Unit-III

4.1. INTRODUCTION

A distributed system is a network of sites that exchange information with each other by message passing. A site consists of computing and storage facilities and interface to local users and a communication network. In distributed systems, a process can request and release resources (local or remote) in any order, which may not be known a priori and a process can request some resources while holding others. If the sequence of the allocation of resources to processes is not controlled in such environments, deadlocks can occur. In this chapter, we study deadlock handling strategies in distributed systems. Several deadlock detection techniques based on various control organizations are described. Pros and cons of these techniques are discussed and their performance is compared.

System Model

The problem of deadlocks has been, generally, studied in distributed system under the following model :

* The systems have only reusable resources.
* Processes are allowed only exclusive access to resources.
* There is only one copy of each resource.

A process can be in two states : running or blocked. In the running state (also called the active state), a process has all the needed resources and is either executing or is ready for execution. In the blocked state, a process in waiting to acquire some resources.

4.2. RESOURCES VS COMMUNICATION DEADLOCKS

Two types of deadlocks have been discussed in the literature : resource deadlocks and communication deadlocks. In resource deadlocks, processes can simultaneously wait for several resources and cannot proceed until they have acquired all the resources. A set of processes is resource deadlocked if each process in the set requests resources held by another process in the set and it must receive all of the requested resources before it can become unlocked.

In communication deadlocks, processes wait to communicate with other processes among a set of processes. A waiting process can unblock on receiving a communication from any one of these processes. A set of processes is communication deadlocked if each process in the set is waiting to communicate with another process in the set and no process in the set ever initiates any further communication until it receives the communication for which it is waiting.

Wait For Graph (WFG) : In distributed system, the system state can be modelled or represented by directed graph called wait for graph (WFG).

In WFG, nodes are processes and there is a directed edge from node P_1 to node P_2, if P_1 is blocked and is waiting for P_2 to release some resource. A system is deadlocked, if and only if there is directed cycle present in the WFG.

In distributed database, a wait for graph is referred to as transaction wait for (TWF) Graph. In TWF graph, nodes are transaction and there is a directed edge from node T_1 to node

T_2, if T_1 is blocked and is waiting for T_2 to release some resource. A system is deadlocked, if and only if there is directed cycle in TWF graph.

Necessary Conditions : There are four necessary conditions for a deadlock to occur, known as the *Coffman conditions* from their first description in a 1971 article by E. G. Coffman.

1. **Mutual Exclusion Condition :** A resource that cannot be used by more than one process at a time.
2. **Hold and Wait Condition :** Processes already holding resources may request new resources.
3. **No Promption Condition :** No resource can be forcibly removed from a process holding it, resources can be released only by the explicit action of the process.
4. **Circular Wait Condition :** Two or more processes from a circular chain where each process waits for a resource that the next process in the chain holds.

4.3. DEADLOCK HANDLING STRATGIES IN DISTRIBUTED SYSTEM

4.3.1. Deadlock Prevention

Prevention is the name given to schemes that guarantee that deadlock can never happen because of the way the system is structured. One of the four conditions for deadlock is prevented, thus, preventing deadlocks. One way to do this is to make processes declare all the resources that might eventually need, when the process is first started. Or if all the resources are available in the process are allowed to continuem, all of the resources are acquired together, and the process proceeds, releasing all the resources when it is finished. Thus, *hold and wait* cannot occur.

The major disadvantage of this scheme is that resources must be acquired because they might be used, not because they will be used. Also, the pre-allocation requirement reduces potential concurrency. Another prevention scheme is to impose an order on the resources and require processes to request resources in increasing order. This prevents *cyclic wait* and thus, makes deadlocks impossible.

One advantage of prevention is that process aborts are never required due to deadlocks. While most systems can deal with rollbacks, some systems may not be designed to handle them and thus, must use deadlock prevention.

Features of Deadlock Prevention

- Removing the mutual exclusion condition means that no process should have exclusive access to a resource. This proves impossible for resources that cannot be spooled, and even with spooled resources deadlock could still occur. Algorithm that avoid mutual exclusion are called non-blocking synchronization algorithms.
- The "hold and wait" conditions may be removed by requiring processes to request all the resources they will need before starting up (or before embarking upon a particular set of operations). This advance knowledge is difficult to satisfy and, in any case, is

inefficient use of resources. Another way is to require processes to release all their resources before requesting all the resources they will need. This too is often impractical. (Such algorithms, such as serializing tokens are known as the all-or-none algorithms).

- A "no preemption" (lockout) conditon may also be difficult or impossible to avoid as a process has to be able to have a resource for a certain amount of time, or the processing outcome maybe inconsistent or thrashing may occur. However, inability to enforce preemption may interfer with a *priority* algorithm. (Note : Preemption of a "locked out" resource, generally, implies a rollback, and is to be avoided, since it is very costly in overhead.) Algorithm that allows, preemption include lock-free and wait-free algorithms and optimistic concurrency control.
- The circular wait condition : Algorithms that avoid circular waits include "disable interrupts during critical sections", and "use a hierarchy to determine a partial ordering of resources" (where no obvious hierarchy exists, even the memory address of resources has been used to determine ordering) and Dijkstra's solution.

Circular Wait Prevention : Circular wait prevention consists of allowing processes to wait for resources, but ensure that the waiting cannot be circular. One approach might be to assign a precedence to each resource and force processes to request resources in order of increase precedence. That is to say that if a process holds some resources, and the highest precedence of these resources is m, then this process cannot request any resource with precedence smaller than m. This forces resource allocation to follow a particular and non-circular order, so circular wait cannot occur. Another approach is to allow holding only one resource per process; if a process request another resource, it must first free the one it is currently holding (that is disallow hold-and-wait).

4.3.2. Deadlock Avoidance

In deadlock avoidance, the system considers resource requests while the processes are running and takes action to ensure that those requests do not lead to deadlock. Avoidance based on *banker's algorithm*, sometimes used in centralized systems, is considered not practical for a distributed system. Two popular avoidance algorithms based on timestamps or priorities are *wound-wait* and *wait-die*. They depend on the assignment of unique global timestamps or priority to each process when it starts. Some authors refer to these as prevention.

