Distributed Synchronization

Chapter 18 examines various mechanisms for process synchronization and communication, as well as methods for dealing with the deadlock problem, in a distributed environment. In addition, since a distributed system may suffer from a variety of failures that are not encountered in a centralized system, we also discuss here the issue of failure in a distributed system.

18.1 The difficulty is that each site must maintain its own local wait-for graph. However, the lack of a cycle in a local graph does not ensure freedom from deadlock. Instead, we can ensure the system is not deadlocked only if the union of **all** local wait-for graphs is acyclic.

18.2 A proof of this can be found in the article *Distributed deadlock detection algorithm*, ACM Transactions on Database, June 1982.

18.3

a. Would you use a deadlock-detection scheme, or a deadlock-prevention scheme?

We would choose deadlock prevention as it is systematically easier to prevent deadlocks than to detect them once they have occurred.

b. If you were to use a deadlock-prevention scheme, which one would you use? Explain your choice.

A simple resource-ordering scheme would be used: preventing deadlocks by requiring processes to acquire resources in order.

c. If you were to use a deadlock-detection scheme, which one would you use? Explain your choice.

If we were to use a deadlock detection algorithm, we would choose a fully distributed approach as the centralized approach provides for a single point of failure.

18.4 The following algorithm requires $O(n \log n)$ messages. Each node operates in phases and performs:

a. If a node is still active, it sends its unique node identifier in both directions.

b. In phase k, the tokens travel a distance of 2^k and return back to their points of origin.

c. A token might not make it back to the originating node if it encounters a node with a lower unique node identifier.

d. A node makes it to the next phase only if it receives its tokens back from the previous round.

The node with the lowest unique node identifier is the only one that will be active after $\log n$ phases.

18.5 The options are a (1) centralized, (2) fully distributed, or (3) token-passing approach. We reject the centralized approach as the centralized coordinator becomes a bottleneck. We also reject the token-passing approach for its difficulty in re-establishing the ring in case of failure.

We choose the fully distributed approach for the following reasons:

a. Mutual exclusion is obtained.

b. Freedom from deadlock is ensured.

c. Freedom from starvation is ensured, since entry to the critical section is scheduled according to the timestamp ordering.

d. The number of messages per critical-section entry is $2 \times (n - 1)$. This number is the minimum number of required messages per critical-section entry when processes act independently and concurrently.

18.6 Assume that each node has the following multicast functionality, which we refer to as *basic multicast* or *b-multicast*. *b-multicast(v)*: node simply iterates through all of the nodes in the system and sends an unicast message to each node in the system containing v.

Also assume that each node performs the following algorithm in a synchronous manner assuming that node i starts with the value v_i.

a. At round 1, the node performs b-multicast(v_i).

b. In each round, the node gathers the values received since the previous round, computes the newly received values, and performs a b-multicast of all the newly received values.

c. After round $f + 1$, if the number of failures is less than f, then each node has received exactly the same set of values from all of the other nodes in the system. In particular, this set of values includes all values from nodes that managed to send its local value to any of the nodes that do not fail.

The above algorithm works only in a synchronous setting and when the message delays are bounded. It also works only when the messages are delivered reliably.

18.7 In the wound–wait scheme an older process never waits for a younger process; it instead rolls back the younger process and preempts its resources. When a younger process requests a resource held by an older

process, it simply waits and there are no rollbacks. In the wait–die scheme, older processes wait for younger processes without any rollbacks but a younger process gets rolled back if it requests a resource held by an older process. This rollback might occur multiple times if the resource is being held by the older process for a long period of time. Repeated rollbacks do not occur in the wound–wait scheme. Therefore, the two schemes perform better under different circumstances depending upon whether older processes are more likely to wait for resources held by younger processes or not.

18.8 Possible failures include (1) failure of a participating site, (2) failure of the coordinator, and (3) failure of the network. We consider each approach in the following:

a. **Failure of a Participating Site**—When a participating site recovers from a failure, it must examine its log to determine the fate of those transactions that were in the midst of execution when the failure occurred. The system will then act accordingly depending upon the type of log entry when the failure occurred.

b. **Failure of the Coordinator**—If the coordinator fails in the midst of the execution of the commit protocol for transaction T, then the participating sites must decide on the fate of T. The participating sites will then determine to either commit or abort T or wait for the recovery of the failed coordinator.

c. **Failure of the Network**—When a link fails, all the messages in the process of being routed through the link do not arrive at their destination intact. From the viewpoint of the sites connected throughout that link, the other sites appears to have failed. Thus, either of the approaches discussed above apply.

18.9 Vector clocks can be used to distinguish concurrent events from events ordered by the happens-before relationship. A vector clock works as follows. Each process maintains a vector timestamp that comprises of a vector of scalar timestamp, where each element reflects the number of events that have occurred in each of the other processes in the system. More formally, a process i maintains a vector timestamp t_i such that t_i^j is equal to the number of events in process j that have occurred before the current event in process i. When a local event occurs in process i, t_i^i is incremented by one to reflect that one more event has occurred in the process. In addition, any time a message is sent from a process to another process it communicates the timestamp vector of the source process to the destination process, which then updates its local timestamp vector to reflect the newly obtained information. More formally, when s sends a message to d, s communicates t_s along with the message and d updates t_d such that for all $i \neq d$, $t_d^i = max(t_s^i, t_d^i)$.