered in the same order to each receiver. Lamport timestamps can be used to implement **totally-ordered multicasts** in a completely distributed fashion.

Video

Vector Timestamps

Lamport timestamps lead to a situation where all events in a distributed system are totally ordered with the property that if event a happened before event b, then a will also be positioned in that ordering before b, that is, C(a) < C(b).

However, with Lamport timestamps, nothing can be said about the relationship between two events a and b by merely comparing their time values C(a) and C(b), respectively. In other words, if C(a) < C(b), then this does not necessarily imply that a indeed happened before b.

Consider a messaging system in which processes post articles and react to posted articles. One of the most popular examples of such a messaging system is the Internet's electronic bulletin board service, **network news**. Users, and hence processes, join specific discussion groups. Postings within such a group, whether they are articles or reactions, are multicast to all group members. To ensure that reactions are delivered after their associated postings, we may decide to use a totally-ordered multicasting scheme as described above. However, such a scheme does not imply that if message B is delivered after message A, that B is a reaction to what is posted by means of message A. In fact, the two may be completely independent. Totally-ordered multicasting is too strong in this case.

The problem is that Lamport timestamps do not capture **causality**. In our example, the receipt of an article always causally precedes the posting of a reac-

tion. Consequently, if causal relationships are to be maintained within a group of processes, then the receipt of the reaction to an article should always follow the receipt of that article. No more, no less. If two articles or reactions are independent, their order of delivery should not matter at all.

Vector Clock :

The main problem with Lamport time stamps is that, it do not capture causality. Causality can be captured by means of vector timestamp. A vector timestamp VT (a) assigned to an event a has the property that if VT (a) < VT (b) for some event b, then event a is known to causality precede event b. vector timestamps are constructed by letting each process Pi maintain a vector Vi with following two properties :

(i) Vi [i] is the number of events that have occurred so for at Pi.
(ii) If Vi [j] = k then Pi knows that k events occurred so for at Pj.

First property is maintained by incrementing Vi [i] at the occurance of each new event that happens at process Pi. The second property is maintained by piggybacking vector along with messages that are sent.

When Pi send the message m, it sends along with its current vector as a timestamp Vt. In this way, receiver is informed about number of events that have occurred at Pi. In other words timestamp Vt of m tells the receiver how many events in other processes have preceded m and on which m may causally depend. When process Pi receives m, it adjusts its own vector by setting each entry Vi [k] to max {Vi [k], Vt [k]}. The vector now reflects the number of messages that Pi must receive to have at least seen the same messages that preceded the sending of m. Hereafter, entry VT [i] is incremented by, representing the event of receiving a next message.

GLOBAL STATE

On many occasions, it is useful to know the global state in which a distributed system is currently residing. The global state of a distributed system consists of the local state of each process, together with the messages that are currently in transit, that is, that have been sent but not delivered. What exactly the local state of a process is depends on what we are interested in. In the case of a distributed database system, it may consist of only those records that form part of the database and exclude temporary records used for computations. In our example of tracing-based garbage collection as discussed in the previous chapter, the local state may consist of variables representing markings for those proxies, skeletons, and objects that are contained in the address space of a process.

Knowing the global state of a distributed system may be useful for many reasons. For example, when it is known that local computations have stopped and that there are no more messages in transit, the system has obviously entered a state in which no more progress can be made. By analyzing such a global state, it may be concluded that we are either dealing with a deadlock or that a distributed computation has correctly terminated.