In wound-wait, if process A requests a resource currently held by process B, their timestamps are compared. Process B is wounded and must restart if it has a larger timestamp (is younger) than A. Process A is allowed to wait if process B has the smaller timestamp. Deadlock cycles cannot occur since processes only wait for older processes. In wait-die, if a request from process A conflicts with process B, A will wait if process B has the larger timestamp (is younger). If process B is the older process then A is not allowed to wait, so it dies and restarts.

In timeout based avoidance, a process is blocked when it requests a resource that is not currently available. If it has been blocked longer than a timeout period, it is aborted and

restarted. Given the uncertainty of message delays in distributed ... determine good timeout values.

These avoidance strategies have the disadvantages that the abort... been actually involved in a deadlock.

Deadlock can be avoided, if certain information about processes ... of resource allocation. For every resource request, the system sees, if ... mean that the system will enter an *unsafe* state, meaning a state that could ... The system then only grants requests that will lead to *safe* states. In order for ... able to figure out whether the next state will be safe or unsafe, it must know in ... time the number of types of all resources in existence, available and requested. ... algorithm that is used for deadlock avoidance is the Banker's algorithm, which requires ... usage limit to be known in advance. However, for many system, it is impossible to ... advance what each process will request. This mean that deadlock avoidance is often impr...

Two other algorithms are Wait/Die and Wound/Wait, each of which uses a symmetr... breaking technique. In both these algorithms, there exists an older process (O) and a younger process (Y). Process age can be determined by a time stamp at process creation time. Smaller timestamps are older processes, while larger timestamps represent younger processes.

	Wait/Die	Wound/Die
O needs a resource held by Y	O waits	Y dies
Y needs a resource held by O	Y dies	Y waits

It is important to note that a process may be in an unsafe state but would not result in deadlock. The notion of safe/unsafe states only refers to the ability of the system to enter deadlock state or not. For example, if a process requests A which would result in an unsafe state, but release B which would prevent circular wait, then the state is unsafe but the system is not in deadlock.

4.3.3. Deadlock Detection

Deadlock detection attempts to find and resolve actual deadlocks. These strategies ... on a Wait For Graph (WFG) that is some schemes is explicitly built and analyzed for cycles. ... the WFG, the nodes represent processes and the edges represent the blockages or dependenc... Thus, if process A is waiting for a resources held by process B, then there is an edge in W... from the node for process A to the node for process B.

In the AND model (resource model), a cycle in the graph indicates a deadlock. In the ... model, a cycle may not mean a deadlock since any set of requested resources may unblock ... process. A *knot* in the WFG is needed to declare a deadlock. A knot exists when all nodes ... can be reached from some node in a directed graph can also reach that node.

In a centralized system, a WFG can be constructed fairly easily. The WFG can be chec... for cycles periodically or every time a process is blocked, thus, potentially adding a new e... to the WFG. When a cycle is found, a victim is selected and aborted.

...n neither deadlock avoidance nor deadlock prevention may be used. Instead deadlock ...on and process restart are used by employing an algorithm that tracks resource allocation ...process states, and rolls back and restarts one or more of the processes in order to remove ...deadlock. Detecting a deadlock that has already occurred is easily possible since the ...sources that each process has locked and/or currently requested are known to the resource ...heduler or OS.

Detecting the possibility of a deadlock before it occurs is much more difficult and is, in fact, generally undecidable, because the halting problem can be rephrased as a deadlock scenario. However, in specific environments, using specific means of locking resources, deadlock detection may be decidable. In the general case, it is not possible to distinguish between algorithms that are merely waiting for a very unlikely set of circumstances to occur and algorithms that will never finish because of deadlock.

4.4. CONTROL ORGANIZATION FOR DISTRIBUTED DEADLOCK DETECTION

The following three control organizations are common for deadlock detection in distributed systems :

4.4.1. Centralized Control

In centralized deadlock detection algorithms, a designated site (often called a *control site*) has the [redacted]. The control site may maintain the global WFG constantly or it may build it whenever a deadlock detection is to be carried out by soliciting the local WFG from every site.

Advantages of centralized algorithms :

Detection as well as resolution are conceptually simple and easy to implement since the control site has complete information of the WFG and the deadlock cycle.

Disadvantages

Single point of failure, congestion likely at the control site and at communication links near that site, design not scalable to growth of system.

4.4.2. Distributed Control

In distributed deadlock detection algorithms, the responsibility of detecting a global deadlock is shared equally among all sites. A deadlock detection is initiated only when a waiting process is suspected to be part of a deadlock. When the detection of global deadlock is to be carried out, several sites participate in the detection and the WFG is spread over many sites.

Distributed control is not vulnerable to a single point of failure; no site will become a bottleneck due to deadlock activities, and the design is much more scalable than the centralized organization. However, [redacted]. Additional problems include : several sites may initiate detection of the same deadlock, deadlock resolution is often cumbersome since several sites may detect the same deadlock without being aware of it, and the proof of correctness of these algorithms is difficult.

4.4.3. Hierarchical Control

In hierarchical deadlock detection algorithms, sites are arranged in a hierarchical fas[hion] and [redacted]. If the hierarchy struc[ture] coincides with resource access pattern local to clusters of sites, this approach can prov[ide] efficient detection of deadlocks and rip the benefits of both the centralized and the distribut[ed] control organizations. On the otherhand, if deadlocks are not mostly localized to a few cluster[s] but often span several clusters, this approach will be inefficient.

4.5. ALGORITHMS FOR DISTRIBUTED DEADLOCK DETECTION

There are three types of algorithms or methods for deadlock detection :

1. Centralized Deadlock Detection
2. Distributed Deadlock Detection
3. Hierarchical Deadlock Detection

P_0, P_1, P_2 ✓
R, S, T

4.5.1. Centralized Deadlock Detection

In this scheme, there are two methods to detect :

(a) Completely centralized algorithm

(b) Ho-Ramamoorthy Algorithm

Completely Deadlock Detection : Centralized deadlock detection attempts to imitat[e] the non-distributed algorithm through a central coordinator. Each machine is responsible fo[r] maintaining a resource graph for its processes and resources. A *central coordinator* maintain[s] the resource utilization graph for the entire system. This graph is the union of the individua[l] graphs. If the coordinator detects a cycle, it finishes off one process to break the deadlock.

