CMPSCI 677 Operating Systems **Spring 2022**

Lecture 14: March 21

Lecturer: Prashant Shenoy *Scribe:* ***Y. Vayunandhan Reddy***

14.1 Overview

This section covers the following topics:

Leader Election: Bully Algorithm, Ring Algorithm, Elections in Wireless Networks

Distributed Synchronization: Centralized, Decentralized, Distributed algorithms

Chubby Lock Service

14.2 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.

14.2.1 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.

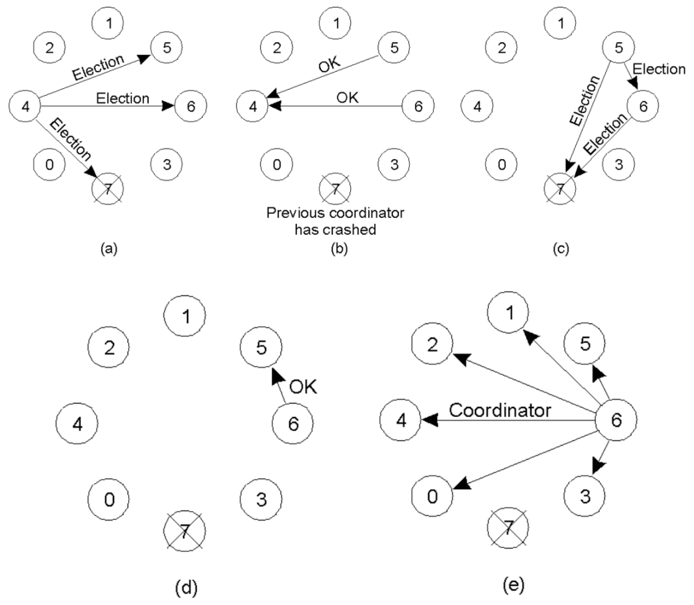

Figure 14.1: Depiction of Bully Algorithm

An example of Bully algorithm is given in Figure 14.1. 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 being 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 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: What happens if 7 has not crashed? Who will it send message to?
Answer: Suppose if 7 has not crashed in the example, it would have sent the response when 4 has started the election. 4 would have dropped out and the recursion would have continued and 7 would have elected as leader finally.

Question: Can 7 never initiate the election?
Answer: If 7 is already a leader there is no reason for it to initiate an election.

Question: When does a smaller ID process know it should start an election?
Answer: This is not particularly specified by this algorithm. Ideally this is done when it has not heard from the coordinator in a while (timeout period).

Question: In the above example what happens if 7 is recovered?
Answer: Any process that is recovered will initiate an election. It will see 6 is the coordinator. In this case 7 will initiate an election and will win.

Question: In the example, how will 7 see 6 is the coordinator (How does a process know who the coordinator is)?
Answer: Discovering who is the coordinator is not part of the algorithm. This should be implemented separately (storing it somewhere, broadcasting the message).

Question: What happens when we have a highly dynamic system where processes regularly leave and join (P2P system)?
Answer: The bully algorithm is not adequate for all kinds of scenarios. If you have a dynamic system, you might want to take into account the more stable processes (or other metrics) and give them higher ids to have them win elections.

14.2.2 Ring 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 about the new coordinator.

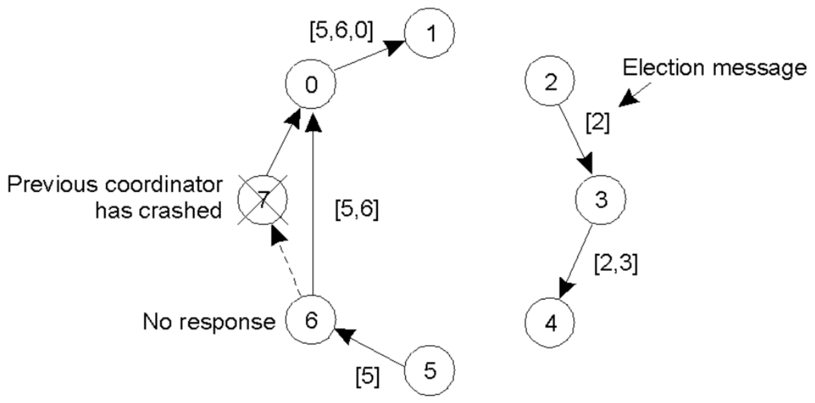

Figure 14.2: Depiction of Ring Algorithm

An example of Ring algorithm is given in Figure 14.2. 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: Does every process need to know about the network topology?
Answer: Every process know the IDs and IP addresses of other processes (assumption). There is no real topology, from the table we can find the neighbours.