A simple, straightforward way for recording the global state of a distributed system was proposed by Chandy and Lamport who introduced the notion of a distributed snapshot. A **distributed snapshot** reflects a state in which the distributed system might have been. An important property is that such a snapshot reflects a consistent global state. In particular, this means that if we have recorded that a process P has received a message

from another process Q, then we should also have recorded that process Q had actually sent that message. Otherwise, a snapshot will contain the recording of messages that have been received but never sent, which is obviously not what we want. The reverse condition (Q has sent a message that P has not yet received) is allowed, however.
The notion of a global state can be graphically represented by what is called a cut, as shown in Fig. 1. A consistent cut is shown by means of the dashed line crossing the time axis of the three processes P1, P2, and P3 figure 1(a). The cut represents the last event that has been recorded for each process. In this case, it can be readily verified that all recorded message receipts have a corresponding recorded send event. In contrast, Figure 1(b) below shows an inconsistent cut. The receipt of message m2 by process P3 has been recorded, but the snapshot contains no corresponding send event.

To simplify the explanation of the algorithm for taking a distributed snapshot, we assume that the distributed system can be represented as a collection of processes connected to each other through unidirectional point-to-point communication channels. For example, processes may first set up TCP connections before any further communication takes place.

Any process may initiate the algorithm. The initiating process, say P, starts by recording its own local state. Then, it sends a marker along each of its outgoing channels, indicating that the receiver should participate in recording the global state.

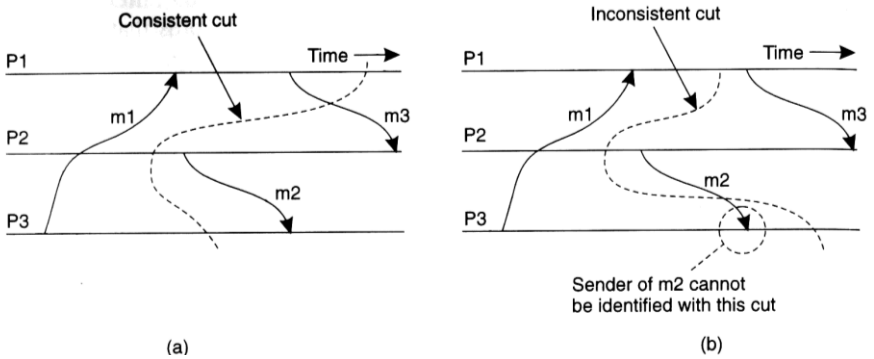


Fig. 1 : (a) A consistent cut. (b) An inconsistent cut.

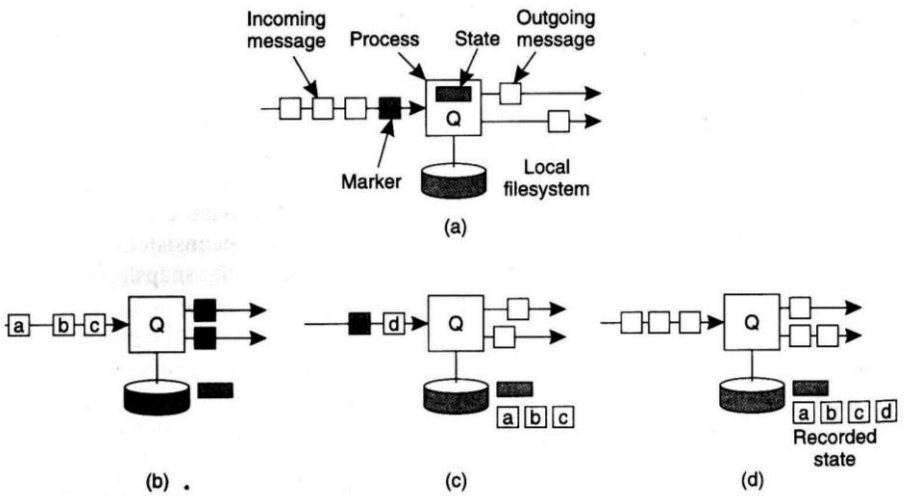


Fig. 2 : (a) Organization of a process and channels for a distributed snapshot.
(b) Process Q receives a marker for the first time and records its local state.
(c) Q records all incoming messages.
(d) Q receives a marker for its incoming channel and finishes recording the state of the incoming channel.