In the non distributed case, all the information on resource usage lies on one system an[d] the graph may be constructed on that system. In the distributed case, the individual subgraph[s] have to be propogated to a central coordinator. A message can be sent each time an arc i[s] added or deleted. If optimization is needed, a list of added or deleted arcs can be sen[t] periodically to reduce the overall number of message sent. Here is an example (fron[m] Tanenbaum). Suppose machine A has a process P_0 which holds resource S and want[s] resource R, which is held by P_1. The local graph on A is shown in Fig. 4.1. Another machine B

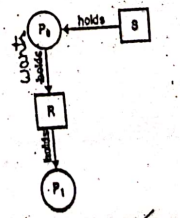


Fig. 4.1. Resource graph on A

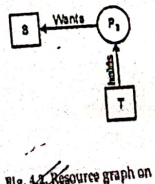


Fig. 4.2. Resource graph on B

has a process P_2 which is holding resource T and wants resource S. Its local graph is shown in Fig. 4.2. Both of these machines send their graphs to the central coordinator, which maintains the union Fig. 4.3. All is well. There are no cycles and hence, no deadlock. Now two events occur. Process P_1 release resource R and asks machine B for resource T. Two messages are sent to the coordinator :

message 1 (from machine A) : *"releasing R"*

message 2 (from machine B) : *"waiting for T"*

This should cause no problems (no deadlock). However, if message 2 arrives first, the coordinator would then construct the graph as Fig. 4.4. and detect a deadlock. Such a condition is known as **false deadlock**. A way to fix this is to use Lamport's algorithm to impose global time ordering on all machines. Alternatively, if the coordinator suspects deadlock, it can send a reliable message to every machine asking whether it has any.

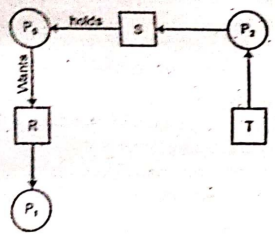


Fig. 4.3. Resource graph on coordinator

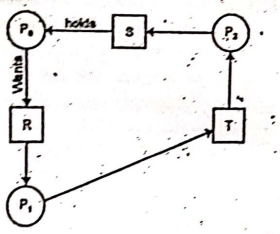


Fig. 4.4. False deadlock

Ho-Ramamoorthy Algorithm : ~~This algorithms works in two phases :~~

(a) Two phase algorithm

(b) One phase algorithm

Ho and Ramamoorthy gave two centralized deadlock detection algorithms,

(a) Two-phase algorithm : In the two-phase algorithm, every site maintains a status table that contains the status of all processes initiated at that site. The status of a process includes all resources locked and all resources being waited upon. Periodically, a designated site requests the status table from all sites, constructs a WFG from the information received, and searches it for cycles. If there is no cycle, then the system is free from deadlocks, otherwise, the designated site again requests status tables from all the sites and again constructs a WFG, using *only* those transactions which are common to both reports. If the same cycle is detected again, the system is declared deadlock.

Steps :

- Each site has a status table of locked and waited resources.
- The control site will periodically ask for this table from each node.
- The control node will search for cycles and, if found,
- Only the information common in both reports will be

cycle.

(b) One-phase algorithm : The one-phase algorithm requires only

each site; however, each site maintains two status tables : a

status table. The resource status table at a site keeps track of the transactions

or are waiting for resources stored at that site. The process status table at a site

the resources locked by or waited for all the transactions at the site. Periodically,

site requests both tables from every site, constructs a WFG using only those transactions

which the entry in the resource table matches the corresponding entry in the process table,

and searches the WFG for cycles. If no cycle is found, then the system is not deadlocked,

otherwise a deadlock is detected.

Steps

- Each site keeps two tables; process status and resource status.
- The control site will periodically ask for these tables (both together in a single message) from each node.
- The control site will then build and analyze the WFG, looking for cycles and resolving them when found.

~~phase algorithm~~ ~~phase algorithm~~ ~~tables and exchanges bigger messages (each message contains two tables instead of one~~

4.5.2. Distributed Deadlock Detection

Global state detection based deadlock detection algorithms exploit the following facts :

- A consistent snapshot of a distributed system can be obtained without freezing the underlying computation.
- A consistent snapshot may not represent the system state at any moment in time, but if a stable property holds in the system before the snapshot collection is initiated, this property will still hold in the snapshot.

Therefore, distributed deadlocks can be detected by taking a snapshot of the system and examining it for the condition of a deadlock.

Distributed deadlock detection algorithms can be divided into following four classes :

- Path-pushing
- Edge-chasing
- Diffusion computation
- Global state detection

...ing all the resources when it is finished. Thus [illegible] ...if all the resources are acquired together, and th... ...all the resources ...available

...ahing Algorithms

In path-pushing algorithms, distributed deadlocks are detected by maintaining an explicit global WFG.

- The basic idea is to build a global WFG for each site of the distributed system.
- In this class of algorithms, at each site whenever deadlock computation is performed, it sends its local WFG to all the neighbouring sites.
- After the local data structure of each site is updated, this updated WFG is then passed along to other sites, and the procedure is repeated until some site has a sufficiently complete picture of the global state to announce deadlock or to establish that no deadlocks are present.
- This feature of sending around the paths of global WFG has led to the term path-pushing algorithms.

In path-pushing deadlock detection algorithms, information about the wait-for dependencies is propagated in the form of paths. Obermarck's algorithm is chosen to illustrate a path-pushing deadlock detection algorithm.

Obermarck's algorithm has following two interesting features :

- The nonlocal portion of the global transaction wait for (TWF) graph at a site is abstracted by distinguished node (called External or Ex), which helps in determining potential multisite deadlocks without requiring a huge global TWF graph to be stored at each site.
- Transactions are totally ordered, which reduces the number of messages and consequently, decreases deadlock detection overhead. It also ensures that exactly one transaction in each cycle detects the deadlock.

Obermarck's Algorithm : Deadlock detection at a site follows the following iterative process :

1. The site waits for deadlock-related information (produced in Step 3 of the previous deadlock detection iteration) from other sites. (Note that deadlock-related information is passed by sites in the form of path).
2. The site combines the received information with its local TWF graph to build an updated TWF graph. It then detects all cycles and breaks only those cycles, which do not contain the node 'Ex'. Note that these cycles are local to this site. All other cycles have the potential to be a part of global cycle.
3. For all cycles 'Ex T_1 T_2 Ex', which contain the node 'Ex' (these cycles are potential candidates for global deadlocks), the site transmits them in string from 'Ex, T_1, T_2, Ex' to all other sites where an agent of T_2 is waiting for a resource being held by another transaction. The algorithm reduces message traffic by lexically ordering transactions and sending the string (Ex, T_1, T_2, T_3, Ex' to other sites only if T_1 is higher than T_3 in the

lexical ordering. Also, for a deadlock, the highest priority transaction detects th... deadlock.

