CMPSCI 677 Distributed and Operating Systems **Spring 2025**

Lecture 13: March 24

Lecturer: Prashant Shenoy *Scribe:* ***Anindya Das (2025), Adwait Bhope (2024)***

13.1 Overview

This section covers the following topics:

Distributed Snapshots: Distributed Snapshot Algorithm

Leader Election: Bully Algorithm, Ring Algorithm

Distributed Locks: Centralized, Decentralized, Distributed algorithms

13.1.1 Distributed Snapshot

A simple technique to capture is to assume a global synchronized clock and freeze all machines at the same time and capture whatever that are in all memories at the exactly same time, and capture everything that is in transit. But there is no global clock synchronization. Distributed snapshotting is a mechanism that gives a consistent global state without a global clock synchronization. This is done so that in case one or more components of the distributed system fail, the system as a whole can restart from a snapshot of all the components rather than starting from scratch, as that would waste all the CPU cycles that have already been consumed in the processing before the system failed.

13.1.2 Distributed Snapshot Algorithm

Assume each process communicates with another process using undirectional point-to-point channels (e.g., TCP connections). Any process can initiate the snapshot algorithm. When a process initiates the algorithm, it will first checkpoint its local state, and then send a marker on every outgoing channel. When a process receives a marker, it will have different actions depending on whether it has already seen the marker for this snapshot. If the process sees a marker at the first time, it will checkpoint its state, send markers out and start saving messages on all its other channels. If the process sees a marker for the snapshot at the second time, it will stop saving messages for the corresponding channel from which the marker comes. A process will finish a snapshot until it receives a marker on each incoming channel and processes them all.

For example, in Figure 13.1, process 1, 2 and 3 communicates with each other through the duplex TCP connections. When process 1 initiate a snapshot, it will first save its memory contents (state) into the disk, and then send a marker (in blue) to 2 and 3. It will also start to save the incoming messages(in red) (TPC buffers) from 2 and 3. When 2 receives a marker from 1, it will start to checkpoint state, send a marker out to 1 and 3, and start saving messages from 3. Assume 3 sees the marker sent from 1 first, it will checkpoint its state, sends out a marker to 1 and 2, and start saving messages from 2. 1 will stop saving messages for 2 or 3 until a marker from 2 or 3 arrives. 2 will stop saving messages for 3 until a marker from 2 arrives. And P3 will stop saving messages for 2 until a marker from 3 arrives. Each process will finish this snapshot when it sees a marker from every incoming channel. Thus, a distributed snapshot captured.

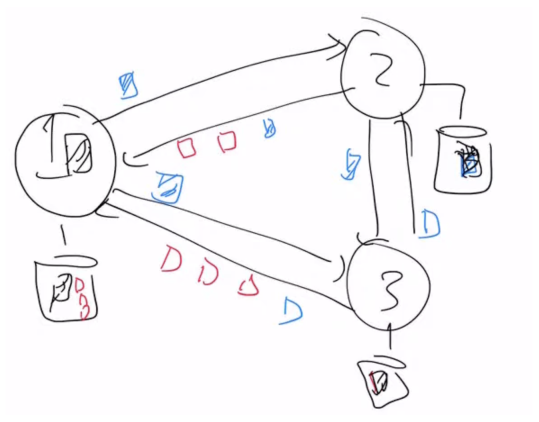


Figure 13.1: Distributed Snapshot Example

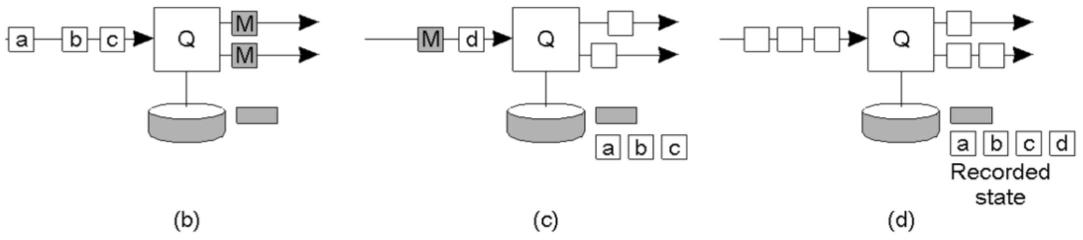


Figure 13.2: Snapshot algorithm example.