Question: If we already know the IDs of all processes why there is a need to go around the ring?
Answer: There can be different kind of failures, e.g., the process may crash or the network may crash while partitioning the ring. We do no know how many processes have disconnected from the ring. We need to actually query and check what is the issue.

Question: How does 6 know that it has to send message to 0 when 7 is down?
Answer: Here we can assume that every node not only has information of its neighbors, but neighbors' neighbors as well. In general, in a system where we expect a max of k nodes to go down during the time leader election takes to execute, each node needs to know at least k successive neighbors in either direction. This is still less than what each node needs to know in Bully algorithm.

14.2.3 Time Complexity

The bully algorithm runs in

- $O(n^2)$ in the worst case (this occurs when the node with lowest ID initiates the election)
- $O(n-2)$ in the best case (this occurs when the node with highest ID that is alive initiates the election)

The ring algorithm always takes 2(n-1) messages to execute. The first (n-1) is during the election query and second time to announce the results of the election. It is easy to extend ring algorithm for other metrics like load, etc.

Question: How do you know a node is not responding?
Answer: If it has actually crashed then TCP will fail while setting up socket connection. Otherwise, it can be a slow machine which is taking time. It is a classical problem in distributed systems to distinguish between a slow process and a failed process which is a non-trivial problem. Timeout is not an ideal solution but can be used in practice.

14.3 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 has completed, the lock is released. Consider the example below:

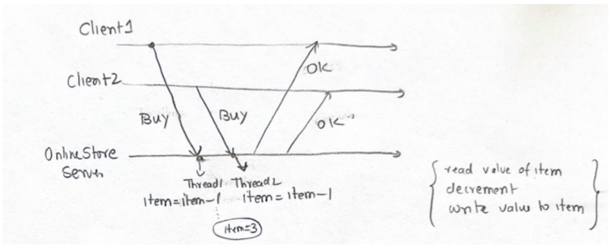

Figure 14.3: Example of a race condition in an online store.

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 chance of race condition and 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 first thread is updating the value (it will read 3) ans update it to 2 and save it, which is incorrect. This is an example of trivial race condition.

14.3.1 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.

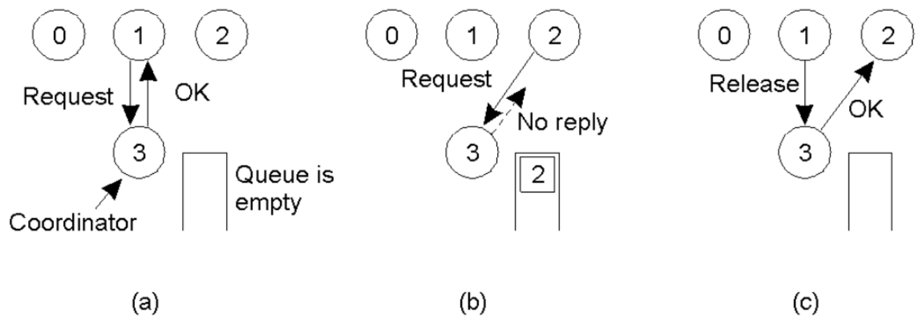

Figure 14.4: Depiction of centralized mutual exclusion algorithm.

An example of the algorithm is given in Figure 14.4. There are two major issues with this algorithm, related to failures. When 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 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.

The harder problem occurs when one of the client process crashes while it is holding the lock (during one of its transactions). In such a case, coordinator is just waiting for the lock to be released while the other process has gone down. We cannot use timeout in this case, because usually transactions take 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 has crashed, it may not always be advisable to take the lock forcibly back 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 an centralized algorithm.

14.3.2 Decentralized Algorithm

Decentralized algorithms use voting to figure out which lock requests to grant. 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 a 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, 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.