Edge-Chasing Algorithms

- In an edge-chasing algorithm, the presence of a cycle in a distributed graph structur... is to be verified by propagating special messages called probes, along the edges of th... graph.
- These probe messages are different than the request and reply messages.
- The information of cycle can be deleted by a site if it receives the matching probe sen... by it previously.
- Whenever a process that is executing receives a probe message, it discards this messag... and continues.
- Only blocked processes propagate probe messages along their outgoing edges.
- Main advantages of edge-chasing algorithms is that probes are fixed size message... which are normally very short.

Edge-Chasing Algorithm : Chandy et al.'s algorithm uses a special message called probe. A probe is a triplet (i, j, k) [illegible] probe message travels along the edges of the global TWF graph, and a deadlock is detecte... when a probe message returns to its initiating process. The system maintains a Boolean array dependent, for each process P_i, where *dependent* (j) is true only if P_i knows that P_j is depender... on it. Initially, *dependent* (j) is false for all i and j.

Chandy Et Al' Algorithm : To determine if a blocked process is deadlocked, the syster... executes the following algorithm :

If P_i is locally dependent on itself
then declare a deadlock
else for all P_j and P_k such that
(a) P_i is locally dependent upon P_j and
(b) P_j is waiting on P_k and
(c) P_j and P_k are on different sites,
send probe (i, j, k) to the home site of P_k
On the receipt of probe (I, j, k), the site takes the following actions :
if
(a). P_k is blocked, and
(b) $dependent_k(i)$ is false, and
(c) P_k has not replied to all requests of P_j
then
begin
$dependent_k(i)$ = true;
if k = i

then declare that P_i is deadlocked
else for all P_m and P_n such that
(a) P_i is locally dependent upon P_m, and
(b) P_m is waiting on P_n, and
(c) P_m and P_n are on different sites,
send probe (i, m, n) to the home site of P_n
end.

Thus, a probe message is successively propagated along the edges of the global TWF graph and a deadlock is detected when a probe message returns to its initiating process. Chandy et al.'s algorithm sends one probe message (per deadlock detection initiation) on each edge of the WFG, which spans two sites. Thus, the algorithm at most exchanges $m(n-1)/2$ messages to detect a deadlock that involves m processes and spans over n sites. The size of messages exchanged is fixed and very small (only three integer words). The delay in detecting the deadlock is $O(n)$.

For example, process P_1 initiates deadlock detection, it sends probe (1, 3, 4) to S_2. Since P_6 is waiting for P_8 and P_7 waiting for P_{10}, S_2 sends probes (1, 6, 8) and (1, 7, 10) to S_3, which in turns sends probe (1, 9, 1) to S_1.

On receiving probe (1, 9, 1), S_1 declares that P_1 is deadlocked.

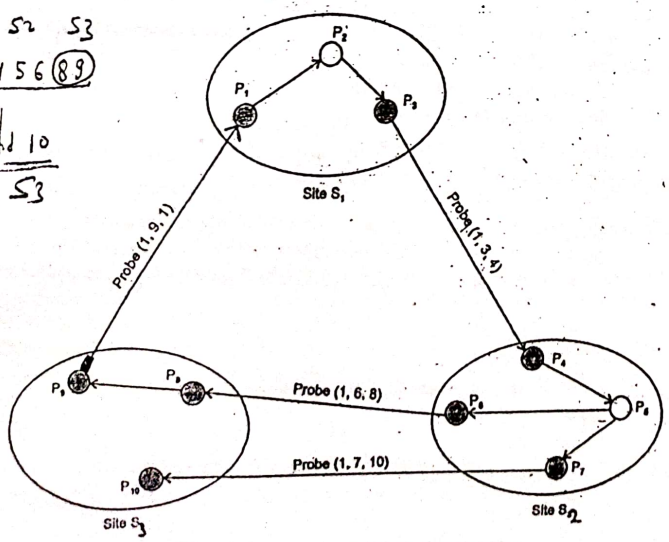


Fig. 4.5. Eg. of Chandy Et Al edge Chasing Algorithm

Distributed Deadlocks

This algorithms at most exchange $m(n-1)/2$ messages to detect a ... m processes and spans over n sites. The size of message exchanged is fixed. ... complexity is of order $O(n)$.

Diffusing Computations Based Algorithms

- In *diffusion computation* based distributed deadlock detection algorithm ... detection computation is diffused through the WFG of the system.
- These algorithms makes use of echo algorithms to detect deadlocks.
- This computation is superimposed on the underlying distributed computation. ... computation terminates, the initiator declares a deadlock.
- To detect a deadlock, a process sends out query messages along all the outgoing edges in the WFG.
- These queries are successively propagated (*i.e.*, diffused) through the edges of the WFG.

Chandy et al.'s Diffusion Computation Based Algo

- **Initiate a diffusion computation for a blocked process P_i:** Send query (i, i, j) to each process P_j in the dependent set DS_i of P_i; $num_i(i) := |DS_i|$; $wait_i(i) := true$.
- **When a blocked process P_k receives a query (i, j, k):** If this is the engaging query for process P_k then
  send query (i, k, m) to all P_m in its dependent set DS_k;
  $num_k(i) := |DS_k|$; $wait_k(i) := true$
  else if $wait_k(i)$ *then send a reply* (i, k, j) to P_j
- **When a process P_k receives a reply (i, j, k):**
  If $wait_k(i)$ then begin $num_k(i) := num_k(i) - 1$;
  if $num_k(i) = 0$
  then if $i = k$ then *declare a deadlock*
  else send reply (i, k, m) to the process P_m which sent the engaging query
- In practice, several diffusion computations may be initiated for a process (A diffusion computation is initiated every time the process get blocked), but at any time only a diffusion computation is current for any process.
- However, message for outdated diffusion computations may still be in transit.
- The current diffusion computation can be distinguished from outdated ones by using sequence numbers.

mance Analysis : For every deadlock detection, the algorithm exchanges a query and e reply messages, where $e = n(n-1)$ is the number of edges.