13.1.3 Snapshot Algorithm Example

Q decides to take a snapshot. Consider Figure 13.2. b) It receives a marker for the first time and records its local state; c) Q records all incoming message; d) Q receives a marker for its incoming channel and finishes recording the state of the incoming channel. First incoming marker says start recording and all other incoming messages say stop recording.

Question: Why is it required to save messages on the incoming channels?
Answer: The process will save all the messages that were in transit and replay them when the system restarts as if those messages have just been received. If these messages are not saved, the information in them will get lost when the system restarts.

Question: After a process has saved its state but has not received the second marker, can it continue sending messages?
Answer: Yes, the process continues execution normally. The process of taking snapshots does not interfere with the regular execution of the system. Snapshotting is only there to determine at a certain point of time, which states and messages to store in the disk.

Question: Are there different types of markers, such as a START marker and a STOP marker?

Answer: There are no differences in the markers sent between processes. However, in terms of interpretation, the first marker received on any channel is both a START and a STOP marker for the channel it comes from and all other markers received across all channels are STOP markers for corresponding channels.

Question: What if there are some messages in the channel before the marker?
Answer: The messages before the marker are not part of the checkpoint.

Question: How are messages replayed?
Answer: All messages in transit between receiving the first and second marker get stored on disk. And when the process is reinitialized, the corresponding socket buffers are populated with the messages that are part of the snapshot.

Question: What happens if another checkpoint starts when a checkpoint is already going on?
Answer: You can have as many checkpoints going on simultaneously as you want. However, each checkpoint will have a unique Id associated with it so that all artifacts belonging to that checkpoint can be distinguished. All marker messages exchanged between processes are required to contain the checkpoint Id along with them.

Question: How to ensure that the ID of a checkpoint is unique?
Answer: The initiator process can create ID by using its IP address, or process ID or any other information it likes. The ID is not necessarily a number, it can be string as well so there are multiple options to choose unique snapshot IDs.

13.2 Termination Detection

This involves detecting the end of a distributed computation. We need to detect this since no process can just exit after completing its tasks if there is atleast one process that is still running. Let sender be the predecessor, and receiver be the successor. There are two types of markers: Done and Continue. After finishing its part of the snapshot, process Q sends a Done or a Continue to its predecessor. Q can send a Done only when:

- All of Q's successors(and children processes) send a Done
- Q has not received any message since it check-pointed its local state and received a marker on all incoming channels
- Else send a Continue

Computation has terminated if the initiator receives Done messages from everyone.

Question: Isn't it inefficient for all the processes to keep running even though only a subset of them have not finished?
Answer: It is necessary for all remaining processes to be alive as well because the unfinished process(es) might send them a message to obtain a result without which they cannot terminate. Since we don't know beforehand what is remaining to be done, we cannot exit other processes in a distributed system.

Question: What is the benefit of termination detection over distributed snapshot?

Answer: This is just extending the distributed snapshot algorithm. It's not an alternative. So you're still taking snapshots regularly, but as part of taking the snapshots, you're also querying all the processes in the network, whether they have finished their work or are still computing something. So it's just a querying through the network. It's asking whether the computation has finished. If everybody says they finished, then the processes can quit because the processes are independent, so they need to figure out whether they need to keep running or finish. So this allows them to figure that out.

13.3 Leader Election

Many tasks in distributed systems require one of the processes to act as the *coordinator*. Election algorithms are techniques for a distributed system of N processes to elect a coordinator (leader). An example of this is the Berkeley algorithm for clock synchronization, in which the coordinator has to initiate the synchronization and tell the processes their offsets. A coordinator can be chosen amongst all processes through leader election.

13.3.1 Bully Algorithm

The bully algorithm is a simple algorithm, in which we enumerate all the processes running in the system and pick the one with the highest ID as the coordinator. In this algorithm, each process has a unique ID and every process knows the corresponding ID and IP address of every other process. A process initiates an election if it just recovered from failure or if the coordinator failed. Any process in the system can initiate this algorithm for leader election. Thus, we can have concurrent ongoing elections. There are three types of messages for this algorithm: *election*, *OK* and *I won*. The algorithm is as follows:

1. A process with ID i initiates the election.
2. It sends *election* messages to all process with ID $> i$.
3. Any process upon receiving the election message returns an OK to its predecessor and starts an election of its own by sending *election* to higher ID processes.
4. If it receives no OK messages, it knows it is the highest ID process in the system. It thus sends *I won* messages to all other processes.
5. If it received OK messages, it knows it is no longer in contention and simply drops out and waits for an *I won* message from some other process.
6. Any process that receives *I won* message treats the sender of that message as coordinator.

