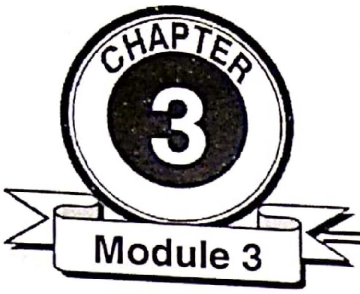


Synchronization

Syllabus

Clock Synchronization, Logical Clocks, Election Algorithms, Mutual Exclusion, and Distributed Mutual Exclusion-Classification of mutual Exclusion Algorithm, Requirements of Mutual Exclusion Algorithms, and Performance measure. Non Token based Algorithms : Lamport Algorithm, Ricart–Agrawala's Algorithm, Maekawa's Algorithm, Token Based Algorithms : Suzuki-Kasami's Broardcast Algorithms, Singhal's Heurastic Algorithm, Raymond's Tree based Algorithm, Comparative Performance Analysis.

Introduction

- Apart from the communication between processes in distributed system, cooperation and synchronization between processes is also important. Cooperation is supported through naming which permits the processes to share resources. As an example of synchronization, multiple processes in distributed system should agree on some ordering of events occurred.
- Processes should also cooperate with each other to access the shared resources so that simultaneous access of these resources will be avoided. As processes run on different machines in distributed system, synchronization is no easy to implement with compare to uniprocessor or multiprocessor system.

3.1 Clock Synchronization

- In centralized system, if process P asks for the time to kernel, it will get it. After sometime if process Q asks for time, naturally it will get time having value greater than or equal to the value of time of process P received. In distributed system, agreement on time is important and necessary to achieve it.
- Practically, different machines clock in distributed system differs in their time value. Therefore, clock synchronization is required in distributed system. Consider example of UNIX make program. Editor is running on machine A and Compiler is running on machine B. Large UNIX program is collection of many source files. If few files suppose modified then no need to recompile all the files in program. Only modified files require recompilation.
- After changing source files, programmer starts make program which checks timestamp of creation of source and object file. For example xyz.c has timestamp 3126 and xyz.o has timestamp 3125. In this case, since object file xyz.o has greater timestamp, make program confirms that xyz.c has been changed and recompilation is required. If abc.c has timestamp 3128 and abc.o has 3129 then no compilation is required.
- Suppose there is no global agreement on time. In this case suppose abc.o has timestamp 3129 and abc.c is modified and assigned time 3128.

- This is due to slightly lagging of machine clock on which abc.c is modified. Here make program will not call compiler and object file abc.o will be old version although its source file abc.c is changed. The resulting executable binary of large size UNIX program will contain mix of object files from old and new sources.

3.1.1 Physical Clocks

- Every machine has timer which is machined quartz crystal. This quartz crystal oscillates when kept under tension. The frequency of oscillation depends on type of crystal, how it is cut and how much tension is applied. Two registers, **counter** and **holding register** is associated with crystal. Counter value gets decremented by one after completion of one oscillation of the crystal. When counter value gets to 0 then interrupt is generated. Again counter gets reloaded from holding register.
- Each interrupt is called **clock tick** and it is possible to program a timer to generate 60 clock ticks per second. Battery backed-up CMOS RAM keeps time and date. After booting, with each clock tick interrupt service procedure adds 1 to current time. Practically it is not true that crystal in all the machines runs with same frequency. The difference between two clocks time value is called **clock skew.**
- In real time system, actual clock time is important. So consideration of multiple physical clocks is necessary. How to synchronize these clocks with real world clocks and how to synchronize clocks with each other are two important issues that need to be considered.
- Mechanical clock was invented in 17^{th} century and since then time has been measured astronomically. When sun reaches at its highest apparent point in sky is called as **transit of sun.** Solar day is interval between two consecutive transits of sun. Each day have 24 hour. 1 hour contain 60 minutes and in each minute contain 60 seconds. Hence, each day have (24x60x60) total 84600 seconds. **Solar second** is $1/84600^{th}$ solar day.
- In 1940 it was invented that earth is slowing down and hence period of earth's rotation is also varying. As a results, astronomers considered large number of days and taken the average of it before dividing by 84600. This is (1/84600) is called **mean solar second.**
- In 1948, physicist started measuring time with cesium 133 clock. One mean solar second is equal to time that is needed to cesium 133 atom to make 9,192631,770 transitions. Now days, 50 laboratories worldwide kept cesium clock and reports periodically time to BIH in Paris which takes average of all the time called as **International atomic time (TAI).** Now 84600 TAI seconds is about 3 msec less than mean solar day. BIH solved this problem by using leap second when difference between TAI and solar time grows 800 msec. After correction of time, it is called **universally coordinated time (UTC).**
- National institute of standard time (NIST) run shortwave broadcast radio station that broadcast short pulse at start of each UTC second with accuracy of ± 10 msec. In practice, this accuracy is no better than ±10msec. Several earth satellites also provide UTC service.

3.1.2 Global Positioning System (GPS)

- GPS was launched in 1978. It is a satellite-based distributed system which has been used mainly for military applications. Now days it is used in many civilian applications such as traffic navigation and in GPS phones.

- GPS makes use of 29 satellites which travels in an orbit at a height of about 20,000 km. Every satellite has 4 atomic clocks which are calibrated regularly from special stations on earth. Each satellite constantly broadcast its current position. This message is with timestamp as per its local time.
- The receiver of this message on earth can accurately calculate its actual position by using three satellites. Three satellites are required to resolve the longitude, latitude, and altitude of a receiver on Earth. Practically it is not true that all clocks are absolutely synchronized. Satellite message also takes some time to reach to receiver.
- It may also happen that, receiver clock is lagging or leading with respect to clock time of satellite. GPS does not consider the leap second. Hence, measurement is not perfectly accurate.

3.1.3 Clock Synchronization Algorithms

- If one machine is with UTC time then all other machines clocks time should be synchronized with respect to this UTC time. Otherwise, all the machines clocks together should be with same time. Consider timer of each machine interrupts X times a second. Assume that value of this clock is Y. When UTC time is t then value of clock at machine p is $Y_p(t)$. If all the clocks value is UTC time t then $Y_p(t) = t$ for all p, which is required. It means, dY/dt should be 1.
- In real world, timer do not interrupt exactly X times a second. If X=0 then timer should generate 216,000 ticks per hour. Practically, machine gets this value in the range 215,998 to 216,002 ticks per hour. For any constant α, if $1-\alpha \leq dY/dt \leq 1+\alpha$ then timer is working within its specification. The constant α is specified by manufacturer of timer and called as **maximum drift rate.**
- If dY/dt is greater than 1 then clock is faster. If dY/dt is equal to 1 then clock is perfect clock. If dY/dt is less than 1 the clock is slower with respect to UTC time.

Cristian's Algorithm

- In this algorithm, it is assumed that one machine (time server) has WWV receiver. Hence, this machine clock value is UTC time. All other machines (client) synchronize their clock with this UTC time. Let C_{UTC} is the time replied by time server.

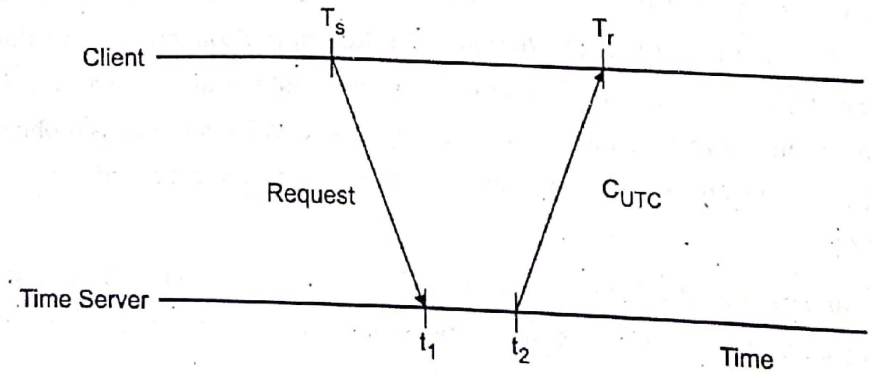


Fig. 3.1.1

- When client receives response from time server at time Tr, it sets its clock at time C_{UTC}. This algorithm has two problems, one is major and other is minor problem. Major problem occur when clients clock is fast. The received C_{UTC} will be less than client's current value C.

- In UNIX program example explained previously, if on client machine object file is compiled just after clock change will have timestamp less than the source which was modified just before clock change.
- The solution to this problem is to introduce the change slowly. If each interrupt of timer adds 5 msec to the time then add 4 msec to time until time gets corrected. Time can be advanced by adding 6 msec for each interrupt instead of forwarding all at once.
- As a minor problem, it takes some time to reach the reply from time server to client. This propagation time is (Tr – Ts)/2. Add this time to the C_{urc} time which is replied by server to set the client's clock time. Let time elapsed between t_1 and t_2 is interrupt processing time (I) at server. Then message propagation time is (Tr – Ts – I). One way propagation time will be the half of this time. In this algorithm, time server is passive as it replies as per query of client.

The Berkeley Algorithm

- In Berkeley UNIX, time daemon (server) is active as it polls every machine asking current time there. It then calculates average time of received answers from all machines. It then tells other machines either advance their clocks to new time or slow their clocks until specified time is set.
- This algorithm is used in system where WWV receiver is not there for UTC time. The daemon time is periodically set manually by operator.