Global State Detection Based Algorithm

Global state detection based deadlock detection algorithms exploit the following facts :

1. A consistent snapshot of a distributed system can be obtained without freezing the underlying computation.
2. If a stable property holds in the system before the snapshot collection is initiated, this property will still hold in the snapshot.

Therefore, distributed deadlocks can be detected by taking a snapshot of the system and examining it for the condition of a deadlock.

Global State Detection Algorithm – Data Structures

$wait_i$: boolean (:= false) /* records the current status */
t_i : integer (:= 0) /* current time*/
in (i) : set of nodes whose requests are outstanding at i
out (i) : set of nodes on which i is waiting
/p_i : integer (:= 0)/ number of replies required for unblocking*/*
w_i : real (:= 1.0) / weight to detect termination of deadlock detection algorithm*/*

Global State Detection Algorithm

- **REQUEST_SEND (i) :**
  /*executed by node *i* when it blocks on a p_i - out of - q_i request*/
  For every node *j* on which *i* is blocked do
  out (i) ← out (i) U{i) ; send REQUEST (i) to j;
  set p_i to the number of replies needed; $wait_i$ = true
- **REQUEST_RECEIVE (j) :**
  /* executed by node *i* when it receives a request made by *j**/
  in (i) ← in (i) U {j}
- **REPLY_SEND (j) :**
  /*executed by node i when it replies to a request by *j**/
  in (i) ← in (i) - {j} send REPLY (i) to j;

XXX

4.5.3. Hierarchical Deadlock Detection

An alternative to centralized deadlock detection is the building of a hierarchy of deadlock detection detectors. Deadlocks that are local to a single site would be detected at that site

using the local WFG. Each site also sends its local WFG to the deadlock detector at the next level. Thus, distributed deadlocks involving two or more sites would be detected by a deadlock detector in the lowest level that has control over these sites.

The hierarchical deadlock detection reduces the dependence on the central sites, thus, reducing the communication cost. It is however, considerably more complicated to implement and would involve nontrivial modification to the lock and transactions manager algorithms.

Distributed deadlock algorithms delegate the responsibility of detection deadlocks to individual sites. Thus, in the hierarchical deadlock detection, there are local detectors at each site, which communicate their local WFGs with one another.

Ho-Ramamoorthy algorithm

- Uses only two levels
  - Master control nodes
  - Cluster control nodes
- Cluster control nodes are responsible for detecting deadlock among their members and reporting dependencies outside their cluster to the Master control node (they use the one phase version of the Ho-Ramamoorthy algorithm discussed earlier for centralized detection).
- The master control node is responsible for detecting intercluster deadlocks.
- Node assignment to cluster is dynamic.

The Menasce–Muntz Algorithm

- Leaf controllers allocate resources
- Branch controllers are responsible for the finding deadlock among the resources that their children span in the tree.
- Network congestion can be managed
- Node failure is less critical than in fully centralized
- Detection can be done many ways such as :
  - Continuous allocation reporting
  - Periodically allocation reporting

REVIEW QUESTIONS

1. What are the phantom deadlock or false deadlock ? Explain the algorithm which could detect phanton deadlocks.
2. What are differences in resources and communication deadlocks?

In distributed system, where sites (or processors) often compete as well as cooperate to achieve a common goal, it is often required that sites reach mutual agreement. For example, in distributed database systems, data managers at sites must agree on whether to commit or to abort a transaction. Reaching an agreement typically requires that sites have knowledge about the values of other sites. For example, in distributed commit, a site should know the outcome of local commit at each site.

When the system is free from failures, an agreement can easily be reached among the processors (or sites). For example, processors can reach an agreement by communicating their values to each other and then by taking a majority vote or a minimum, maximum, mean etc. of those values. However, when the system is prone to failure, this method does not work. This is because faulty processors can send conflicting values to other processors preventing them from reaching an agreement. In the presence of faults, processors must exchange their values with other processors and relay the values received from other processor several times to isolate the effects of faulty processors. A processor refines its value as it learns the values of other processors. (This entire process of reaching an agreement is called an agreement protocol).

5.2. SYSTEM MODEL

Agreement problems have been studied under the following system model :

- There are n processor in the system and atmost m of the processors can be faulty.
- The processors can directly communicate with other processors by message passing. Thus, the system is logically fully connected.
- A receiver processor always knows the identity of the sender processor of the message.
- The communication medium is reliable (*i.e.*, it delivers all messages without introducing any errors) and only processors are prone to failures.

5.2.1. Synchronous Vs Asynchronous Computation

In a synchronous computation, processes in the system run in lock step manner, where in each step, a process receives messages (sent to it in the previous step), performs a computation, and sends messages to other processes (received in the next step). A step of a synchronous computation is also referred to as a round. In synchronous computation, a process knows all the messages it expects to receive in a round. A message delay or a slow process can slow down the entire system or computation.

In an asynchronous computation, on the other hand, the computation at processes does not proceed in lock steps. A process can send and receive messages and perform computation at any time.

5.2.2. Model of Processor Failures

In agreement problems, we consider a very general model of processor failures. A processor can fail in three modes : crash fault, omission fault, and malicious fault. In a crash fault, a processor stops functioning and never resumes operation. In an omission fault, a processor "omits" to send messages to some processors. (These are the messages that the processor should have sent according to the protocol or algorithm it is executing). For example, a processor is supposed to broadcast a message to all other processors, but it sends the message to only a few processors. In a malicious fault, a processor behaves randomly and arbitrarily. For example, a processor may send fictitious messages to other processors to confuse them. Malicious faults are very broad in nature and thus, most other conceivable faults can be treated as malicious faults. Malicious faults are also referred to as Byzantine faults.

5.2.3. Authenticated vs Non-Authenticated Message

There are two types of messages : authenticated and non-authenticated. In an authenticated message system, a faulty processor cannot forge a message or change the content of a received message. A processor can verify the authenticity of a received message. An authenticated message is also known as a signed message.

In a non-authenticated message system, a faulty processor can forge a message and claim to have received it from another processor or change the contents of a received message before it relays the message to other processor. A non-authenticated message is also known as an oral message.