An example of Bully algorithm is given in Figure 13.3. Communication is assumed to be reliable during leader election. If the communication is unreliable, it may happen that the elected coordinator goes down after it is elected, or a higher ID node comes up after the election process. In the former case, any node might start an election process after gauging that the coordinator isn't responding. In the latter case, the higher ID process asks its neighbors who is the coordinator. It can then either accept the current coordinator as its own coordinator and continue, or it can start a new election (in which case it will probably be elected as the new coordinator). This algorithm runs in $O(n^2)$ time in the worst case when the lowest ID process initiates the election. The name bully is given to the algorithm because the higher ID processes are bullying the lower ID processes to drop out of the election.

Question: Can multiple processes elect their own leaders? How do you decide who is the leader finally?
Answer: There are no subsets or groups that we are dealing with here. All processes are in the same group and they have a consensus that the process with the highest ID will be the leader. If there are multiple

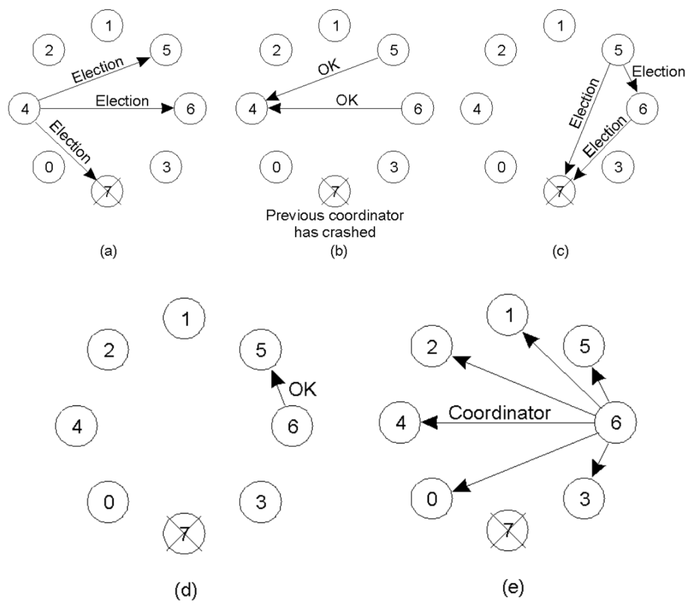


Figure 13.3: Depiction of Bully Algorithm

groups, we are only worried about the leader in that group.

Question: Can the leader be a single machine?
Answer: We don't actually care about the number of machines in the system. Here, it is a process that is chosen as the leader, regardless of the number of machines and the number of processes running on each machine.

Question: Are we assuming here that every node knows about every other node in the system and that they can communicate with each other without a middle-man?
Answer: Yes, here we assume that all processes are aware of the presence of all other processes and can communicate with each other.

Question: Does the chosen leader also need to know about the presence of all other processes in the system?
Answer: That depends on the type of work that is expected out of the leader. For instance, if it is a time-server, then it only needs to answer time queries and not anything about other processes but if the leader's job is distributed lock synchronization, then it needs to know the state and other information of all the other processes. Responsibilities of the leader are not a part of the leader election mechanism.

Question: How do you tell whether a process (leader in this case) has really failed/stopped or is just slow?
Answer: There is no way to differentiate between a slow and a stopped process as such. So all these algorithms work based on time-outs. The process is declared to be dead if it is slow enough to respond after the time-out. Essentially, a failed/stopped process is equivalent to an infinitely slow process. However, when the old leader (slow process) discovers that another leader has been chosen and their process ID is smaller, the old leader can reinitiate the leader election process.

Question: From the figure above, if node 4 knows which processes are alive, why is 4 probing all higher ID

processes rather than sending a message only to 6 and waiting?
Answer: A process does not know which of the higher-ID nodes are actually alive. It may only have heard from the coordinator, and it must probe all higher-ID processes. This ensures that it can accurately determine which nodes are active, especially since some of those nodes (like process 7 in the example) might have crashed.

Question: How are the IDs assigned in this case?
Answer: In this example, the IDs are assigned randomly (or simply numbered from 0 through n). Although other policies could be used (for example, assigning higher IDs to nodes with more resources), the algorithm itself assumes that the nodes already have their IDs assigned.