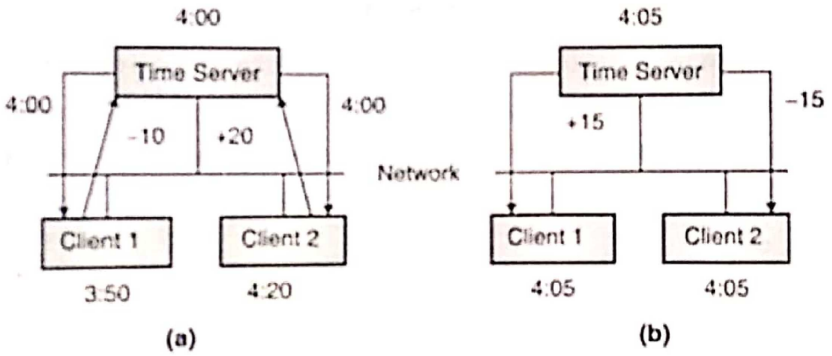


Fig. 3.1.2

- As shown in Fig. 3.1.2(a), initially time daemon has time value of 4:00. Client 1 has clock time value 3:50 and client 2 has clock time value of 4:20. Time server asks to other machines about their current time including it. Client 1 replies that, it is lagging by 10 with respect to server time which is 4:00. Similarly, client 2 replies that, it is leading by 20 with respect to server time which is 4:00. Server will reply as 0 to itself.
- Meanwhile server time increases to 4:05. Therefore server sends client 1 and 2 messages about how to adjust their clocks. It tells client 1 to advance clock time by 15. Similarly, it tells to client 2 to decrement the time value by 15. Server itself tells to add 5 time its own clock time. It is shown in Fig. 3.1.2(b).

Averaging Algorithms

- Cristian algorithm and Berkeley algorithm both are centralized algorithms. Centralized algorithms have certain disadvantages. Averaging algorithm is decentralized algorithm. One of these algorithms works by dividing time in fixed length intervals. Let T0 is any agreed upon moment in past. The k^{th} interval start at time T0 +kS and runs until T0 + (k+1)S. Here S is a system parameter.

- At the start of each interval, every machine broadcast its clock time. This broadcast will happen at different time at different clocks as machines clock speed can be different. After machine finish the broadcast, it starts local timer in order to collect broadcast from other machines in some time interval I. When all the broadcasts are collected at every machine, following algorithms are used.
  - Average the time values collected from all machines.
  - Other version is to discard n highest and n lowest time values and take the average of rest.
  - Another variation is to add propagation time of message from source machine in received time. Propagation time can be calculated using known topology of the network.

Clock Synchronization in Wireless Networks

- In wireless network, algorithm needs to be optimized considering energy consumption of nodes. It is not easy to deploy time server just like traditional distributed system. So, design of clock synchronization algorithms requires different insights.
- Reference broadcast synchronization (RBS) is a clock synchronization protocol. It focuses on internal synchronization of clocks just like the Berkeley algorithm. It does not consider synchronization with respect to UTC time. In this protocol, only receiver synchronizes while keeping sender out of loop.
- In sensor network, when message leaves the network interface of sender, it takes constant time to reach destination. Here multi-hop routing is not assumed. Propagation time of the message is measured from the point message leaves the network interface of the sender. RBS considers only delivery time at receivers. Sender broadcasts reference message say m. When receiver x receives this message, it records time of receiving of m which is say $T_x m$. This time is recorded with its local clock.
- Two nodes x and y exchanges their receiving time of m to calculate the offset. M is total number of sent reference messages.

$$\text{Offset } [x, y] = \frac{\sum_{k=1}^{M} (Tx, k - Ty, k)}{M}$$

- As clocks drift apart from each other, calculating average only will not work. Standard linear regression is used to calculate offset function.

Network Time Protocol (NTP)

- In cristian algorithm when time server send reply message with UTC time, the problem is to find actual propagation time. The propagation time of reply will definitely affect the reporting of actual UTC time.
- When reply will reach at client, message delay will have outdated reported time. The good estimation for this delay can be calculated as shown in following Fig. 3.1.3.

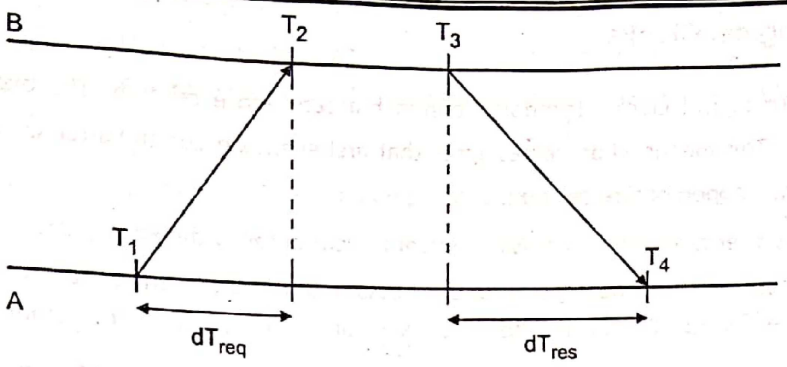


Fig. 3.1.3

- A sends message to B containing timestamp T_1. B also records its receiving time T_2 by its local clock. B then send reply containing timestamp T_3 along with its recorded timestamp T_2. Finally, A records time T_4 at which response arrives. Let $T_2 - T_1 = T_4 - T_3$. Estimation of its offset by A is given as below.

$$\theta = T_3 - \frac{(T_2 - T_1) + (T_4 - T_3)}{2} = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

- The value of the θ will be less than zero if A's clock is fast. In this case A has to set its clock to backward time. Practically it will produce problem like object file is compiled just after clock change will have timestamp less than the source which was modified just before clock change. The solution to this problem is to introduce the change slowly. If each interrupt of timer adds 5 msec to the time then add 4 msec to time until time gets corrected. Time can be advanced by adding 6 msec for each interrupt instead of forwarding all at once.
- In case of NTP, this protocol is set up pair-wise between servers. In other word, B will also query to A for its current time. Along with calculation of offset as above, estimation of μ for delay is carried out as

$$\mu = \frac{(T_2 - T_1) + (T_4 - T_3)}{2}$$

- Eight pairs of (θ, μ) values are buffered. Smallest value for μ is the best estimation for the delay between the two servers. Subsequently the associated value of θ is the most reliable estimation of offset. Some clocks are more accurate, say B's clock. Servers are divided in strata.
- If A contacts B, it will only correct its time provided its own stratum level is higher than that of B. After synchronization, A's stratum level will become one higher than that of B. Because of the symmetry of NTP, if A's stratum level was lower than that of B, B will adjust itself to A. Stratum-I server is with WWV receiver or with cesium clock. Many features of NTP are related to identification and masking of errors and security attacks.

3.2 Logical Clocks

- In many applications, all machines should agree on same time although this time does not agree with UTC time or real time. In this case, internal consistency of clocks is essential. Many algorithm works based on internal consistency of clocks and not with real time. For these algorithms, clocks are said to be logical clocks.
- Lamport showed that, if two processes are not interacting then lack of their clocks synchronization will not cause any problem. He also pointed out that, processes should agree on the order in which events occur and not on exactly what time it is.

3.2.1 Lamport's Logical Clocks

- For synchronization of logical clocks, Lamport defined **happen-before** relation. The expression x → y is read as "x happens before y". This means all processes agree that first event x occur and after x, event y occur. Following two situations indicates the happen before relation.
  - If x and y are the events in same process and event x occur before y then x → y is true.
  - If x is event of sending the message by one process and y is the event of receiving same message by other process then x → y is true. Practically, message takes nonzero time to deliver to other process.
- If x → y and y → z then x → z. It means happen before relation is transitive. If any two processes does not exchange messages directly or indirectly, then x → y and y → x where x and y are the events that occur in these processes. In this case, events are said to be **concurrent.**
- Let T(x) be the time value of event x. If x → y then T(x) < T(y). If x and y are the events in same process and event x occur before y then x → y then T(x) < T(y). If x is event of sending the message by one process and y is the event of receiving same message by other process then x → y then all the processes should agree on the values T(x) and T(y) with T(x) < T(y). Clock time T assumed to go forward and should not be decreasing. Time value can be corrected by adding positive value.
- In Fig. 3.2.1 (a), three processes A, B and C are shown. These processes are running on different machines. Each clock runs with its own speed. Clock has ticked 5 times in process A, 7 times in process B and 9 times in process C. This rate is different due to different crystal in timer.