5.3. CLASSIFICATION OF AGREEMENT PROBLEMS

This section introduces the problem of consensus [Pease *et al.* 1980 Lamport *et. al* 198 and the related problems of Byzantine generals and interactive consistency. We shall refer t these collectively as problems of agreement. Roughly speaking, the problem is for processes t agree on a value after one or more of the processes has proposed what that value should be

This section defines consensus more precisely and relates it to three related agreemer problems : Byzantine generals, interactive consistency and totally ordered multicast. We g on to examine under what circumstances the problems can be solved, and sketch some solution In particular, we shall discuss the well-known impossibility result of Fischer *et. al.* [1985], whic states that in an asynchronous system a collection of processes containing only one fault process cannot be guaranteed to reach consensus. Finally, we consider how practical algorithm exist despite the impossibility result.

There are three well known problems in distributed systems :

(a) Consensus problem
(b) Byzantine agreement (generals) problem
(c) Interactive consistency problem

CBI

Problem Definitions : Our system model includes a collection of processes P_i ($i = 1, 2, \ldots N$) communicating by message passing. An important requirement that applies in many practical situations is for consensus to be reached even in the presence of faults. We assume, as before, that communication is reliable but that processes may fail. In this section, we shall consider Byzantine (arbitrary) process failures, as well as crash failures. We shall sometimes specify an assumption that up to some number f of the N processes are faulty, that is, they exhibit some specified types of faults; the remainder of the processes are correct.

5.3.1. Consensus Problem

To reach consensus, every process P_i beings in the undecided state and proposes a single value v_i drawn from a set D ($i = 1, 2, \ldots, N$). The processes communicate with one another

exchanging values. Each process then sets the value of a *decision variable* d_i. In doing so, it enters the *decided* state, in which it may no longer change d_i ($i = 1, 2, \ldots, N$). Fig. 5.1. shows three processes engaged in a consensus algorithm. Two processes propose 'proceed' and a third proposes 'abort' but then crashes. The two processes that remain correct each decide 'proceed'.

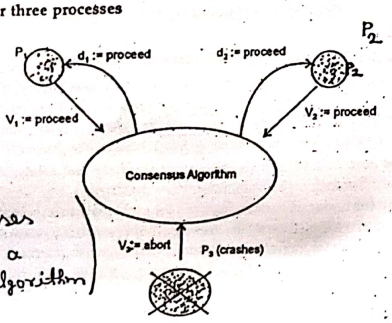
In a consensus algorithm, the following conditions should hold for every execution of it:

Termination : Eventually each correct process sets its decision variable.

Agreement : The decision value of all correct processes is the same; if P_i and P_j are correct and have entered the *decided* state, then $d_i = d_j$ ($i, j = 1, 2, \ldots, N$).

Integrity : If the all correct processes propose the same value, then any correct process in the *decided* state has chosen that values.

Consensus for three processes



Three processes engaged in a consensus algorithm

Fig. 5.1.

To understand how the formulation of the problem translates into an algorithm, consider a system, in which processes cannot fail. It is then straight forward to solve consensus. For example, we can collect the processes into a group and have each process reliably multicast its proposed value to the members of the group. Each process waits until it has collected all N values (including its own). It then evaluates the function majority ($v_1, v_2, \ldots v_N$), which returns the value that occurs most often among its arguments, or the special value $\perp \notin D$ if no majority exists. Termination is guaranteed by the reliability of the multicast operation. Agreement and integrity are guaranteed by the definition of majority, and the integrity property of a reliable multicast. Every process receives the same site of proposed values, and every process evaluates the same function of those values. So they must all agree, and if every process proposed the same value, then they all decide on this value.

5.3.2. Byzantine Generals Problem

In the informal statement of the *Byzantine generals problem* [Lamport et a
more generals have agree to attack or to retreat. One, the commander issues
others, lieutenants to the commander, are to decide to attack or to retreat. But o
the generals may be "treacherous" – that is faulty. If the commander is treacherous, he
attacking to one general and retreating to another. If a lieutenant is treacherous, he
his peers that the commander has told him to attack and to others that they are to retr

The Byzantine generals problem differs from consensus in that a distinguished pro
supplies a value that the others are to agree upon, instead of each of them proposing a val
The requirements are :

Termination : Eventually each correct process sets its decision variable.

Agreement : The decision value of all correct processes is the same. If P_i and P_j are correct and have entered the *decided* state, then $d_i = d_j$ ($i, j = 1, 2, \ldots N$).

Integrity : If the commander is correct, then all correct processes decide on the value that the commander has proposed.

5.3.3. Interactive Consistency Problem

The interactive consistency problem is another variant of consensus in which every process proposes a single value. The goal of the algorithm is for the co[illegible] to agree on a vector of values, ~~one for each process~~. We shall call this the '*decision vector*'. For example, the goal could be for each set of processes to obtain the same information about their respective states.

The requirements for interactive consistency are :

Termination : Eventually every correct process sets its decision variable.

Agreement : The decision vector of all correct processes is the same.

Integrity : If P_i is correct, then all correct processes decide on V_i as the ith component of their vector.

Relating consensus to other problems : Although it is common to consider the Byzantine generals problem with arbitrary process failures, in fact each of the three problems – consensus Byzantine generals and interactive consistency – is meaningful in the context of either arbitrary or crash failures. Similarly, each can be framed assuming either a synchronous or an asynchronous system.

It is sometimes possible to derive a solution to one problem using a solution to another This is a very useful property, because it increases our understanding of the problems and also by reusing solutions we can potentially save on implementation effort and complexity.

Suppose that there exists solutions to consensus (C), Byzantine generals (BG) and interactive consistency (IC) as follows :

$C_i(v_1, v_2, \ldots, v_N)$ returns the decision value of p_i in a run of the solution of the consensus problem, where $v_1, v_2, \ldots, v_N$ are the values that the processes proposed.

$BG_i(j, v)$ returns the decision value of p_i in a run of the solution to the Byzantine generals problem, where p_j the commander, proposes the value v.

$IC_i(v_1, v_2, \ldots, v_N)[j]$ returns the jth value in the decision vector of p_i in a run of the solution to the interactive consistency problem, where $v_1, v_2, \ldots, v_N$ are the values that the processes proposed.

It is possible to construct solutions out of the solutions to other problems. We give three examples :

IC from BG : We construct a solution to *IC* from *BC* by running *BG N* times, once with each process p_i ($i, j = 1, 2 \ldots N$) acting as the commander :

$$IC_i(v_1, v_2, \ldots, v_N)[j] = BG_i(j, v_j) \quad (i, j = 1, 2, \ldots, N)$$

C from IC : We construct a solution to *C* from *IC* by running *IC* to produce a vector of values at each process, then applying an appropriate function on the vector's values to derive a single value :

$$C_i(v_1, \ldots, v_N) = majority\ (IC_i(v_1, \ldots, v_N)[1], \ldots IC_i(v_1, \ldots, v_N)[N])$$

($i = 1, 2, \ldots N$), where *majority* is as defined above.