13.3.2 Ring-based Election Algorithm

The ring algorithm is similar to the bully algorithm in the sense that we assume the processes are already ranked through some metric from 1 to n. However, here a process i only needs to know the IP addresses of its two neighbors ($i + 1$ and $i - 1$). We want to select the node with the highest ID. The algorithm works as follows:

- Any node can start circulating the election message. Say process i does so. We can choose to go clockwise or counter-clockwise on the ring. Say we choose clockwise where $i + 1$ occurs after i.
- Process i then sends an election message to process $i + 1$.
- Anytime a process $j \neq i$ receives an election message, it piggybacks its own ID (thus declaring that it is not down) before calling the election message on its successor (j+1).
- Once the message circulates through the ring and comes back to the initiator i, process i knows the list of all nodes that are alive. It simply scans the list and chooses the highest ID.
- It lets all other nodes know about the new coordinator.

Note that a process might have to jump over its neighbor and contact the neighbor's neighbor in case the neighbor has failed, otherwise, the circulation of the election message across the ring will never finish.

An example of Ring algorithm is given in Figure 13.4. If the neighbor of a process is down, it sequentially polls each successor (neighbor of neighbor) until it finds a live node. For example, in the figure, when 7 is down, 6 passes the election message to 0. Another thing to note is that this requires us to enforce a logical ring topology on the underlying application, i.e. we need to construct a ring topology on top of the whole system just for leader election.

Question: What initiates the election?
Answer: Any process can initiate the election. Say the leader L was a time-server and a process X discovers that X is not responding anymore, then X can initiate the election process.

Question: Is the ring topology applied only for elections?
Answer: Yes, as this is a distributed system, communication between any pair of processes can take place. But for the purpose of leader election, a process of ID i will only communicate with the processes having id $i-1$ and $i+1$, i.e. as a ring topology because this brings the complexity of the leader election down to $O(n)$.

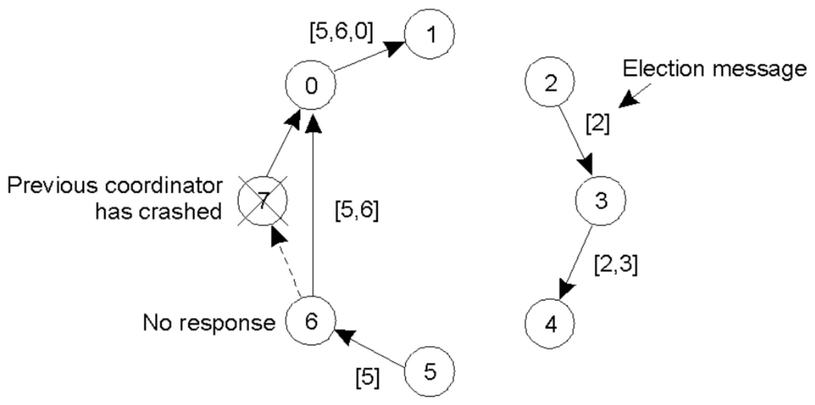


Figure 13.4: Depiction of Ring Algorithm

Question: How many members of the ring does a process need to know?
Answer: In the worst case, a process will have to communicate with all members of the system during the election process, i.e. in a scenario when most of the processes have failed and stopped responding, the remaining processes will have to jump over multiple processes in between in order to elect the next leader.

Question: If every node knows the process ID of all other nodes in the system, why do we need to perform leader election?
Answer: This is because the process starting the election does not know the highest ID process that is alive at that point in time.

Question: If two elections are going on simultaneously, what happens when the first one concludes?
Answer: The second election continues until its completion.

Question: Does election have an ID?
Answer: Yes, so as to facilitate multiple elections occurring simultaneously.

Question: What if the leader crashes even before it is announced?
Answer: You will still announce the chosen process as the leader and it will continue to remain the leader until some process notices that it is no longer alive and initiates another election algorithm.

Question: What if a node with a higher process ID comes up in the middle of an election?
Answer: Leader Election results are stable only if the nodes are stable throughout the election process. If a node suddenly comes up, it may or may not be elected as the leader, but eventually it will be elected when there is another election.

Question: What if the ring is so large that it's not feasible to get a stable result?
Answer: That will be a problem as there is a higher chance of nodes going down or coming up. Some sort of a hierarchy can be introduced in this situation such that there are smaller stable groups and they can elect their own leaders leading to decentralization of the system. However, since this is a P2P system, the stability problem in large rings will remain.