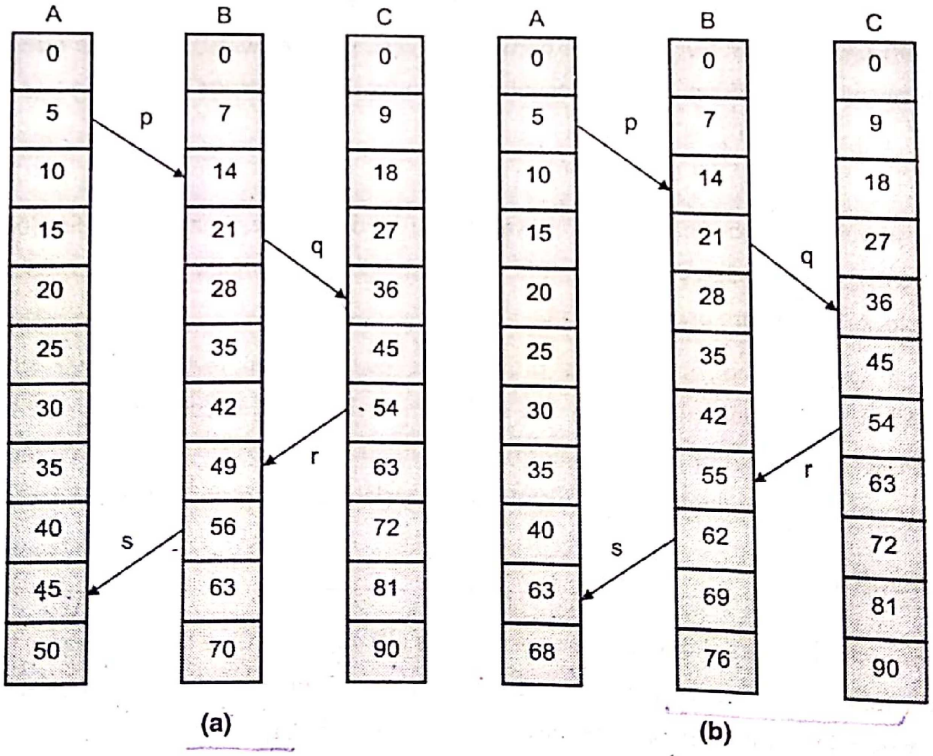


Fig. 3.2.1

- At time 5, process A sends message p to process B and it is received at time 14. Process B will conclude that, message has taken 9 ticks to travel from process A to process B, if message contains sending time. Same is true about message q from process B to process C.
- Process B sends message s at time 56 and it is received by process A at time 45. Similarly, Process C sends message r at time 54 and it is received by process A at time 49. This should not happen and it should be prevented. It shows sending time is larger than receiving time.
- Message r from process C leaves at time 54. As per happen before relation, it should reach at process B at time 55 or later. Same is true for message s which should reach to process A at time 63 or later. Receiver fast forwards the clock by one more than the sending time. That is why the sender always sends sending time in message. This is shown in Fig. 3.2.1 (b). If two events occur in sequence, then clock must tick at least once. If process sends and receive message in quick succession, then clock should be advanced at least by 1 between these two send and receive events.
- If two events occur at the same time in different processes, then it should be separated by decimal point. If such events occur at time 20 then former should be considered at 20.1 and later at 20.2.
  - If $x \rightarrow y$ in the same process then $T(x) < T(y)$.
  - If x is event of sending the message y is the event of receiving the message then $T(x) < T(y)$.
  - For all distinguishing events x and y, $T(x) \neq T(y)$.
- Total ordering of all the events can be carried out with above algorithm.

3.2.2 Application of Lamport Timestamp : Total Order Multicasting

- Ordering of events between communicating processes is very important to achieve consistency at different sites. For example, two servers in different cities A and B maintains bank database. For example, customer accounts balance initially is Rs 1000. If customer deposited Rs 100 in city A and at the same time branch manager gives 1% interest on account balance in city B.
- These are two events related to same copy of database that is present on two different servers. Lets us say these events as event A (updating account balance by adding Rs 100) and event B (updating account balance by adding 1% interest). If these events executes in different order at these databases then inconsistency in account balances would occur.
- In city A, if event A is executed first and then suppose event B is executed. In this case account balance in city A will be updated as : 1000+100 = 1100 and then adding Rs 11 interest = **Rs 1111**. In city B, if event B is executed first and then suppose event A is executed. In this case, account balance in city B will be updated as: 1000+10 = 1010 and then adding Rs 100 = Rs **1110.**
- These Event A and B messages arrive at both sites A and B in order explained above due to communication delay in network. If clocks of all machines in network show same time then ordering of messages can be done as per timestamp when message was sent. Practically, it is not possible.
- Lamport proposed notion of logical time to arrange the messages at different sites in same order for execution. **Totally order multicast** is multicasting operation in which all messages are delivered to receiver in same order.

- Suppose group of processes multicast messages to each other. Sender of the message also receives its own sent message. Each message carries its sending time. Assume all the processes are receiving messages in the order they were sent and three is no loss of messages.
- Each process sends acknowledgement (ACK) for the received message. Hence as per Lamport algorithm, timestamp of the ACK will be greater than the timestamp of received message. Here Lamport's clock ensures that, no any two messages will have same timestamp. Timestamps also gives global ordering of events.

3.2.3 Vector Clocks

- In Lamport's clock, if $x \rightarrow y$ then $T(x) < T(y)$. But, it does not tell about relationship between event x and y. Lamport clocks do not capture causality.
- Suppose user and hence processes join discussion group where processes post articles and post reaction to these posted articles. Posting of articles and reactions are multicast to all the members of the group.
- Consider the total order multicasting scheme and reactions should be delivered after their corresponding postings. The receipt of article should be causally preceded by posting of reaction. Within group, receipt of reaction to an article must always follow the receipt of that article. For independent articles or reactions, order of delivery does not matter.
- The causal relationship between messages is captured through vector clocks. Event x is said to be causally precede event y, if $VT(x) < VT(y)$. Here, VT(x) is a vector clock assigned to event x and VT(y) is vector clock assigned to event y.
- It is assumed that, each process P_i maintains a vector V_i having following properties.
  - $VT_i[i]$ maintains number of events that have taken place at process P_i.
  - If $VT_i[j] = k$ then P_i is aware about k number of events occurred at process P_j.
- Each time event occurs at process P_i, $VT_i[i]$ gets incremented by 1 and it ensures to satisfy first property. Whenever process P_i sends the message m to other process, it sends its current vector with timestamp v_t along with it (piggybacking), which ensures to satisfy second property said above.
- Because of this, receiver knows the number of events occurred at other processes before receipt of message m from process P_i and on which message m may causally depends.
- When process P_j receives message m, it modifies each entry in its table to maximum of the $VT_j[k]$ and vt[k]. The vector at P_j now shows the number of messages it should receive to have at least seen the same messages that preceded the sending of message m. After this every VTj[j] is incremented by 1.

3.3 Election Algorithms

- To fulfill the need of processing in distributed system, many distributed algorithms are designed in which one process acts as coordinator to play some special role. This can be any process. If this process fails, then some approach is required to assign this role to other process. In this case, election is required to elect the coordinator.
- In group of processes, if all processes are same then the only way to assign this responsibility is on the basis of some criterion. This criterion can be the identifier which is some number assigned to process. For example, it could be network address of machine on which process is running. This assumption considers that only one process is running on the machine.

- Election algorithm locates process with highest number in order to elect it as a coordinator. Every process is aware about the process number of other processes. But, processes do not know about which processes are currently up or which ones are currently crashed. Election algorithm ensures that, all processes will agree on newly elected coordinator.

3.3.1 Bully Algorithm

- In this algorithm, if any process finds that the coordinator is not responding to its request, then it initiates the election. Suppose, process P initiates the election. Election is carried out as follows :
  - Process P sends **Election** message to all the process having higher number than it.
  - If no one replies to **Election** message of P then it wins the election.
  - If anyone higher number process responds then it takes over. Now P's work is done.
- Any process may receive **Election** message from its lower-numbered colleague. Receiver replies with **OK** message to sender of **Election** message. This reply conveys the message that, receiver is alive and will take over the further job. If receiver is currently not holding an election, it starts election process.
- Eventually all processes will quit except the highest number process. This process wins the election and send **Coordinator** message to all processes to convey the message that he is now new coordinator.
- In Fig. 3.3.1, initially process 17 (higher-numbered) is coordinator. Process 14 notices that, coordinator 17 has just crashed as it has not given response to request of process 14. Process 14 now initiates election by sending **Election** message to process 15, 16 and 17 which are higher-numbered processes.

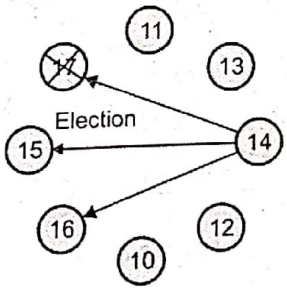


Fig. 3.3.1

- As shown in Fig. 3.3.2, process 15 and 16 responds with **OK** message. Process 17 is already crashed and hence does not send reply with **OK** message. Now, job of process 14 is over.

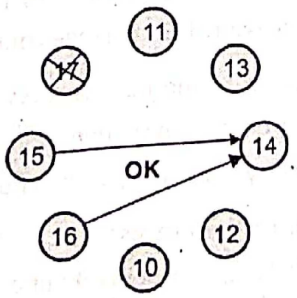


Fig. 3.3.2

- Now process 15 and 16 each hold an election by sending **Election** message to its higher-numbered processes. It is shown in following Fig. 3.3.3 (a). As process 17 is crashed, only process 16 replies to 15 with **OK** message. This is shown in Fig. 3.3.3 (b).