BG from C : We construct a solution to *BG* from *C* as follows :

- The commander p_j sends its proposed value v to itself and each of the remaining processes.
- All processes run *C* with the values $v_1, v_2, \ldots v_N$ that they receive (p_j may be faulty).

Byzantine Generals Problem in a Synchronous System : We discuss the Byzantine generals problem in a synchronous system. Unlike the algorithm for consensus described in the previous section, here we assume that processes can exhibit arbitrary failures. That is, a faulty process may send any messge with any value at any time; and it may omit to send any message. Up to f of the N processes may be faulty. Correct processes can detect the absence of a message through a timeout; but they cannot conclude that the sender has crashed, since it may be silent for sometime and then send messages again.

We assume that the communication channels between pairs of processes are private. If a process could examine all the messages that other processes send, then it could detect the inconsistencies in what a faulty processes sends to different processes. Our default assumption of channel reliability means that no faulty processes can inject message into the communication channel between correct processes.

5.4. BYZANTINE AGREEMENT AMONG THREE PROCESSORS

Lamport *et al.* [1982] considered the case of three processes that send unsigned messages to one another. They showed that there is no solution that guarantees to meet the conditions of

the Byzantine generals problem, if one process is allowed to fail. They generalized this res to show that no solution exists if $N \leq 3f$, for unsigned (they call them 'oral') message.

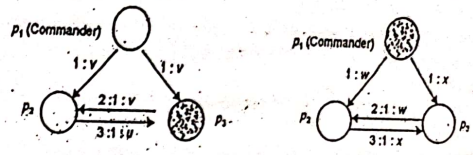


Fig. 5.2. Faulty processes are shown shaded

5.4.1. Impossible Result

Fig. 5.2 shows two scenarios, in which just one of three process is faulty. In the l configuration, one of the lieutenants, p_3 is faulty; on the right the commander, p_1 is faulty. Ea scenario in Fig. 5.2 shows two rounds of messages; the values the commander sends, and t values that the lieutenants subsequently send to each other. The numeric prefixes serve specify the sources of messages and to show the different rounds. Read the ':' symbol messages as 'says'; for example, '3 : 1 : u' is the message '3 says 1 says 'u'.

In the left-hand scenario, the commander correctly sends the same value v to each of t other two processes, and p_2 correctly echoes this to p_3. However, p_3 sends a value $u \neq v$ to p_2. p_2 knows, at this stage it has received differing values; it cannot tell which were sent out by t commander.

In the right-hand scenario, the commander is faulty and sends differing values lieutenants. After p_3 has correctly echoed the value x that it received, p_2 is in the same situati as it was in when p_3 was faulty; it has received two differing values.

If a solution exists, then process p_2 is bound to decide on value v when the commander correct, by the integrity condition. If we accept that no algorithm can possibly distingui between the two scenarios, p_2 must also choose the value sent by the commander in the righ hand scenario.

Following exactly the same reasoning for p_3, assuming that it is correct, we are forced conclude, by symmetry, that p_3 also chooses the value sent by the commander as its decisi value. But this contradicts the agreement condition (the commander sends differing values, it is faulty). So no solution is possible. $\Rightarrow$ $N \leq 3f$

5.4.2. Impossibility with $N \leq 3f$

Pease *et al.* generalized the basic impossibility result for three processes, to prove that a solution is possible if $N \leq 3f$. In outline, the argument is as follows : Assume that a soluti exists with $N \leq 3f$. Let each of three processes p_1, p_2 and p_3 use the solution to simulate t behaviour of n_1, n_2 and n_3 generals, respectively, where $n_1 + n_2 + n_3 = N$ and $n_1, n_2, n_3 \leq N/3$. V assume, furthermore, that one of the three processes is faulty. Those of p_1, p_2 and p_3 that a

correct simulate correct generals; they simulate that interactions of their own generals internally and send messages from their generals to those simulated by other processes. The faulty process's simulated generals are faulty; the messages that it sends as part of the simulation to the other two processes may be spurious. Since $N \leq 3f$ and $n_1, n_2, n_3 \leq N/3$, at most f simulated generals are faulty.

Solution with One Faulty Process : There is no sufficient space to describe fully, the algorithm of Please *et al.* that solves that Byzantine generals problem in a synchronous system with $N \geq 3f + 1$. Instead, we give the operation of the algorithm for the case $N \geq 4, f = 1$ and illustrate it for $N = 4, f = 1$.

The correct generals reach agreement in two rounds of messages :

- In the first round, the commander sends a value to each of the lieutenants.
- In the second round, each of the lieutenants sends the value it received to its peers.

A lieutenant receives a value from the commander, plus $N - 2$ values from it peers. If the commander is faulty then all the lieutenants are correct and each will have gathered exactly the set of values that commander sends out. Otherwise, one of the lieutenants is faulty; each of its correct peers receives $N - 2$ copies of the value that the commander sends, plus a value that the faulty lieutenants sent to it.

5.5. LAMPORT ALGORITHM FOR BYZANTINE AGREEMENT SOLUTION

Lamport et al proposed an algorithm for solving the Byzantine agreement problem. This algorithm is also called *Lamport Shostak Peace Algorithm.*

The algorithms works if and if only if

$n >= 3m + 1$ where $N \geq 3f + 1$

n = total number of processors

m = number of faulty processors

OM(0) Algorithm ($m = 0$)

Step 1 : Source processor sends its initial value to each and every processor.

Step 2 : Each processor uses this received value (if no value is received, it uses a default value 0).

OM(m) Algorithm (m > 0)

Step 1 : The source processor sends its value to every processor.

Step 2 : For each i, let vi be the value of processor i receives from the source. (If it receives no value, then it uses a default value of 0). Processor i acts as the new source and initiates algorithm OM(m–1) wherein it sends the value vi to each of the $n - 2$ other processors.

Step 3 : For each i and each j, let vj be the value processor i received from processor j in Step 2 using algorithm OM (m – 1). If it receives no value, then it uses a default value of 0. Processor i uses the value majority ($v1, v2, \ldots\ldots v_{n-1}$).