Question: How do you announce the leader? Is it a message broadcast?
Answer: The leader information is circulated along the ring itself by the initiator of the election. It is not a one-to-all broadcast.

Question: How does node 6 know how to send a message to node 0?
Answer: In a ring topology, each node only needs to know its left and right neighbors. However, failures can occur in the network, causing the ring to break and effectively become a chain. When a node fails, a ring restoration process is required to reconnect the two nodes that were separated by the failure. For example, if node 7 fails while node 6 was communicating with it, node 6 might not know its new neighbor immediately. By sending a message in the backward direction, node 6 can determine that node 0 was node 7's neighbor and then re-establish the connection. This healing protocol is essential to ensure that the ring can continue operating despite node failures. For the election algorithm, it is assumed that the ring has already healed itself.

13.3.3 Time Complexity

- Bully Algorithm
  - $O(n^2)$ in the worst case (this occurs when the node with the lowest ID initiates the election)
  - $O(n-2)$ in the best case (this occurs when the node with the highest ID that is alive initiates the election)
- Ring Algorithm
  - Takes $2(n-1)$ messages to execute. First, $(n-1)$ messages are sent during the election query, and then again $(n-1)$ messages to announce the election results. It is easy to extend the Ring Algorithm for other metrics like load, etc.

Question: Will the ring algorithm take a longer time because you are essentially passing messages sequentially?
Answer: Yes. While the ring-based election algorithm has lower message complexity (linear, since it circulates through the ring), it takes more time because messages are passed sequentially through each node in the ring.

Question: Will the bully algorithm get things done in two time steps because you send a message to all higher-ID processes and then they recursively start an election?
Answer: If all the nodes start the election synchronously, then yes, the election could converge in two time steps. However, even in this case with this fast convergence, the total number of messages exchanged is high (on the order of n–1 elections being initiated recursively), and slight asynchrony in starting times could change the exact timing.

Question: Is there any benefit to having the election message go through the whole network (in the ring algorithm), since once a node like process 6 is reached, it's already known to be the highest?
Answer: Because of the ring topology's restrictions, messages must travel sequentially through every node. Even though process 6 might be the highest, the message still needs to circulate completely through the ring to ensure every active process is informed of the outcome.

Question: How, if both process 7 and process 3 are down, does the message go back to process 6?
Answer: In this case, the professor mentioned that the ring must "heal" itself. This healing is typically

achieved by having nodes send messages in a backward direction or by maintaining extra neighbor information so that if a direct neighbor fails, a node can determine the next available neighbor. This extra protocol ensures that the ring connectivity is restored even when multiple nodes have failed.

Question: What happens if there are multiple failures at the same time?
Answer: The multiple simultaneous failures complicate the maintenance of a ring topology. In practice, nodes may keep track of not only their immediate neighbors but also the neighbors of their neighbors (or more) so that the ring can be reconnected even if several nodes fail. The exact recovery mechanism depends on how many failures the system is designed to tolerate.

Question: Is communication bidirectional? For example, can process 0 communicate directly with process 6?
Answer: It depends on the ring's design. In a bidirectional ring, communication can occur in both directions; *however*, in the example discussed above, a simple unidirectional (clockwise) ring is assumed. Thus, process 0 cannot directly communicate with process 6 unless the ring is configured to support bidirectional communication.

z

13.4 Distributed Synchronization

Every time we wish to access a shared data structure or critical section in a distributed system, we need to guard it with a lock. A lock is acquired before the data structure is accessed, and once the transaction is complete, the lock is released. Consider the example below:

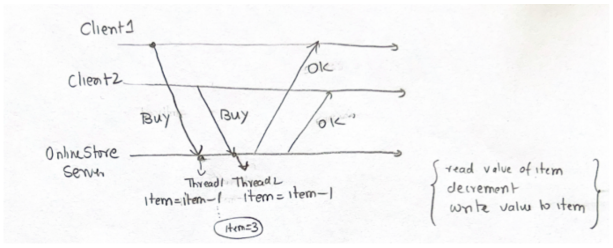


Figure 13.5: Example of a race condition in an online store.

In this example, there are two clients sending a buy request to the Online Store Server. The store implements a thread-pool model. Initially, the item count is 3. The correct item count should be 1 after two buy operations. If locks are not implemented there may be a chance of race condition and the item count can be 2. This is because the decrement is not an atomic operation. Each thread needs to read, update, and write the item value. The second thread might read the value while the first thread is updating the value (it will read 3) and update it to 2 and save it, which is incorrect. This is an example of a trivial race condition.