14.3.3 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.

Any other process upon receiving such a request either sends reply if it does not want the lock for itself, or is already in the transaction phase (in which case it doesn't send any reply and the initiator has to wait), or it itself wants to acquire the same lock in which case it compares its own request timestamp with that of the incoming request. The one with the lower timestamp gets the lock first.

- Process k enters critical section as follows:
  - Generate new time stamp $TS_k = TS_{k+1}$
  - Send request(k,TS_k) all other n-1 processes
  - Wait until reply(j) received from all other processes
  - Enter critical section
- Upon receiving a request message, process j
  - Sends reply if no contention
  - If already in critical section, does not reply, queue request
  - If wants to enter, compare TS_j with TS_k and send reply 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.

14.3.4 Token Ring Algorithm

In the *token ring algorithm*, the actual topology is not a ring, but for locking purpose there is a logical ring and processes only talk to neighboring processes. A process with the token has the lock at that time. To acquire the lock one needs to wait. The token is circulated through the logical ring. No method is there to request the token, the process needs to wait to get a lock. Once the process has token it can enter the critical section.

This was designed as part of a networking protocol Token Ring. In physical networking, only one node can transmit at a time. If multiple nodes transmit at a time there is a chance of collision. Ethernet handled it by detecting collisions. A node transmits and if there is a collision it backs off and will succeed eventually. In Token Ring this was handled using locks. Only one machine has lock at an instance in the network and it transmit at that particular time.

In this algorithm one problem is loss of token. Regenerating the token is non-trivial, as you can not use timeout strategy.

Question: In a Token Ring, when should you hold the token?
Answer: If a process want to send data on the network it will wait until it receives the token. It will hold the token until it completes the transmission and pass the token to the next. If it do not require the token it simply passes it.

Question: Is the token generated by the process?
Answer: A token is special message that is circulating in the network. It is generated when the system started. A process can hold the message or pass it to the next.

Question: In Token Ring, if the time limit is finished and the process is still in the critical section, what happens?
Answer: In general network transmission this can limit the amount of data you can transmit. But if this is used for locking the message can be held for arbitary amount of time (based on the critical section). This may complicate the token recovery because we can't distinguish if a token is lost or any process is in long critical section.

14.4 Chubby Lock Service

This was developed by Google to provide a service that can manage lots of locks for different subgroups of applications. Each Chubby cell has group of 5 machines supporting 10,000 servers (managing the locks of applications running on them). This was designed for coarse-grain locking with high reliability. One of the 5 machines is maintained outside the data center for recovery.

Chubby uses distributed lock to elect a leader. The process with the lock is elected as leader. The 5 processes in the cell run an internal leader election to elect a primary. The other are lock workers. These are used for replication. The applications in the system ask for locks and release locks using RPC calls. All of the locks requests go to the primary. The lock state is kept persisted on disk by all the machines. This uses a file abstraction for locks. To lock and unlock, file is locked and unlocked respectively. State information can also be kept in the file. It supports reader-writer locks. If the primary fails it triggers a new leader election.

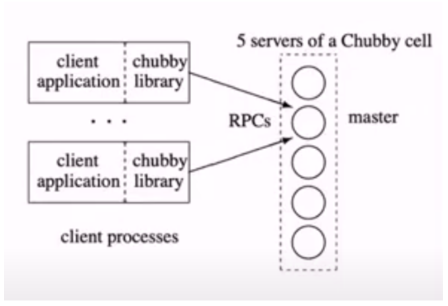

Figure 14.5: Chubby lock service.

Question: What is the purpose of multiple replicas?
Answer: To ensure fault tolerance and lock state doesn't disappear.

Question: Do the replicas have any purpose till the primary fails?
Answer: They store all the lock information in the database. They work as hot standby and can takeover with current state if the primary fails.

Question: Where are the lock files created?
Answer: We have distributed file system which looks same on all the machines.

Question: If Chubby is used for leader election how does it handle coordinator failure?
Answer: It has a notion of lease, every process can hold lease for particular lease period and needs to be renewed. This ensures that the lock is never with a failed process.

Question: While writing is it through all nodes or just through master?
Answer: All client requests are directed to master. The master sends the operations performed to others to keep them in sync.