When a process Q receives a marker through an incoming channel C, its action depends on whether or not it has already saved its local state. If it has not already done so, it first records its local state and also sends a marker along each of its own outgoing channels. If Q had already recorded its state, the marker on channel C is an indicator that Q should record the state of the channel. This state is formed by the sequence of messages that have been received by Q since the last time Q recorded its own local state, and before it received the marker. Recording this state is shown in Figure 2.

A process is said to have finished its part of the algorithm when it has received a marker along each of its incoming channels, and processed each one. At that point, its recorded local state, as well as the state it recorded for each incoming channel, can be collected and sent, for example, to the process that initiated the snapshot. The latter can then subsequently analyze the current state. Note that, meanwhile, the distributed system as a whole can continue to run normally.

Example : Termination Detection

As an application of taking a snapshot, consider detecting the termination of a distributed computation. If a process Q receives the marker requesting a snapshot for the first time, it considers the process that sent that marker as its predecessor. When Q completes its part of the snapshot, it sends its predecessor a DONE message. By recursion, when the initiator of the distributed snapshot has received a DONE message from all its successors, it knows that the snapshot has been completely taken.

What is needed is a snapshot in which all channels are empty. The following is a simple modification to the algorithm described above. When a process Q finishes its part of the snapshot, it either returns a DONE message to its predecessor, or a CONTINUE message. A DONE message is returned only when the following two conditions are met:

1. All of Q's successors have returned a DONE message.
2. Q has not received any message between the point it recorded its state, and the point it had received the marker along each of its incoming channels.

In all other cases Q sends a CONTINUE message to its predecessor.

Video

Election Algorithms

Q.4 Explain Election Algorithms.

(A) If all processes are exactly the same, with no distinguishing characteristics, there is no way to select one of them to be special. Consequently, we will assume that each process has a unique number, for example, its network address (for simplicity, we will assume one process per machine). In general, election algorithms attempt to locate the process with the highest process number and designate it as coordinator. The algorithms differ in the way they do the location.

Furthermore, we also assume that every process knows the process number of every other process. What the processes do not know is which ones are currently up and which ones are currently down. The goal of an election algorithm is to ensure that when an election starts, it concludes with all processes agreeing on who the new coordinator is to be.

The Bully Algorithm

As a first example, consider the bully algorithm devised by Garcia-Molina . When any process notices that the coordinator is no longer responding to requests, it initiates an election. A process, P, holds an election as follows:

i) P sends an ELECTION message to all processes with higher numbers.
ii) If no one responds, P wins the election and becomes coordinator.
iii) If one of the higher-ups answers, it takes over. P's job is done.

At any moment, a process can get an ELECTION message from one of its lower-numbered colleagues. When such a message arrives, the receiver sends an OK message back to the sender to indicate that he is alive and will take over. The receiver then holds an election, unless it is already holding one. Eventually, all processes give up but one, and that one is the new coordinator. It announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator.

If a process that was previously down comes back up, it holds an election. If it happens to be the highest-numbered process currently running, it will win the election and take over the coordinator's job. Thus the biggest guy in town always wins, hence the name "bully algorithm."

In Figure below we see an example of how the bully algorithm works. The group consists of eight processes, numbered from 0 to 7. Previously process 7 was the coordinator, but it has just crashed. Process 4 is the first one to notice this, so it sends ELECTION messages to all the processes higher than it, namely 5, 6, and 7, as shown in Figure 1(a) below. Processes 5 and 6 both respond with OK, as shown in Figure 1(b) below. Upon getting the first of these responses, 4 knows that its job is over. It knows that one of these bigwigs will take over and become coordinator. It just sits back and waits to see who the winner will be (although at this point it can make a pretty good guess).