13.4.1 Centralized Mutual Exclusion

In this case, locking and unlocking coordination are done by a master process. All processes are numbered 1 to n. We run leader election to pick the coordinator. Now, if any process in the system wants to acquire a lock, it has to first send a lock acquire request to the coordinator. Once it sends this request, it blocks execution and awaits reply until it acquires the lock. The coordinator maintains a queue for each data structure of lock requests. Upon receiving such a request, if the queue is empty, it grants the lock and sends the message, otherwise it adds the request to the queue. The requester process upon receiving the lock executes the transaction, and then sends a release message to the coordinator. The coordinator upon receipt of such a message removes the next request from the corresponding queue and grants that process the lock. This algorithm is fair and simple, as shown below in Figure 13.6.

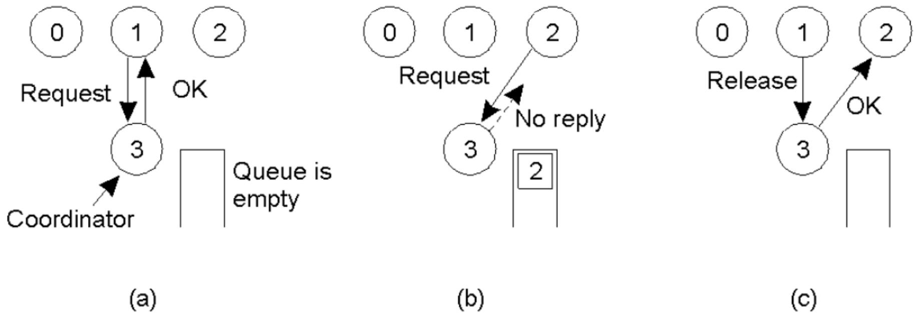


Figure 13.6: Depiction of centralized mutual exclusion algorithm.

Advantages:

- **Fair allocation**: Queue maintains First Come First Served system
- **Simplicity**: Only 3 types of messages are required - *request* (client requests the lock), *grant* (coordinator grants the lock), *release* (client releases the lock)

Issues with handling failure:

- **Coordinator crashes:** When the coordinator process goes down while one of the processes is waiting on a response to a lock request, it leads to inconsistency. The new coordinator that is elected (or reboots) might not know that the earlier process is still waiting for a response. This issue can be tackled by maintaining persistent data on the disk whenever a queue of the coordinator is altered. Even if the process crashes, we can read the file and persist the state of the locks on storage and recover the process.
- **Client process crashes while holding the lock:** This is a bigger problem. In such a case, the coordinator is just waiting for the lock to be released while the other process has gone down. We cannot use a timeout here, because client process transactions can take an arbitrary amount of time to go through. All other processes that are waiting on that lock are also blocked forever. Even if the coordinator somehow knew that the client process crashed, it may not always be advisable to forcibly withdraw the lock because the client process may eventually reboot and think it has the lock and continue its transaction. This causes inconsistency. This is a thorny problem which does not have any neat solution. This limits the practicality of such a centralized algorithm.

Question: Can you check whether the process holding the lock is alive rather than forcing it to release the lock?
Answer: Yes, this is a feasible solution. When the lock holding time expires, the coordinator can just ask

the process whether it wants to renew the lock for another time slice or give the lock up. You can grant locks in intervals of time and have the provision to extend that time so long as the process is alive and willing to hold the lock.

Question: What about consistency when the process holding the lock crashes?
Answer: This is an application-level problem. It needs to be handled by the application logic such that all transactions happening within the synchronized block adhere to the desired ACID properties.

Question: If a client process makes a request and does not get a reply from the coordinator, will it assume that the coordinator has not replied or that the reply was lost?
Answer: We assume that network transmissions are handled by TCP which takes care of lost packets and other transmission-related issues. Hence, the client process assumes that the coordinator has not replied yet.

Question: What if the coordinator crashes *while* writing to disk resulting in a mismatch in the data on disk and the process that actually holds the lock?
Answer: Essentially, we need atomicity for the two operations: granting the lock and writing this action to disk. We will cover this topic in greater detail in the Distributed Transactions section of this course.

