CMPSCI 677 Operating Systems **Spring 2014**

Lecture 14: March 9

Lecturer: Prashant Shenoy *Scribe: Nikita Mehra*

14.1 Distributed Snapshot Algorithm

A distributed snapshot algorithm captures a consistent global state of a distributed system. In case of a distributed system, there is always a high probability of node failure, in which case the distributed snapshot algorithm can be used by all the processes to load the most recent snapshot.A global state can be described by a cut that indicates the time at which each process "checkpoints" its local state and messages. In the case of a consistent cut C, if a message crosses C, its "send" should be before C and its "receive" should be after C. When the system is recovered from a consistent cut, every message will be sent exactly once. If a message's "send" is after C while its "receive" is before C, C becomes inconsistent. It will cause problems when the processes are restarted from an inconsistent cut. For example, message m_2 can be executed twice.Please refer to the diagram provided in the lecture slides.(Lecture 14 page 3).

The algorithm assumes that:

- There are no network failures
- Each process communicates with another process using unidirectional point-to-point channels
- There is no message reordering on each channel
- There is a communication path between any two processes

Marker messages are essentially the special messages to take a checkpoint. Any process can initiate the algorithm by: 1) checkpointing its local state which contains all necessary information to restart itself, and 2) sending a marker on every outgoing channel. Then for every message from every incoming channel, the process writes a copy to disk, until a subsequent marker is received.

Now every other process may receive markers. If one process receives its first marker, it also checkpoints its local state, sends a marker on every outgoing channel and saves messages from all other incoming channels until a subsequent marker comes.

A process finishes when it receives a marker on each incoming channel. It finally collects states of all channels and send them with its own local state to initiator.

Multiple snapshots may be in progress. Each of them is separate and distinguished by tagging the marker with the initiator ID (and sequence number).

This algorithm will always stop because: 1) every process sends markers and saves incoming messages only when it receives the first marker; 2) every process stops saving messages from an incoming channel when it receives a subsequent marker from that channel, and such a marker always comes.

There is a case that process B, for example, dont have any other incoming channels except the one receiving the marker, it should send the marker through its outgoing channels and then stop.

Intuitively, this algorithm can capture a consistent global state because: 1) no message is recorded/sent twice; 2) no message is lost, since all messages in transit are always between the two markers on the channel and therefore recorded.

14.2 Termination Detection Algorithm

A termination detection algorithm detects the end of a distributed computation when there is no coordinator in the system. The end here means all processes are done and there are no messages(requests) in transit. A process is done when all its local computation is done and there are no incoming messages(requests) from other processes. The sender of messages is referred to as the "predecessor" and the receiver of messages is referred to as a"successor".

The algorithm needs two types of markers: *Done* indicates that a process is done, and *Continue* indicates that a process is still working. It works as follows:

- Any process can initiate the algorithm by broadcasting *Done* to all its neighbors
- Any process that receives the request finishes its part of snapshot and sends the request to all its neighbors except its sender. It returns a *Done* if it receives no message from predecessors, after it checkpoints its local state and all its neighbors(successors) reply *Done*. Otherwise, it returns *Continue*
- The system is done if the initiator receives *Done* from everyone

14.3 Election Algorithms

It is important to pick a unique coordinator in distributed systems. The coordinator plays the role of the master role many applications. It can be any peer of the system as long as it can be uniquely identified. The coordinator cannot be fixed(hard-coded) since peers dynamically join and leave the network and hence the new leader should also be dynamically chosen at run-time. The tasks performed by the coordinator vary from application to application.

There are two typical algorithms: Bully Algorithm and Ring-Based Algorithm. They both assume that each process has a unique numerical ID and the goal is to pick the one with highest ID as the coordinator.

14.3.1 Bully Algorithm

Bully Algorithm assumes that processes know the ID and address of every other process.

There are three types of messages: *Election*, *OK*, and *I Won*. Processes with higher IDs have high priority to win.

A process can initiate the algorithm when it is just recovered from failure or its coordinator failed. It broadcasts *Election* to all processes with higher IDs and awaits *OK* (there is no need to ask the processes with lower IDs). If it receives no *OK*, it becomes the coordinator and sends *I Won* to all processes with lower IDs. If it receives *OK*, it drops out and awaits *I Won* message from a process with higher ID.

Any process that receives *Election* returns *OK* and starts a new election. Any process that receives *I Won* treats the sender as the coordinator.