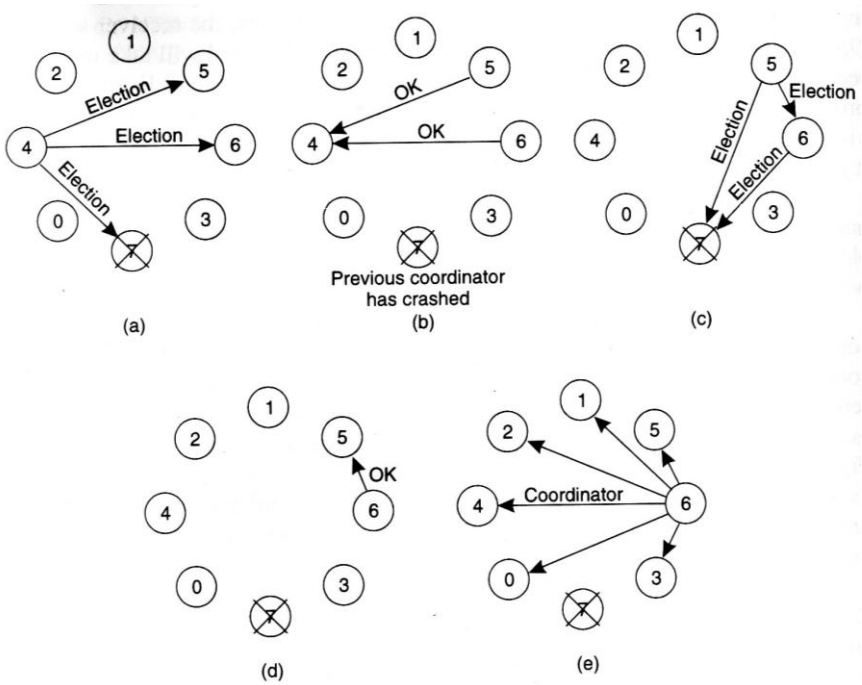


Fig. 1 : The bully election algorithm.
(a) Process 4 holds an election, (b) Processes 5 and 6 respond, telling 4 to stop.
(c) Now 5 and 6 each hold an election. (d) Process 6 wins and tells everyone.

In Figure 1(c) below, both 5 and 6 hold elections, each one only sending messages to those processes higher than itself. In Figure 1(d) below process 6 tells 5 that it will take over. At this point 6 knows that 7 is dead and that it (6) is the winner. If there is state information to be collected from disk or elsewhere to pick up where the old coordinator left off, 6 must now do what is needed. When it is ready to take over, 6 announces this by sending a COORDINATOR message to all running processes. When 4 gets this message, it can now continue with the operation it was trying to do when it discovered that 7 was dead, but using 6 as the coordinator this time. In this way the failure of 7 is handled and the work can continue.

If process 7 is ever restarted, it will just send all the others a COORDINATOR message and bully them into submission.

A Ring Algorithm

Another election algorithm is based on the use of a ring. Unlike some ring algorithms, this one does not use a token. We assume that the processes are physically or logically ordered, so that each process knows who its successor is. When any process notices that the coordinator is not functioning, it builds an ELECTION message containing its own process number and sends the message to its successor. If the successor is down, the sender skips over the successor and goes to the next member along the ring, or the one after that, until a running process is located. At each step, the sender adds its own process number to the list in the message effectively making itself a candidate to be elected as coordinator.

Eventually, the message gets back to the process that started it all. That process recognizes this event when it receives an incoming message containing its own process number. At that point, the message type is changed to COORDINATOR and circulated once again, this time to inform everyone else who the coordinator is (the list member with the highest number) and who the members of the new ring are. When this message has circulated once, it is removed and everyone goes back to work.

In Figure 2 below we see what happens if two processes, 2 and 5, discover simultaneously that the previous coordinator, process 7, has crashed. Each of these builds an ELECTION message and each of them starts circulating its message, independent of the other one. Eventually, both messages will go all the way around, and both 2 and 5 will convert them into COORDINATOR messages, with exactly the same members and in the same order. When both have gone around again, both will be removed. It does no harm to have extra messages circulating; at worst it consumes a little bandwidth, but this not considered wasteful.