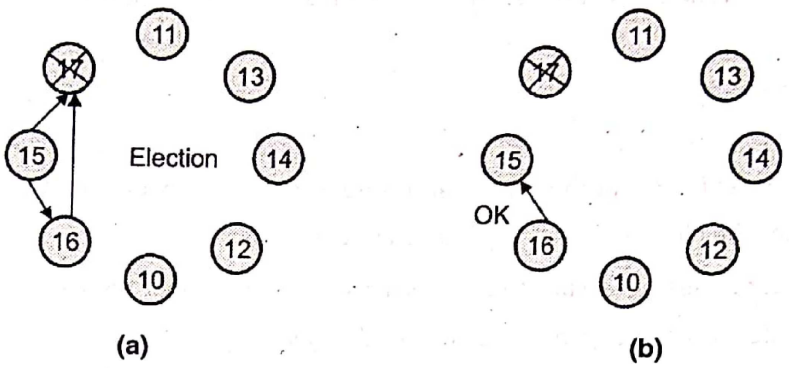


Fig. 3.3.3

- Finally, process 16 wins the election and send **coordinator** message to all the processes to inform that, it is now new coordinator as shown in Fig. 3.3.4. In this algorithm, if two processes detect simultaneously that the coordinator is crashed then both will initiate the election. Every higher-number process will receive two **Election** messages. Process will ignore the second **Election** message and election will carry on as usual.

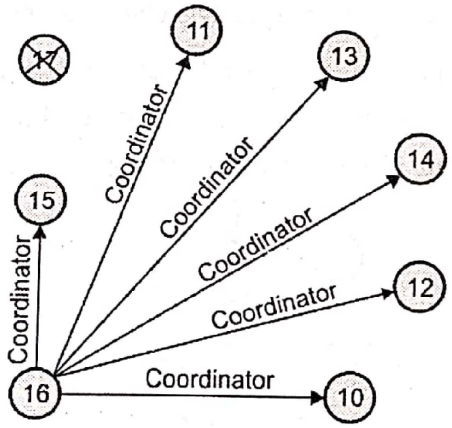


Fig. 3.3.4

3.3.2 Ring Algorithm

- This ring algorithm does not use token and Processes are physically or logically ordered in ring. Each process has its successor. Every process knows about who is its successor. When any process notices that coordinator is crashed, it builds the **Election** message and sends to its successor. This message contains the process number of sending process.
- If successor is down then process sends message to the next process along the ring. Process locates next running process along the ring if it finds many crashed processes in sequence. In this manner, the receiver of **Election** message also forwards the same message to its successor by appending its own number in message.
- In this way, eventually message returns back to the process that had started the election initially. This incoming message contains its own process number along with process numbers of all the processes that had received this message.

- At this point, this message gets converted to **coordinator** message. Once again this **coordinator** message is circulated along the ring to inform the processes about new coordinator and members of the ring. Off course, the highest process number is chosen from the list for new coordinator by process which has started the election.

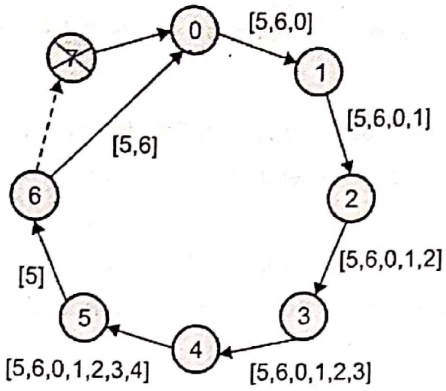


Fig. 3.3.5

- In Fig. 3.3.5, process 5 notices crash of coordinator which was initially process 7. It then sends **Election** message to process 6. This message contains number of the process 5. As process 7 is crashed, process 6 forward this message to its new successor process 0 and append its number in the same list. In this way message is received by all the processes in ring.
- Eventually, message arrives at process 5 which had initiated the election. Highest number in this list is 6. So message is again circulated in previous manner to inform all the processes that, process 6 is now new coordinator.

3.3.3 Elections in Wireless Networks

- Distributed system may also have nodes connected through wireless networks. In this system, election of coordinator is carried out with first step as build network phase followed by election. Let the wireless ad hoc network includes nine nodes P, Q, R, S, T, U, V, W, X and Y each having capacity in the range 0 to 8.
- It is assumed that node cannot move. Let p is the source node which initiates the election. There are minimum of two nodes connected to each node as shown in Fig. 3.3.6. Values within the circle show capacity of respective node. Total numbers of interconnections are kept less to have minimum network traffic. Let P be the source node which initiates election.

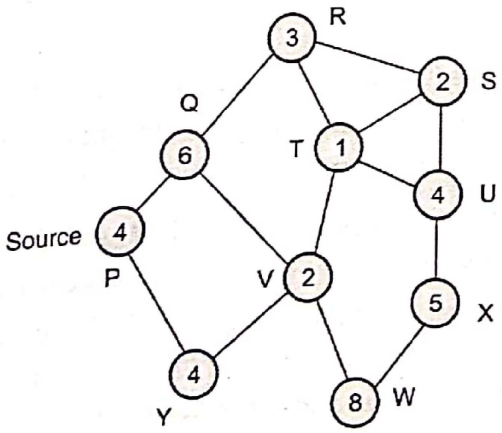


Fig. 3.3.6

- Let any node (consider, source as node A) initiates election and send **Election** message to its immediate neighbors which are in its range. When any node receives this **Election** message first time, it chooses sender of message as its parent. The receivers then send out this received **Election** message to its immediate neighbors which are in its range except to its chosen parent. When node receives **Election** message from other node except parent, it sends its acknowledgement.
- The build tree phase is shown in Fig. 3.3.7. Node P broadcast **Election** message to Q and Y. Node Q broadcast **Election** message to R and V. In this scenario, Y is slow in sending message to V. As a result, V receives message fast from Q compared to from Y. Node V broadcast message to T and W and node R broadcast to S and T. After this, node T receives the broadcast message from V before R. Node W forward the message to T and X and further, T forward message to node U which gets the message faster from T and S. This process continues till traversing of all the nodes is finished.

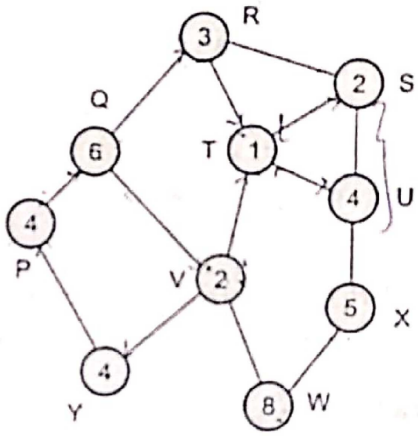


Fig. 3.3.7

- In this method of forwarding the message, each process records the sender node of message and the node to which message is forwarded. Receiving node of the message prepares message in order to reply in the form [node, capacity] and send it to the node from which message was received. In following Fig. 3.3.8, reporting of best node to source is shown.
- The first reply node is generated from the last node and same return path is traversed. The receiver of replied message selects the message of highest capacity if it receives multiple messages. This message it then forwards to next node along the return path.
- As shown in Fig. 3.3.8, The reply message [U,4] would be initiated from U to T and [X,5] from X to W. Node T forward the message as it is [U,4] to V. During the same time, W receives the message [X,5] and as its capacity greater than that of X, it replaces the message to [W,8] and forward it to V.
- As stated previously, node V selects the higher capacity message [W,8] and forward it to node Q. In the meantime, S initiate the reply message [S,2] to node R which compares its own capacity and forward the new message [R,3] to node Q. The node Q now receives two messages. One is [R,3] from R and [W,8] from V. The best node value [W,8] is sent to source node P. AT the same time, P also receives message [Y,4] from Y. It then selects the higher value out of these as [W,8] and reports W as a best node. Finally W is elected as new coordinator.

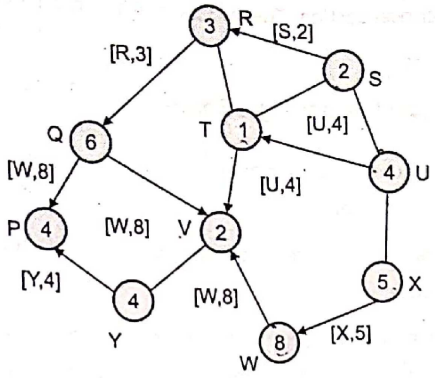


Fig. 3.3.8

3.4 Mutual Exclusion

- In environment where multiple processes are running, processes shares resources. When process has to read or update shared data structures, it enters in critical section (CS) in order to achieve mutual exclusion.
- This ensures that, no other process enters the critical in order to use shared data structure when a process is already using it.

3.4.1 Distributed Mutual Exclusion

- In uniprocessor system, semaphore, monitor and other tools are used to achieve mutual exclusion. In distributed system, since processes runs of different machines and also resources are distributed on many machines, different solutions are required to achieve mutual exclusion. In distributed system shared memory may does not exist. Hence, algorithms based on message passing are used to achieve mutual exclusion in distributed system.
- In distributed system, the problem of mutual exclusion becomes much more complex as shared memory is not available and there can be problem of lack of global physical clock. The message delays can be also unpredictable. Therefore, node in the system cannot have the complete knowledge of the overall state of the system.

3.4.2 Classification of Mutual Exclusion Algorithms

- Many algorithms are developed to achieve mutual exclusion in distributed system. These algorithms differ as per topology of network and amount of information which each node maintains about other nodes. These algorithms are classified as follows :
- **Non-token based Algorithms :** These algorithms require two or more consecutive exchanges of messages between nodes. These algorithms are based on declaration, as algorithm permits process to enter in critical section on any node as declaration of local variable on that node permits it. As declaration of local variable becomes true, process enters in critical section on that node. At any node, the mutual exclusion is enforced in this class of algorithms due to true value of local variable declared.
- **Token based Algorithms :** These algorithms uses unique tokens shared among nodes in distributed system. If process on any node receives token and process it, then it can enter in critical section as long as it holds the token.