There can be multiple elections at the same time.

One question is how can a sender know if a receiver has already failed so it needs not to await the reply. Luckily in some cases, the sender knows the receiver's failure immediately because it cannot setup a connection. In other cases, a timeout mechanism may work. A receiver is considered failed if it does not respond after a while. Even if the receiver is just slow and responds after timeout period expired, it can initiate a new election if it has a higher ID than current coordinator's.

Its a simple algorithm that only depends on the IDs. But in some other cases, we could change to consider the current work load factor, network connection factor as the main criteria to decide on the leader.

14.3.2 Ring-Based Algorithm

Ring-Based Algorithm assumes that processes are arranged in a logical ring and each one knows its two neighbors' IDs and addresses.

Similarly, a process can initiate the algorithm when it is recovered or its coordinator failed. It sends *Election* to its closest alive downstream node. Then every process tags its ID on the message. Finally the message comes back to the initiator. It picks the process with highest ID and sends a coordinator message.

There can also be multiple elections in progress.

Suppose there are n processes and one election in progress: Ring-Based Algorithm always needs $2(n-1)$ messages, while Bully Algorithm needs $(n-2)$ messages in the best case, and $O(n^2)$ messages in the worst case.

14.3.3 Election in Large-Scale Systems

In a real large-scale distributed system, the requirement for a leader may be different from the ideal case above. For example, instead of picking the process with the highest ID, people may want the one with the best network/CPU performance. Performance information can be attached as a modification of the above algorithms. Sometimes there can even be multiple leaders(superpeers) in the system. Requirements for superpeers selection are:

- Normal nodes should have low-latency access to superpeers
- Superpeers should be evenly distributed across the overlay network
- There should be a predefined portion of superpeers relative to the total number of nodes in the overlay network
- Each superpeer should not serve more than a fixed number of normal nodes

14.4 Distributed Synchronization

When multiple processes in a distributed system access some shared data or data structure, people can use critical sections with mutual exclusion. This problem can be solved in a single process with multiple threads by using semaphores, locks or monitors. In a distributed system, a lock mechanism is required.

14.4.1 Centralized Algorithm

Assume processes are numbered and the one with highest ID is elected coordinator. 1) Every process needs to send a *request* to the coordinator and await *grant* before entering the critical section. When it is finished, it sends a *release* to the coordinator. 2) When the coordinator receives a *request*, it sends a *grant* if the lock is available; Otherwise, it puts the process to a queue. When the coordinator receives a *release*, it sends *grant* to the process at the front of the queue.

Although this is a straightforward method, there are many issues. For example, what will the system do when the coordinator crashes. A new coordinator may not know which process is holding the lock. One may suggest to use a log file and let the new coordinator check the log. Then how to maintain a global log in a distributed system becomes a new problem. Another problem may be the case that a process with the lock crashes. The coordinator cannot take the lock back since it cannot tell whether the process has failed or it is just slow.

Advantages:

- Fair: requests are granted the lock in the order they were received (actually other strategies can also be implemented)
- Simple: three messages per use of a critical section (request, grant, release)

Disadvantages:

- The coordinator is a single point of failure
- Hard to detect a dead coordinator
- Performance bottleneck in large distributed systems

14.4.2 Decentralized Algorithm

Decentralized algorithm uses voting. It assumes n replicas of coordinator threads. Any process needs majority vote $m > n/2$ to acquire the lock. This algorithm works even when some coordinators crash.

14.4.3 Distributed Algorithm

This algorithm is based on event ordering and timestamps. It grants the lock to the process which requests first.It can use Lamport's Algorithm(Happened Before Relation) to decide the ordering of two request. But using simple Lamport's Alogrithms is not sufficient since the alogrithm can only decide a partial order. The algorithm requires total ordering of events. By multicasting all the messages to all other processes total ordering of events can be obtained.

When a process k asks for the lock, it:

- Generates a new time stamp $TS_k = TS_k + 1$
- Sends $request(k, TS_k)$ to all other processes
- Waits for $reply(j)$ from all other processes

- Enters critical section

When a process j receives $request(k, TS_k)$, it:

- Sends $reply(j)$ if it is not interested
- Queues $request(k, TS_k)$ if it is in critical section
- If it wants to enter: if $TS_k < TS_j$ sends $reply(j)$, otherwise queues $request(k, TS_k)$