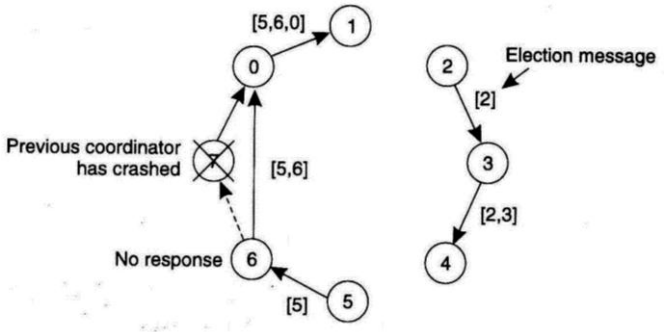


Fig. 2 : Election algorithm using a ring

Video

MUTUAL EXCLUSION

Q.5 What is Mutual Exclusion.

(A) Systems involving multiple processes are often most easily programmed using critical regions. When a process has to read or update certain shared data structures, it first enters a critical region to achieve mutual exclusion and ensures that no other process will use the shared data structures at the same time. In single-processor systems, critical regions are protected using semaphores, monitors, and similar constructs. We will now look at a few examples of how critical regions and mutual exclusion can be implemented in distributed systems.

A Centralized Algorithm

The most straightforward way to achieve mutual exclusion in a distributed system is to simulate how it is done in a one-processor system. One process is elected as the coordinator (e.g., the one running on the machine with the highest network address). Whenever a process wants to enter a critical region, it sends a request message to the coordinator stating which critical region it wants to enter and asking for permission. If no other process is currently in that critical region, the coordinator sends back a reply granting permission, as shown in Figure (a) below. When the reply arrives, the requesting process enters the critical region.

Now suppose that another process, 2 in Figure (b) below, asks for permission to enter the same critical region. The coordinator knows that a different process is already in the critical region, so it cannot grant permission. The exact method used to deny permission is system dependent. In Figure (b) below, the coordinator just refrains from replying, thus blocking process 2, which is waiting for a reply. Alternatively, it could send a reply saying "permission denied." Either way, it queues the request from 2 for the time being and waits for more messages.

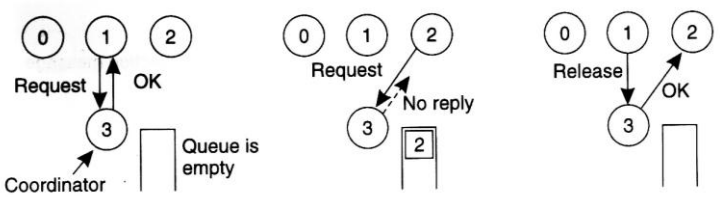


Fig. : (a) Permission 1 asks the coordinator for permission to enter a critical region. Permission is granted.
(b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply.
(c) When process 1 exits the critical region, it tells the coordinator, which then replies to 2.

When process 1 exits the critical region, it sends a message to the coordinator releasing its exclusive access, as shown in Figure (c) below. The coordinator takes the first item off the queue of deferred requests and sends that process a grant message. If the process was still blocked (i.e., this is the first message to it), it unblocks and enters the critical region. If an explicit message has already

been sent denying permission, the process will have to poll for incoming traffic or block later. Either way, when it sees the grant, it can enter the critical region.
The centralized approach also has shortcomings. The coordinator is a single point of failure, so if it crashes, the entire system may go down. If processes normally block after making a request, they cannot distinguish a dead coordinator from "permission denied" since in both cases no message comes back. In addition, in a large system, a single coordinator can become a performance bottleneck.

Q.6 What is a Distributed Algorithm?