- When process finish execution in critical section. These classes of algorithms differ on the basis of the way in which nodes/processes searches the tokens.

3.4.3 Requirements of Mutual Exclusion Algorithms

- Algorithm should guarantee that only one process should be in critical section to read or update shared data. Mutual exclusion algorithms should have following features.
- **Algorithms should not involve in deadlocks :** More than two nodes in distributed system should not wait endlessly for the messages which will never be received.
- **Avoidance of Starvation :** If some nodes are repetitively executing critical section and some any other node is indefinitely waiting for the execution of critical section, such situation should be avoided. In finite time, every requesting node should get opportunity to execute in critical section (CS).
- **Fairness :** As there is absence of physical global clock, either requests at any node should be executed in the order of arrival or in the order these were issued. This is due to consideration of logical clocks.
- **Fault Tolerance :** In case of any failure in the course of carrying the work, if any failure occurs, then mutual exclusion algorithms should reorganize itself to carry out the further designated job.

3.4.4 Performance Measure of Mutual Exclusion Algorithms

Following four metrics measure the performance of mutual exclusion algorithms in distributed system.

1. The number of required messages for invocation of critical section in first attempt.
2. The delay incurred between the moments when one node leaves the CS and before next node enters the CS. This time is also called as synchronization delay. In this case, more messages exchange may require during this delay duration.
3. First node's CS request arrives. Then its request message is sent out to enter in CS. Response time is the time interval between the request message sent out to enter in CS and when node exit the CS. It is shown in Fig. 3.4.1.
4. The rate at which system executes request that arrives for critical section (CS) which is called as system throughput.

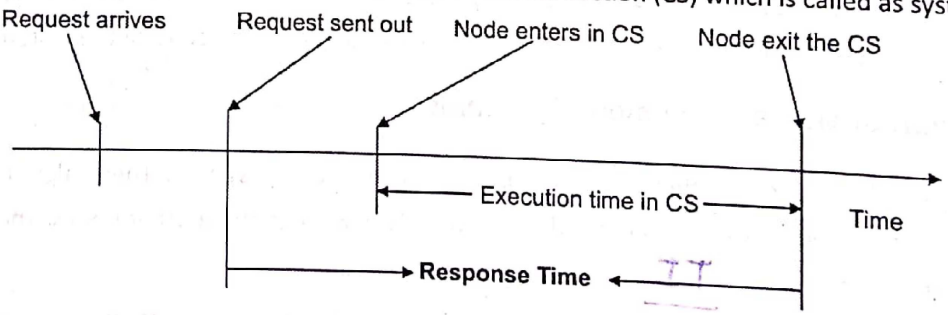


Fig. 3.4.1

3.4.5 Performance in Low and High Load Conditions

In low load condition of the system, there is rarely more than one request at the same time in system. In high load condition of the system, there is often pending request for mutual exclusion at the node (site). A node rarely remains idle in high load condition of the system. In best case performance, performance metrics achieves best possible values.

3.5 Non-Token Based Algorithms

- In non-token based mutual exclusion algorithms, a particular node communicates with set of nodes to decide who should enter in critical section next. Requests to enter in CS are ordered on the basis of timestamps in non-token based mutual exclusion algorithms.
- Simultaneous requests are also handled by using timestamp of the request. Lamport's logical clock is used and request for CS having less timestamp value gets priority over requests having larger timestamp values.

3.5.1 Lamport's Algorithm

Request for CS

- In this algorithm, for all $i : 1 \leq i \leq N :: R_i = \{S_1, S_2, \ldots, S_N\}$, that is there are N number of sites. Single process run on each site. Site S_i represents process P_i. The **request_queue$_i$**, is maintained by every site i to store mutual exclusion requests in the order of timestamps. It is necessary to deliver messages in FIFO order between each pair of sites.
- Whenever site S_i want to enter in critical section (CS), it sends a message **REQUEST(ts_i, i)** to all the sites in its request set R_i and puts the message in **request_queue$_i$**. The timestamp of the request is (ts_i, i).
- The site S_j sends timestamped **REPLY** message to Si after receiving **REQUEST(ts_i, i)** message from S_i. This request of S_i, site S_j puts in its **request_queue$_j$**.

Executing the CS

- Site Si can enter in CS if following two conditions are satisfied.
- **C1 :** Site S_i has received message from all the sites with timestamp larger than (ts_i, i).
- **C2 :** S_i's request is at the top of **request_queue$_i$**.

Exiting the CS

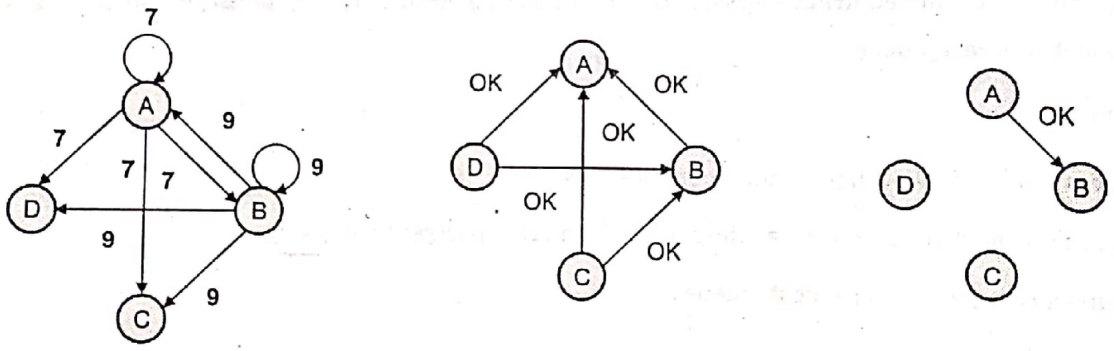
- When S_i exit the CS, it removes the request from top of **request_queue$_i$**. S_i then sends timestamped **RELEASE** message to all the sites in its request set.
- After receiving **RELEASE** message from S_i, site S_j removes S_i's request from its **request_queue$_j$**.

Correctness

- Lamport's algorithm ensures that, only one process will be execution in CS at any time. If we assume both S_i and S_j executing in S then both have their own requests at the top of their request queues and C1 holds.
- If request of S_i has lesser timestamp than request of S_j ($T(S_i) < T(S_j)$), then as per C1 and FIFO delivery of channels, request of S_i should present in **request_queue$_j$**. If S_j is executing then its request is at top of **request_queue$_j$**. It is not possible as $T(S_i) < T(S_j)$. Hence algorithm achieves mutual exclusion.

3.5.2 Ricart–Agrawala's Algorithm

- Ricart–Agrawala's algorithm is an optimization of Lamport's algorithm. The working of the algorithm is as follows :
- When process wants to enter in CS, it sends the **REQUEST** message to all other processes. This **REQUEST** message contains the name of CS in which it (sender process) wants to enter, its process number and the current time. This message also conceptually delivers to the sender. Each message is acknowledged by the receiver.
- The receiver of the message takes action as per its current state with respect to name of the CS mentioned in the received message. Following three cases occur.
  1. Currently if receiver is not in CS and also does not want to enter it then it replies **OK** message to sender.
  2. If receiver is already running in CS then it does not reply. It puts the message in its queue.
  3. If receiver is about to enter in CS and yet has not done then it compares the timestamp in incoming request message with the timestamp in request message which it has already sent to other processes. The lowest one wins. Receiver sends OK message to the sender if incoming request message has lower timestamp value. If its own request message has lower timestamp value then it puts the incoming request message in its queue and does not send any reply.
- When requesting process for CS receives reply as **OK** messages from all the other processes, it enters in CS. If process is already in CS then it replies OK message to other requesting process after coming out of CS.



(a) (b) (c)

Fig. 3.5.1

- As shown in Fig. 3.5.1 (a), process A and B both wants to enter in CS simultaneously. Process A and B both sends **REQUEST** message to everyone with timestamp 7 and 9 respectively.
- As shown in Fig. 3.5.1 (b), as process C and D both are not interested in entering the CS, both sends OK message as reply to sender processes A and B. Process A and B both see conflict and compare timestamps. In this case process A wins as its message timestamp 7 is lower than the timestamp of message sent by process B, which is 9.
- The process A enters in CS and keeps request of process B in its queue. When process A exits the critical section, it sends OK message to process B so that later process B will enter in CS. It is shown in Fig. 3.5.1 (c). Hence, algorithm ensures the mutual exclusion in distributed system as lower timestamp process is allowed to enter in CS in conflict situation.

- If we consider n number of processes in the system, then process sends (n–1) REQUEST messages to enter in CS. It receives (n–1) OK messages from other processes to get permission to enter in CS. Hence, 2(n–1) messages are required to exchange to enter in CS. There is no single point of failure.
- In this algorithm, if reply or request if lost then sender gets blocked due to time outs. Sender of request message then may conclude that receiver is dead or it may try continuously sending the request. Sender has to wait for **OK** message from destination.
- For large size group of processes, each process should maintain list of members. Group members may leave the group, new members may join the group or group member may crash. Hence, algorithm produces best performance and success for small size groups. This algorithm is complex, slower, expensive and less robust.