13.4.2 Decentralized Algorithm

Decentralized algorithms use voting to figure out which lock request should be granted. In this scenario, each process has an extra thread called the coordinator thread which deals with all the incoming locking requests. Essentially, every process keeps track of who has the lock, and for a new process to acquire a new lock, it has to be granted an *OK* or go-ahead vote from the strict majority of the processes. Here, the majority means more than half the total number of nodes (live or not) in the system. Thus, if any process wishes to acquire a lock, it requests it from all other processes and if the majority of them tell it to acquire the lock, it goes ahead and does so. The majority guarantees that a lock is not granted twice. Upon the receipt of the vote, the other processes are also told that a lock has been acquired and thus, the processes hold up any other lock request. Once a process is done with the transaction, it broadcasts to every other process that it has released the lock.

This solves the problem of coordinator failure because if some nodes go down, we can deal with it so long as the majority agrees that whether the lock is in use or not. Client crashes are still a problem here.

Question: If the minority of processes say NO to a lock request, will there be inconsistency?
Answer: Since we are taking a majority vote here, we will assume that the majority of the processes have the correct state of the lock. Normally, there wouldn't be any inconsistency issues. Only inconsistency comes when a process crashes and is not aware of the state of the lock after it restarts. Such a process will tend to give bad replies and this issue is overcome by relying on the majority to give the right replies.

Question: If we want to overcome getting bad replies from processes, can we use techniques like process start time?
Answer: Strictly speaking, the system can force a process to first recover its lock state fully before it can start responding to lock vote requests. However, this is not used in the method currently because we are anyway robust to the bad replies as long as they are in minority.

Question: Is it possible that nobody gets the majority votes?

Answer: If more than half the processes of the system crash, then this will happen as although the minority of processes that are alive have the correct state of the lock but the majority is sending garbage back during voting.

Question: Is there an associated *release* message for every *request* message?
Answer: Yes, once a process is done executing its critical section, it sends a *release* request to all other processes so that they can update their lock state and grant the lock to the next process in the request queue.

Question: In the decentralized algorithm, what are we voting on?
Answer: The vote is simply a *yes* or *no*. Every process responds with *yes* or *yes* based on whether it wants to allow the asking process to acquire the lock or not.

Question: If a process is not using the lock, will it say *yes*?
Answer: In this technique, every process is running the lock manager code as well. So they are aware of the state of the lock and they will grant the lock based on that state.

13.4.3 Distributed Algorithm

This algorithm, developed by Ricart and Agrawala, needs $2(n-1)$ messages and is based on Lamport's clock and total ordering of events to decide on granting locks. After the clocks are synchronized, the process that asked for the lock first gets it. The initiator sends request messages to all $n-1$ processes stamped with its ID and the timestamp of its request. It then waits for replies from *all* other processes.

When any process receives such a request, it either sends a *grant* message (if there are no other requests) or does not reply if it's executing the critical section itself. If there are multiple pending lock requests (including its own, if it needs the lock), it compares the timestamp of the *request* messages and sends the *grant* message to the process that asked for it first.

- Process k enters the critical section as follows:
  - Generate new timestamp $TS_k = TS_{k+1}$
  - Send request(k, TS_k) to all other $n-1$ processes
  - Wait until it receives the *grant* message from *all* other processes
  - Enter the critical section
- Upon receiving a request message, process j
  - Sends reply if no contention
  - If already in the critical section: does not reply, queue request
  - If it wants to enter itself, compare TS_j with TS_k and send *grant* message if $TS_k < TS_j$, else queue (recall: total ordering based on multicast)

This approach is fully decentralized but there are n points of **failure**, which is worse than the centralized one.

Question: How does it deal with failures? In case some process p_j crashes, the requesting process p_i will keep on waiting for the *grant* message from p_j forever without knowing that it has crashed.

Answer: It is a big problem in this system, there are n points of failure. This is one of those cases where a distributed or decentralized system has worse failure resiliency compared to a centralized system.

13.4.4 Token Ring Algorithm

In the *token ring algorithm*, the actual topology is not a ring, but for locking purposes, there is a logical ring and processes only talk to neighboring processes. There is a token that circulates through the ring and any process that has the token is considered to be holding the lock at that time. When the process is done with its transaction, it sends the token over to its neighbor in the ring. The neighbor keeps the token only if it needs the lock at that moment, otherwise, it passes it on to further. If any node wants to acquire the lock, it cannot ask for it, but it just needs to wait until the token comes to it.