(A) Having a single point of failure is frequently unacceptable, so researchers have looked for distributed mutual exclusion algorithms. Lamport's 1978 paper on clock synchronization presented the first one.

The algorithm works as follows. When a process wants to enter a critical region, it builds a message containing the name of the critical region it wants to enter, its process number, and the current time. It then sends the message to all other processes, conceptually including itself. The sending of messages is assumed to be reliable; that is, every message is acknowledged. Reliable group communication if available, can be used instead of individual messages.

When a process receives a request message from another process, the action it takes depends on its state with respect to the critical region named in the message. Three cases have to be distinguished:

i) If the receiver is not in the critical region and does not want to enter it, it sends back an OK message to the sender.
ii) If the receiver is already in the critical region, it does not reply. Instead, it queues the request.
iii) If the receiver wants to enter the critical region but has not yet done so, it compares the timestamp in the incoming message with the one contained in the message that it has sent everyone. The lowest one wins. If the incoming message is lower, the receiver sends back an OK message. If its own message has a lower timestamp, the receiver queues the incoming request and sends nothing.

After sending out requests asking permission to enter a critical region, a process sits back and waits until everyone else has given permission. As soon 'as all the permissions are in, it may enter the critical region. When it exits the critical region, it sends OK messages to all processes on its queue and deletes them all from the queue.

Let us try to understand why the algorithm works. If there is no conflict, it clearly works. However, suppose that two processes try to enter the same critical region simultaneously, as shown in Figure 1(a) below.

Process 0 sends everyone a request with timestamp 8, while at the same time, process 2 sends everyone a request with timestamp 12. Process 1 is not interested in entering the critical region, so it sends OK to both senders. Processes 0 and 2 both see the conflict and compare timestamps. Process 2 sees that it has lost, so it grants permission to 0 by sending OK. Process 0 now queues the request from 2 for later processing and enters the critical region, as shown in Figure 1(b) below. When it is finished, it removes the request from 2 from its queue and sends an OK message to process 2, allowing the latter to enter its critical region, as shown in Figure 1(c) below. The algorithm works

because in the case of a conflict, the lowest timestamp wins and everyone agrees on the ordering of the timestamps.

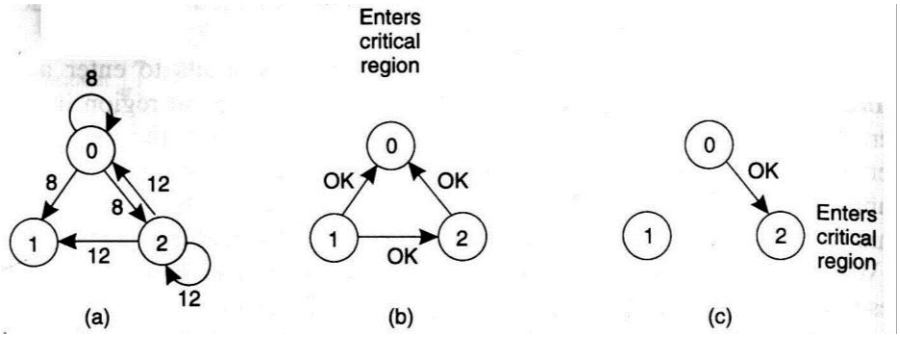


Fig. 1 : (a) Two processes want to enter the same critical region at the same moment.
(b) Process 0 has the lowest timestamp, so it wins.
(c) When process 0 is done, it sends an OK also, so 2 can now enter the critical region.

A Token Ring Algorithm

A completely different approach to achieving mutual exclusion in a distributed system is illustrated in Fig. below. Here we have a bus network, as shown in Figure 2(a) below, (e.g., Ethernet), with no inherent ordering of the processes. In software, a logical ring is constructed in which each process is assigned a position in the ring, as shown in Figure 2(b) below. The ring positions may be allocated in numerical order of network addresses or some other means. It does not matter what the ordering is. All that matters is that each process knows who is next in line after itself.