3.5.3 Centralized Algorithm

- In this algorithm, a central coordinator process is responsible to give permission to other processes to enter in critical section (CS). Coordinator process keeps track of which CS is busy or currently in execution. Process which wants to enter in CS sends **REQUEST** message to coordinator stating the name of CS.
- If CS is free then coordinator sends **OK** message to requesting process to grant permission to enter in stated CS in message. If in stated CS, already other process is executing then it puts the request message in its queue.
- When currently executing process in CS exits, it sends **RELEASE** message to coordinator process. As a result, coordinator knows that CS is now free and it takes request from the queue to grant permission to enter in CS. If requesting process is still blocked, it unblocks and enters in CS.

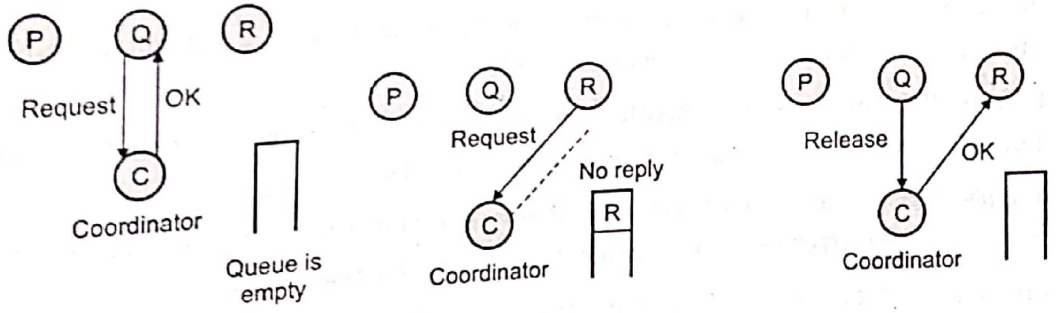


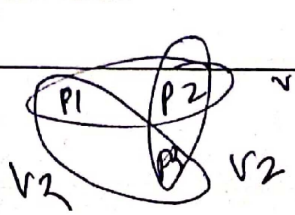
Fig. 3.5.2

- As shown in Fig. 3.5.2 (a), Process C is coordinator and currently no process is executing in CS. Process Q sends **REQUEST** message to coordinator and it permits process Q to enter in CS by replying with **OK** message.
- As shown in Fig. 3.5.2 (b), process R also send **REQUEST** message to coordinator to get permission to enter in same CS. As coordinator knows process R is already in CS, it puts R's request in its queue.
- As shown in Fig. 3.5.2 (c), when process Q exit the CS it sends **RELEASE** message to coordinator. Now coordinator knows that CS is free, it takes message from queue and grants permission to process R by replying with **OK** message. Now process R enters in CS.

- This algorithm guarantees mutual exclusion as at a time one process is allowed to enter in CS. This algorithm requires 3 messages to enter and exit the CS (Request, OK and Release). Limitation of the algorithm is that coordinator may crash.

3.5.4 Maekawa's Algorithm

- In Ricart–Agrawala's algorithm, process should get permission from all other processes to enter in CS. In Maekawa's algorithm, to enter in CS, process has to obtain permission from subset of the processes as long as subsets used by any two processes overlaps. Process need not obtain permission from all the processes in order to enter in CS.
- Processes carry out voting to one another in order to enter in CS. If process collects enough votes, then it can enter in CS. The conflict is resolved by the processes common to both sets of processes. Hence, processes in intersection of two sets of voters guarantee the safety property that; only one process enters in CS, by voting to a single candidate to enter in CS.
- For all the processes 1 to N, there is associated voting set V_i with process P_i. The set V_i is subset of all the processes numbered from 1 to N (P_1, P_2, P_3 ..... P_N). For all processes i and j between 1 to N, this set satisfies following four properties.
- Process P_i is member of V_i.
  1. Voting sets V_i and V_j contains minimum one common member.
  2. Size of voting set is same. Let it be K.
  3. Each process P_j belongs to X of the voting set V_i.
- In this algorithm, optimal solution is provided to minimize the value of voting set size K specified in property 3. This solution guarantees mutual exclusion for K - √ N and X equal to K.
- In order to get access to CS, process P_i sends **REQUEST** message to K–1 members of V_i. Process P_i can enter is CS if it receives reply from K–1 processes of the set. Let Process P_j belongs to set V_i.
- Process P_j sends reply message immediately to process P_i after receiving the **REQUEST** message from it unless either its state is **HELD** or it has replied (**VOTED** message) since it last received the **RELEASE** message.
- When process receives **RELEASE** message, it takes out the outstanding **REQUEST** messages from front of the non-empty queue and sends reply as **VOTE** message. To exit the CS, P_i sends **RELEASE** messages to all (K–1) members of V_i.
- This algorithm guarantees mutual exclusion as common processes in V_i and V_j should cast votes for both processes P_i and P_j which simultaneously want to enter in CS. But this algorithm permits to cast at the most one vote between consecutive receipts of **RELEASE** message. This is impossible and hence, ensures mutual exclusion.
- This algorithm leads to deadlock situation. Let $V_1 = \{P_1, P_2\}$, $V_2 = \{P_2, P_3\}$ and $V_3 = \{P_3, P_1\}$. If all three processes P_1, P_2 and P_3 simultaneously want to enter in CS then P_1 may reply to P_2 while holding P_3. Process P_2 to reply P_3, it may hold P_1. Process P_3 to reply P_1, it may hold P_2. As in this situation each process will receive one out of two reply, none can proceed further.
- In this algorithm, if process crashes and it is not the member of voting set then its failure does not affects other processes. The steps of the algorithm are summarized as shown below.
  1. Initially state is **RELESED** and **VOTED** is set as false.



2. If P_i wants to enter in CS then its state is **WANTED** and it sends **REQUEST** messages to all the processes $(V_i - \{P_i\})$ in its set of voting size K. It waits till arrival of (K-1) reply. State = **HELD**.
3. If P_j receives reply from P_i $(i \neq j)$ and if state = **HELD** or **VOTED** = true then puts the **REQUEST** of P_i in queue and don't reply. Otherwise reply to P_i and **VOTED** = true
4. For P_i to exit the CS, state = **RELESED** and reply **RELESE** message to all processes $V_i - \{P_i\}$.
5. When process P_j receives **RELESE** from P_i where $(i \neq j)$ and if queue of request have messages (non-empty) then remove message from front (head) of queue. Suppose it is from P_k. Then send reply to P_k. Now **VOTED** = true. Otherwise **VOTED** = false.

3.6 Token Based Algorithms

- Token based algorithm makes use of token and this unique token is shared among all the processes. The process possessing token can enter in CS. There are different token based algorithms based on the way process carry out search for the token.
- These algorithm uses sequence numbers instead of timestamps. Each request for the token has sequence number and these sequence numbers for processes advances independently. Whenever process request for the token, its sequence number gets incremented. This helps is differentiating old and current request for the token. As only process acquiring token enters in CS, algorithms ensure mutual exclusion.

3.6.1 Suzuki-Kasami's Broadcast Algorithm

- In this algorithm, suppose process want to enter in CS and does not holds token then it broadcast a **REQUEST** message to all the processes. After receiving the **REQUEST** message, process holding token sends it to requesting process.
- If process is already in CS when receives the **REQUEST** message then it sends token only after it has exited the CS. Process holding token can repeatedly enter in CS till it has a token. Following two issues are important to consider in this algorithm.
  - To differentiate outdated **REQUEST** messages from current **REQUEST** messages.
  - Determining the process having outdated request for CS.
- Let **REQUEST(P_j, n)** is the **REQUEST** message of process P_j. Here, n (n = 1, 2, 3...) is a sequence number that shows P_j is requesting its n^{th} CS execution. Process P_i maintains array of integers $RN_i[1...N]$ where $RN_i[J]$ indicates largest sequence number received from P_j. If process P_i receives **REQUEST(P_j, n)** then this message will be outdated if $RN_i[J] > n$. When process P_i receives **REQUEST(P_j, n)**, it sets $RN_i[J] = \max(RN_i[J], n)$.
- Token has queue of requesting processes and array of integers SN[1...N]. Where SN[j] indicates the sequence number of the request that has executed by process P_j most recently. Process P_i sets $SN[i] = RN_i[i]$ whenever it finishes its execution of CS. It shows that its request corresponding to sequence number $RN_i[i]$ has been executed.
- At process P_i, if $RN_i[j] = SN[j] + 1$ then process P_j is currently requesting the token. The process which has executed the CS now checks this condition for all j's to know about all the processes that are requesting the token and puts their IDs in requesting queue if not already there. Process then sends token to the process at head of requesting queue.
- This algorithm is simple and efficient. It requires 0 or N messages per CS entry. Synchronization delay is 0 or T. If process has idle token while requesting then no message or zero delay is required.

Algorithm

Request for CS

(i) If process P_i wants to enter in CS and does not have token, then it increments its sequence number $RN_i[i]$. It then sends **REQUEST(P_i, sn)** message to all the processes. In this message sn is updated value of $RN_i[i]$.

(ii) After receiving this request message, process P_j sets $RN_j[i]$ to max($RN_j[i]$, sn). If P_j has token then it sends to P_i if $RN_j[i] = SN[i] + 1$.