This algorithm derives from the design of an older networking protocol called the Token Ring, which existed as an alternative to Ethernet. In physical networking, only one node can transmit data at a time. If multiple nodes transmit simultaneously, there is a chance of collision leading to garbled data. Ethernet handles this problem by detecting collisions and retransmitting with a back-off policy. In Token Ring, this was handled using locks. Only one machine on the network has the lock at any moment and it transmits at that particular time only.

One problem in this algorithm is the loss of the token. If a process currently holding the token crashes, the token is lost. In the networking algorithm described above, any coordinator node can regenerate the token after a given timeout, however, it is non-trivial to do so here. This is because we cannot put a timeout on the execution time of the critical section, otherwise, it might lead to a scenario where two processes think that they have the token.

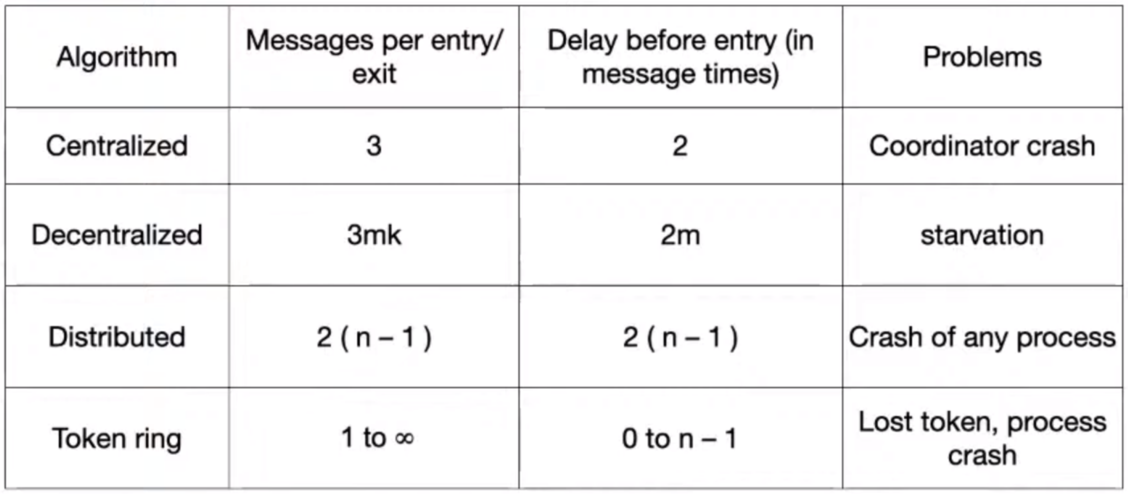
Question: Why is a token necessary?
Answer: In the networking analogy, it is ok to use a simple round-robin strategy that allocates a particular time interval (let's say a few milliseconds) to a node to transmit its messages. If it doesn't complete transmission in that interval, it will have to wait for the next interval. However, in the case of distributed locks, if a process is given the lock but doesn't need it, there is no need to wait for those few milliseconds for its turn to be over. If it does so, it will greatly reduce overall performance. If we use the token, the process can simply pass the token to the next node immediately.

Question: Can we add additional functionality for e.g. we can check whether a process is alive and holds the token and is still executing the critical section or it has crashed requiring the token to be regenerated?
Answer: Yes, these enhancements will work but they are not present in the Token Ring algorithm of the networking world. These features need to be added to the algorithm.

Distributed Locking is a hard problem to solve. As shown in Figure 13.7, there are advantages and disadvantages to all three of these distributed algorithms.

13.4.5 Chubby Lock Service

This is a distributed lock service developed by Google for internal use. It is designed for coarse-grained locking and uses file system abstraction for locks. Each Chubby cell (a set of 5 machines stores the state of the locks and provides service for up to $10,000$ machines. If you need a lock, you can go to Chubby to get a lock and then use it in your application. It may appear to be a centralized algorithm but there is a set of 5 machines that manages the locks.



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

Figure 13.7: Comparison of Distributed Locking Algorithms.

It uses basic file system locking (read-write locks) to grant locks to the clients that are requesting them. E.g. if we create a lock named foo, it will create a file named foo and launch a thread that tries to acquire a lock on that file. If it succeeds, the lock is forwarded back to the client process that requested it.

Question: What do modern databases use to deal with these issues?
Answer: Most modern databases are actually single-machine systems, so they can implement locking in a straightforward, centralized way. In cases where databases are distributed, they typically handle these concerns at a higher level using transactions rather than low-level locking mechanisms. This higher-level abstraction manages mutual exclusion without the need to reimplement distributed lock protocols.