When the ring is initialized, process 0 is given a token. The token circulates around the ring. It is passed from process k to process k + 1 (modulo the ring size) in point-to-point messages. When a process acquires the token from its neighbor, it checks to see if it is attempting to enter a critical region. If so, the process enters the region, does all the work it needs to, and leaves the region. After it has exited, it passes the token along the ring. It is not permitted to enter a second critical region using the same token.

If a process is handed the token by its neighbor and is not interested in entering a critical region, it just passes it along. As a consequence, when no processes want to enter any critical regions, the token just circulates at high speed around the ring.

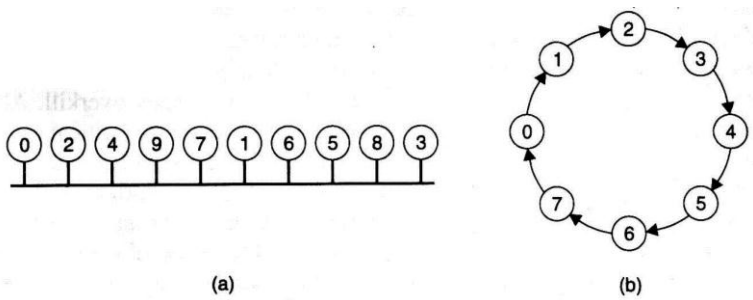


Fig. 2 : (a) An unordered group of processes on a network.
(b) A logical ring constructed in software.

The correctness of this algorithm is easy to see. Only one process has the token at any instant, so only one process can actually be in a critical region. Since the token circulates among the processes in a well-defined order, starvation cannot occur. Once a process decides it wants to enter a critical region, at worst it will have to wait for every other process to enter and leave one critical region.

A Comparison of the Three Algorithms

A brief comparison of the three mutual exclusion algorithms we have looked at is instructive. In Fig. below we have listed the algorithms and three key properties: the number of messages required for a process to enter and exit a critical region, the delay before entry can occur (assuming messages are passed sequentially over a network), and some problems associated with each algorithm.

The centralized algorithm is simplest and also most efficient. It requires only three messages to enter and leave a critical region: a request, a grant to enter, and a release to exit. The distributed algorithm requires n - 1 request messages, one to each of the other processes, and an additional n - 1 grant messages, for a total of 2(n - 1). (We assume that only point-to-point communication channels are used.) With the token ring algorithm, the number is variable. If every process constantly wants to enter a critical region, then each token pass will result in one entry and exit, for an average of one message per critical region entered. At the other extreme, the .token may sometimes circulate for hours without anyone being interested in it. In this case, the number of messages per entry into a critical region is unbounded.

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Distributed	2 (n-1)	2 (n-1)	Crash of any process
Token ring	1 to 0	0 to n - 1	Lost token, process crash

A comparison of three mutual exclusion algorithms.

Finally, all three algorithms suffer badly in the event of crashes. Special measures and additional complexity must be introduced to avoid having a crash bring down the entire system. It is ironic that the distributed algorithms are even more sensitive to crashes than the centralized one. In a fault-tolerant system, none of these would be suitable, but if crashes are very infrequent, they might do.

7.2 TOKEN BASED ALGORITHMS: SUZUKI-KASAMI'S BROADCAST ALGORITHMS

SINGHAL'S HEURASTIC ALGORITHM,

RAYMOND'S TREE BASED ALGORITHM,

References

1. https://www.google.co.in/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=0ahUKEwjgnIXS2Y_RAhULtl8KHdkMD9cQjRwIBw&url=http%3A%2F%2Fsynch.dj%2F&psig=AFQjCNGrTbMb6PNac0ysly5Tz3sRov8g2Q&ust=1482767502851734
2. http://synch.dj/
3. https://www.youtube.com/watch?v=GAZAT068Hbg
4. https://www.youtube.com/watch?v=bnrD2n55dfk
5. https://www.youtube.com/watch?v=7_9CR9aRKBk
6. https://www.youtube.com/watch?v=XDdqF8FfRx8
7. https://www.youtube.com/watch?v=xaISZOQ-PWY
8. https://www.youtube.com/watch?v=aWne_qIR2XI
9. https://www.youtube.com/watch?v=jFULOEHPqgo
10. https://www.youtube.com/watch?v=KC3J52L0jFE