Executing the CS

After receiving the token, process P_i executes in CS.

Exiting the CS

Process P_i carry out following updates after it finishes execution of CS.

(i) It sets $SN[i] = RN_i[i]$.

(ii) For all processes P_j, if their IDs does not belong to the token queue then append these IDs in it if $RN_i[j] = SN[j] + 1$.

(iii) If token queue is still has some messages in it then delete ID at head of queue and sends token to the process of that ID.

3.6.2 Singhal's Heuristic Algorithm

- In this algorithm, each process keeps information about states of other processes in the system. This information is used by the process to determine the set of processes that likely to have the token. The process then requests the token from these processes to reduce the number of messages needed to enter in CS.
- The request for token should go to the process which is holding the token or going to obtain it in near future. Otherwise there can be chances of deadlock or starvation.
- Process P_i keeps two arrays $PV_i[1..N]$ and $PN_i[1..N]$ to store the state and highest known sequence number of each process respectively. Token also maintains two arrays TPV[1..N] and TPN[1..N]. Outdated requests are determined by using sequence numbers. Process can be in one of the following state.
  - **REQ :** Requesting the CS:
  - **EXE :** Executing the CS
  - **HOLD :** Holding the idle token.
  - **NONE :** None of above.

Initially arrays are set as follows

- For every process P_i, for i = 1 to N do
  - Set $PV_i[j]$ = **NONE** for j = N to i
  - Set $PV_i[j]$ = **REQ** for j = i – 1 to 1
  - Set $PN_i[j] = 0$ for j = 1 to N.
- Initially process P_1 is in **HOLD** state. Hence, $S_1[1]$ = **HOLD**.
- For the token TPV[j] = **NONE** and TPN[j] = 0 for j = 1 to N.

Above initialization ensures that either $PV_i[j]$ = **REQ** or $PV_j[i]$ = **REQ.** Hence, for any two processes sending request at the same time, one process will always send token request to other. This guarantees that processes are not inaccessible from each other and REQUEST message from process will be delivered to process holding the token or to the process which will hold it in near coming time.

Algorithm

Request for CS

1.If process P_i wants to enter in CS and does not have token, then it takes following action.

- Sets $PV_i[i]$ = **REQ.**
- Sets $PN_i[i] = PN_i[i] + 1$.
- P_i then sends **REQUEST(P_i, sn)** message to all the processes P_j for which $PV_i[j]$ = **REQ.** In this message **sn** is updated value of $PN_i[i]$

2.After receiving the message **REQUEST(P_i, sn)** by P_j, it discards this message if $PN_j[i] \geq sn$ as message is outdated now. Otherwise, it (P_j) sets $PN_j[j]$ = sn and carry out following activities as per its current state.

- $PV_j[j]$ = **NONE :** Set $PV_j[i]$ = **REQ.**
- $PV_j[j]$ = **REQ :** If $PV_j[i] \neq$ **REQ** then set $PV_j[i]$ = **REQ** and send a **REQUEST(P_j, $PN_j[j]$)** message to P_i (Else do nothing).
- $PV_j[j]$ = **EXE :** Set $PV_j[i]$ = **REQ.**
- $PV_j[j]$ = **HOLD :** Set $PV_j[i]$ = **REQ,** TPV[i] = **REQ,** TPN[i] = sn, $PV_j[j]$ = **NONE** and send the token to process P_i.
- $PV_j[j]$ = **NONE :** Set $PV_j[i]$ = **REQ.**
- $PV_j[j]$ = **REQ :** If $PV_j[i] \neq$ **REQ** then set $PV_j[i]$ = **REQ** and send a **REQUEST(P_j, $PN_j[j]$)** message to P_i (Else do nothing).
- $PV_j[j]$ = **EXE :** Set $PV_j[i]$ = **REQ.**
- $PV_j[j]$ = **HOLD :** Set $PV_j[i]$ = **REQ,** TPV[i] = **REQ,** TPN[i] = sn, $PV_j[j]$ = **NONE** and send the token to process P_i.

Executing the CS

3.Process P_i executes CS after receiving the token. Prior to entering in CS it sets $PV_i[i]$ = **EXE.**

Exiting the CS

4. After finishing the execution of CS, process P_i sets $PV_i[i]$ = **NONE** and TPV[i] = **NONE.** After this, it updates its array as mentioned below.

```
For all Pj for j = 1 ....N do
        If PNi[j] > TPN[j]
                Then
                        // Use token information to update local information.
                        {TPV[j] = PVi[j]; TPN[j] = PNi[j]}

                Else
                        // Use local information to update token information.
                        {PVi[j] = TPV[j]; TNi[j] = TPN[j]}
```

5. If for all j, $PV_i[j]$ = **NONE** then set $PV_i[i]$ = **HOLD,** otherwise send token to process P_j such that $PV_i[j]$ = **REQ.**

- Fairness of algorithm will rely on selection of process P_i, as no queue is available in token. In this algorithm, negotiation rules are used in order to ensure fairness. Algorithm requires average N/2 messages in low to moderate load condition. During high load, when all processes requests CS, N number of messages are required. Synchronization delay required is T.

Example

Assume there are 3 processes in the system. Initially:

- Process 1 : $PV_1[1]$ = **HOLD**, $PV_1[2]$ = **NONE**, $PV_1[3]$ = **NONE**. $PN_1[1]$, $PN_1[2]$, $PN_1[3]$ are 0.
- Process 2 : $PV_2[1]$ = **REQ**, $PV_2[2]$ = **NONE**, $PV_2[3]$ = **NONE**. PNs are 0.
- Process 3 : $PV_3[1]$ = **REQ**, $PV_3[2]$ = **REQ**, $PV_3[3]$ = **NONE**. PNs are 0.
- Token : TPVs are NONE. TSNs are 0.

Assume P_2 is requesting token.

- P_2 sets $PV_2[2]$ = **REQ**, $PN_2[2] = 1$.
- P_2 sends **REQUEST(P_2,1)** to P_1 (As only P_1 is set to REQ in PV[2])
- P_1 receives the **REQUEST**. Accepts the **REQUEST** since $PN_1[2]$ is smaller than the message sequence number.
- As $PV_1[1]$ is **HOLD**: $PV_1[2]$ = **REQ**, TSV[2] = REQ, TSN[2] = 1, $PV_1[1]$ = **NONE**.
- Send token to process P_2.
- P_2 receives the token. $PV_2[2]$ = **EXE**. After exiting the CS, $PV_2[2]$ = TPV[2] = **NONE**.
- Updates PN, PV, TPN, TPV. Since nobody is requesting, $PV_2[2]$ = **HOLD**.
- Suppose P_3 issues a **REQUEST** now. It will be sent to both process P_1 and P_2. Only P_2 responds since only $PV_2[2]$ is **HOLD** ($PV_1[1]$ is **NONE** at the present).

3.6.3 Raymond's Tree-Based Algorithm

- In this algorithm, processes are organized in directed tree. In this tree, edges of the tree are towards the root (process) which currently having token. Each process in tree maintains local variable **HOLDER (H)** that points to an immediate neighbor on path towards root process. The local variable **HOLDER (H)** can also point to the root. The process which currently holds the token has privilege to be root.
- If we chase **HOLDER (H)** variables, every process has path towards process holding token. At root, **HOLDER (H)** points to itself. Every process maintains FIFO **Request-queue** with it which stores requests sent by neighboring processes but has not so far sent the token. Following are the steps in algorithm.
- Consider two processes P_i and P_j where P_j is neighbor of P_i. If **HOLDER** variable $H_i = j$. It means P_i has information that, root can be reached through P_j. In this case, undirected edge between P_i and P_j converts to directed edge from P_i and P_j. Hence, H variable helps to trace path towards process holding the token.

- Consider undirected graph with processes P_1 to P_7 with holder variable H. Process P_1currently hold the token. P_1 and P_2 are the neighbors of root P_1; P_4 and P_5 of P_2 and processes P_6 and P_7 of P_3. Every process keeps information about its neighbor and communicates with only to its immediate neighboring processes which lead to root process P_1.
- Suppose P_4 wants to enter in CS. Process P_4 sends **REQUEST** message to process P_2 as $H_4 = 2$. Process P_2 in turn forward this message to process P_1 ($H_2 = 1$). This **REQUEST** message gets forwarded by the processes along the minimal spanning path on the behalf of the process which initially sends **REQUEST** message to root having token. The Forwarded REQUEST message is noted by using variable "asked" in order to prevent the resending of the same message.