This algorithm is quite complex. The processors are successively divided into smaller and smaller groups and the Byzantine agreement is recursively achieved with each group of

processors. Step 3 is executed during the folding phase of recursion where a majority is applied to select majority value.

The execution of the algorithm OM(m) invokes $n - 1$ seperate execution of the algorithm OM($m - 1$), each of which invokes $n - 2$ executions of the algorithm of the algorithm OM($m - 2$). So message complexity of the algorithm is $O(n^m)$.

We now illustrate the algorithm that we have just outlined for the case of four generals. Fig. 5.3 shows two scenarios similar to those in Fig. 5.2, but in this case there are four processes one of which is faulty. In the left-hand configuration, one of the lieutenants, p_3 is faulty, on the right, the commander, p_1 is faulty.

In the left-hand case, the two correct lieutenant processes agree, deciding on the commander's values:

p_2 decided on *majority* $(v, u, v) = v$

p_4 decides on *majority* $(v, v, w) = b$

In the right-hand case, the commander is faulty, but the three correct processes agree :

p_2, p_3 and p_4 decide on *majority* $(u, v, w) = \perp$ (the special value $\perp$ applies where no majority of values exists).

The algorithm takes account of the faulty process that may omit to send a message. If a correct process does not receive a message within a suitable time limit (the system is synchronous), it proceeds as though the faulty process had sent it the value $\perp$.

Four Byzantine Generals

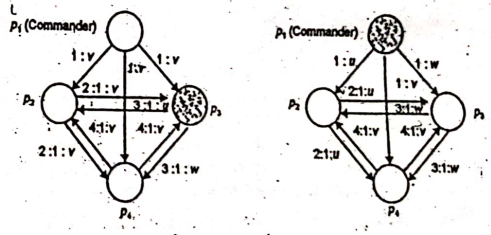


Fig. 5.3. Faulty processes are shown shaded

5.6. APPLICATIONS OF AGREEMENT ALGORITHMS

Algorithms for agreement problems find applications in problems where processors should reach an agreement regarding their values in the presence of malicious failures. The main applications are :

(*i*) Clock synchronization

(*ii*) Atomic commit in distributed database

In distributed systems, it is often necessary that sites or processes maintain physical clocks are synchronized with one another. Since physical clocks have a drift problem, they must periodically resynchronized. The description of clock synchronization in this section is based on the work of Lamport and Miller Smith and the assumption are :

A 1 : All clocks are initially synchronized to approximately the same values.

A 2 : A non-faulty process's clock runs at approximately the correct rate.

A 3 : A non-faulty process can read the clock value of another non-faulty process with at most a small error e.

A clock synchronization algorithm must satisfy following two conditions :

(a) At any time, the values of clocks of all non-faulty processes must be approximately equal.

(b) There is a small bound on the amount, by which the clock of a non-faulty process is changed during each resynchronization.

5.6.2. Atomic Commit in Distributed Database

An atomic commit is an operation, in which a set of distinct changes is applied as a single operation. If all the changes are applied, the atomic commit is said to have succeeded. If there is a failure before the atomic commit can be completed, (*e.g.*, a network cable is disconnected or a conflict arises that cannot be automatically resolved) the "commit" is aborted and all changes that have taken place are reversed (rolled back). In either case, the atomic commit leaves the system in a consistent state. The word atom is used in its classical sense : an indivisible unit.

It has been proven that *no* algorithm can solve the problem via the proof of the Two Generals' Problem. However, algorithms such as the Two-phase commit protocol and Three-phase commit protocol can have *some* of the atomic commitment problems.

Nowadays, atomic commits are most often encountered in database system when committing multiple sets of changes at once. These changes can be different update statements to the same table or changes that span multiple databases.

Atomic commits are employed by modern revision control systems, allowing committing— uploading to the source — changes in multiple files (called a *changeset*) while guaranteeing that all files get fully uploaded and merged. In an atomic commit, typically the files that are committed together are concerned with a single modification, and everything changed in that modificating should be included in the commit. In this way, the code base remains stable, users who update their working copy do not miss changes left to be committed in somebody else's working copy, the changeset in not too messy to read through, and if the atomic commit is rolled back, the single modification is removed from the code base.

SOLVED QUESTIONS

1. Show that Byzantine agreement can be reached for three generals, with one of them faulty if the generals digitally sign their messages.

Ans. Any lieutenant can verify the signature on any message. No lieutenant can forge another signature. The correct lieutenants sign what they each received and send it to one another.

A correct lieutenant decides x if he receives messages [x][signed commander] and either [[x] (singed commander)] (signed lieutenant) or a message that either has a spoiled lieutenant signature or a spoiled commander signature.

Otherwise, he decides on a default course of action (retreat, say).

A correct lieutenant either sees the proper commander's signature on two different courses of action (in which case both correct lieutenants decide 'retreat'); or, he sees one good signature direct from the commander and one improper commander signature (in which case he decides on whatever the commander signed to do); or he sees no good commander signature (in which case both correct lieutenants decided 'retreat').

In the middle case, either the commander sends an improperly signed statement to the other lieutenant, or the other lieutenant is faulty and is pretending that he received an improper signature. In the former case, both correct lieutenants will do whatever the (albeit faulty) commander told one of them to do in a signed message.

In the latter case, the correct lieutenant does what the correct commander told him to do.

2. Construct a solution to reliable, totally ordered multicast in a synchronous system, using a reliable multicast and a solution to the consensus problem. *(UPTU–2005)*

Ans. To RTO-multicast (reliable, totally-ordered multicast) a message m, a process attaches a totally-ordered, unique identifier to m and R-multicasts it.

Each process records the set of messages it has R-delivered and the set of messages it has RTO-delivered. Thus, it knows which messages have not yet been RTO-delivered.

From time to time, it proposes its set of not-yet-RTO-delivered messages as those that should be delivered next. A sequence of runs of the consensus algorithm takes place, where the k'th proposals ($k = 1, 2, 3, \ldots\ldots$) of all the processes are collected and a unique decision set of messages is the result. When a process receives the k'th consensus decision, it takes the intersection of the decision value and its set of not-yet-RTO-delivered messages and delivers them in the order of their identifiers, moving them to the record of messsages it has RTO-deliverd. In this way, every process delivers messages in the order of the concatenation of the sequence of consensus results. Since the consensus results given to different correct processes are identical, we have a RTO multicast.