11. QUIZ

1. In distributed systems, a logical clock is associated with
a) each instruction
b) each process
c) each register
d) none of the mentioned
View Answer
Answer:b

2. If timestamps of two events are same, then the events are
a) concurrent
b) non-concurrent
c) monotonic
d) non-monotonic
View Answer
Answer:a

3. If a process is executing in its critical section
a) any other process can also execute in its critical section
b) no other process can execute in its critical section
c) one more process can execute in its critical section
d) none of the mentioned
View Answer
Answer:b

4. A process can enter into its critical section
a) anytime
b) when it recieves a reply message from its parent process
c) when it recieves a reply message from all other processes in the system
d) none of the mentioned
View Answer
Answer:c

5. For proper synchronization in distributed systems
a) prevention from the deadlock is must
b) prevention from the starvation is must
c) both (a) and (b)
d) none of the mentioned
View Answer
Answer:c

6. In the token passing approach of distributed systems, processes are organized in a ring structure
a) logically
b) physically
c) both (a) and (b)
d) none of the mentioned
View Answer
Answer:a

7. In distributed systems, transaction coordinator
a) starts the execution of transaction
b) breaks the transaction into number of sub transactions
c) coordinates the termination of the transaction
d) all of the mentioned
View Answer
Answer:d

8. In case of failure, a new transaction coordinator can be elected by
a) bully algorithm
b) ring algorithm
c) both (a) and (b)
d) none of the mentioned
View Answer
Answer:c

9. In distributed systems, election algorithms assumes that
a) a unique priority number is associated with each active process in system
b) there is no priority number associated with any process
c) priority of the processes is not required
d) none of the mentioned
View Answer
Answer:a

10. According to the ring algorithm, links between processes are
a) bidirectional
b) unidirectional
c) both (a) and (b)
d) none of the mentioned
View Answer
Answer:b

GQ

1. What is synchronization? Explain.

2. Define Synchronization. Explain clock synchronization.

3. What are physical clocks?

4. Explain :
(i) Cristian's Algorithm
(ii) Benkeley algorithm
(iii) Logical clocks
(iv) Lamport timestamps

5. List all clock synchronization algorithms. Explain any one in detail.

6. Explain the different distributed physical clock synchronization algorithms with their relative advantages and disadvantages.

7. What are synchronized clocks?

8. Explain the distributed algorithms for clock synchronization.

9. What is NTP?

10. Explain the idea of timestamps. What is Lamport timestamp? Explain in detail.

11. Define Multicasting. Explain totally-ordered multicasting.

12. Explain clocks. Explain its types.

13. Explain global state. What is its importance?

14. Discuss Election algorithms. Why are they called so?

15. Write detailed notes on :
(i) Bully Algorithm
(ii) Ring Algorithm

16. Explain mutual exclusion.

17. How to achieve mutual exclusion?

18. Compare and contrast Mutual Exclusion Algorithms.

19. Compare and contrast centralized and distributed algorithm.

20. Explain token ring algorithm.

21. Compare election algorithm.

22. Explain the different distributed physical clock synchronization algorithms with their relative advantages and disadvantages.

23. Describe the different approaches for deadlock detection in DS.

24. Write short note on Ricart Agrawala Algorithm-merits and Demerits.

25. Explain distributed algorithm for mutual exclusion. What are the advantages and disadvantages of it over centralized algorithms?

26. Explain the process of synchronization physical and logical clocks.