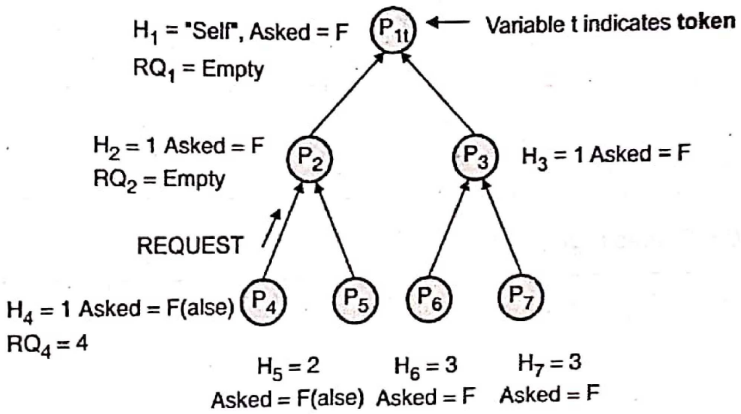


Fig. 3.6.1 : Process P_4 sends REQUEST message to P_2 which forwards it to process P_1

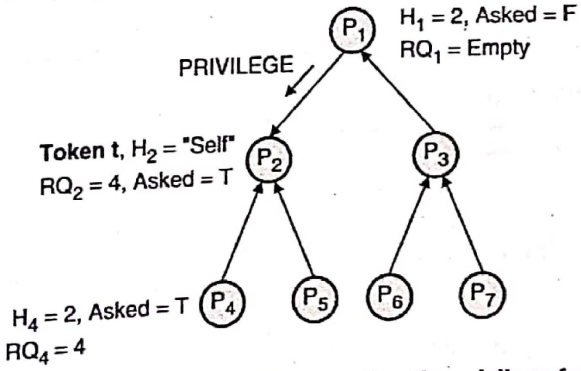


Fig. 3.6.2 : Process P_2 receives token after granting the privilege from process P_1

- Initially token was with process P_1, When it receives REQUEST message (H_1 = "Self") and if not executing CS, it sends PRIVILEGE message to its neighbor P_2. Process P_2 was the sender of REQUEST message to P_1. Now P_1 resets holder variable $H_1 = 2$. Process P_2 in turn forwards PRIVILEGE message to process P_4. Process P_2 had requested token on the behalf of process P_4, it updates holder variable as $H_2 = 4$.

Data Structures

- RQ_i is the request queue maintained by process P_i (i = 1, 2 ...). It holds the identities of immediate neighbor processes that have requested for the privilege but have not yet been sent the token. Maximum size of RQ is number of immediate neighbors + 1 (process itself).

- Each process has variable "Asked" which is initially set as false (F). Its value becomes true (T) only if REQUEST is sent either for itself or neighbors and not received. After receiving the PREVILEGE, its value is again set to false.

Algorithm

1. Request for CS

(If token is already held by the process then no need to send REQUEST message. Process needs token for itself or for its neighbor. If variable Asked is false, means it has not already sent a request message).

If process P_i does not hold the token and RQ_i is not empty and Asked = F

Then

$RQi = RQ_i + 1$ and send message to the process held in holder variable H_i;

Asked = T;

Else Asked = F;

2. P_j Receives REQUEST message from P_i.

If P_j holds token ant if it is not executing CS and RQ_j is not empty and at the head of its RQ ID is not "self"

Then

Send PRIVILEGE message to requesting process P_i; update $H_j = i$;

Else if process is unprivileged process

Process forwards the message to parent process.

3. Receipt of PRIVILEGE message

If P_i had requested the token; H_i = "Self"

Then

$RQi = RQ_i - 1$; Enter in CS; Asked = F;

If P_i had forwarded the request on the behalf of its neighbor;

Then

$RQj = RQ_i - 1$; Send the PRIVILEGE message to requesting process which is top of RQ_i; update H_i and change parent

If RQi is still not empty

Then send request message to new H value; Asked = T;

Else Asked = F

4. Execution of CS

Process Pi can enter the CS if it has token and its own ID is at the head of RQ;

5. Process P_i Exiting the CS

If RQi is not empty

Then

Dequeue RQi; If this element is ID of P_n; Send the token to P_n; $H_j = n$; Asked = F;

If RQi is still not empty

Then send REQUEST to parent process; Asked = F.

Performance

Algorithm exchanges 4 messages if system load is heavy. It exchanges only O(log N) messages under light node.

3.6.4 Token Ring Algorithm

- In this algorithm, processes are logically arranged in ring in bus based network. There is no any fix criterion to arrange the processes in ring. IP addresses of the machines can be used or other criterion can be used. Each process should know the next process in line after itself.
- Initially process P_0 holds the token. Token passes from process k to process k+1 in point to point messages. After acquiring token from its neighbor, process checks if it has to enter in CS. If yes, then it enters the CS. After it finishes its execution of CS and exit the CS, it passes the token to its successor along the ring. Same token cannot be used to enter second CS.
- In this way token circulates in ring. If no process is interested to enter in CS then simply token circulates with high speed in ring. As only one process has the token at any instant, a single process will be there in CS at a time.
- It ensures mutual exclusion. There is no starvation. The problem in this algorithm is loss of token. After lost, token needs to be regenerated. If successor of the process is dead to which token is passed then it is removed from ring. Messages required per entry/exit are 0 to infinity. Delay before entry is 0 to (n–1) messages.

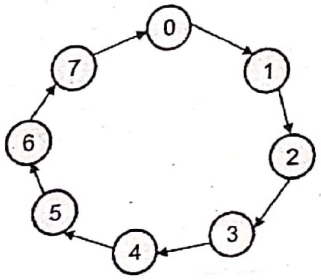


Fig. 3.6.3

3.7 Comparative Performance Analysis of Algorithms

Three parameters are used to compare performance of the algorithms. These are response time, Number of messages required and synchronization delay.

3.7.1 Response Time

- In low load condition of system, response time for many algorithms is round trip time to acquire token *(2T)* plus the time required to execute the CS (which is *E*). In Raymond's Tree-Based algorithm, average distance between requesting process and process holding token is *(log N)/2*. Average round trip delay therefore is *T(log N)*.

- Different algorithms vary in their response time in high load condition of the system. It varies when load in system increases.

3.7.2 Synchronization Delay

- This is delay due exchanges of messages needed after process exit the CS and before next site enters the CS. In most of the algorithms when process exits the CS, it directly sends REPLY or token message to the next process to enter the CS. As a result, the synchronization delay required is T.
- In Maekawa's Algorithm, process exiting the CS unlocks arbiter process by sending RELEASE message. After this, arbiter process sends GRANT message to next process to enter the CS. Hence, synchronization delay required is 2T.
- In Raymond's algorithm, two processes that consecutively execute the CS can be found at any position in tree. Token passes serially along the edges in tree to the process to enter CS next. If tree has N nodes then average distance between two nodes is (log N)/2. The synchronization delay required is T(log N)/2.

3.7.3 Message Traffic

- Lamport's, Ricart-Agrawala, and Suzuki-Kasmi algorithms respectively needs 3(N-1), 2(N-1) and N messages per CS execution in any load condition. In other algorithms, messages required depends on whether light or heavy load present in the system.
- **Light Load :** Raymond's algorithm requires log N messages per CS execution. The request message travel upward towards root node from requester. The token travel backs from root to requester node. If tree has N nodes then average distance between root node and requester node nodes is (log N)/2. Singhal's heuristic algorithm requires N/2 messages per Cs execution.
- **Heavy Load :** Maekawa's Algorithm requires 5√ N messages in high load condition. Raymond's algorithm requires 4 messages whereas, Singhal's heuristic algorithm requires N messages.
- Following table gives comparative performance of different algorithms.

Algorithm	Response Time (Light Load)	Synchronization Delay	Messages (Light Load)	Messages (High Load)
Lamport (Non-Token)	2T+E	T	3(N–1)	3(N–1)
Ricart Agrawala (Non-Token)	2T+E	T	2(N–1)	2(N–1)
Maekawa (Non-Token)	2T+E	2T	3√ N	5√ N
Suzuki and Kasmi (Token)	2T+E	T	N	N
Singhal's Heuristics (Token)	2T+E	T	N/2	N
Raymond (Token)	T(log N)+E	T	Log(N)	4

Review Questions

Q. 1 Why clock synchronization is required in distributed system? Explain.

Q. 2 Explain physical clocks in detail.

Q. 3 Write short note on Global Positioning System (GPS).

Q. 4 Explain Cristian's algorithm for clock synchronization.

Q. 5 Explain Berkeley algorithm for clock synchronization.

Q. 6 Explain Averaging algorithms for clock synchronization.

Q. 7 Explain how Clock Synchronization is carried out in Wireless Networks.

Q. 8 Explain working of Network Time Protocol (NTP).

Q. 9 What is logical clocks? Explain Lamport's logical clock with example.

Q. 10 What is total order multicasting? Explain with example.

Q. 11 What is vector clocks? Explain.

Q. 12 Explain Bully election algorithm with example.

Q. 13 Explain Ring election algorithm with example.

Q. 14 How election of coordinator is carried out in wireless networks? Explain.

Q. 15 What is mutual exclusion? How it is different in distributed system?

Q. 16 Explain in short working ofNon-token based and token-based algorithms for mutual exclusion.

Q. 17 What are the requirements of mutual exclusion algorithms.

Q. 18 Explain metrics to measure performance of mutual exclusion algorithms.

Q. 19 Explain Lamport's algorithm for mutual exclusion in distributed system.

Q. 20 Explain Ricart–Agrawala's algorithm for mutual exclusion in distributed system.

Q. 21 Explain centralized algorithm for mutual exclusion in distributed system.

Q. 22 Explain Maekawa's algorithm for mutual exclusion in distributed system.

Q. 23 Explain Suzuki-Kasami's Broadcast algorithm for mutual exclusion in distributed system.

Q. 24 Explain Singhal's Heuristics algorithm for mutual exclusion in distributed system.

Q. 25 Explain Raymond's Tree-Based algorithm for mutual exclusion in distributed system.

Q. 26 Explain performance analysis of different mutual exclusion algorithms.

❑❑❑