

~~Deadlock~~ Detection in Distributed Systems

Ajay Kshemkalyani and Mukesh Singhal

Distributed Computing: Principles, Algorithms, and Systems

Chapter 10

Introduction

- Deadlocks is a fundamental problem in distributed systems.
- A process may request resources in any order, which may not be known a priori and a process can request resource while holding others.
- If the sequence of the allocations of resources to the processes is not controlled, deadlocks can occur.
- A deadlock is a state where a set of processes request resources that are held by other processes in the set.

✓ Set of processes

Imp

- A distributed program is composed of a set of n asynchronous processes $p_1, p_2, \dots, p_i, \dots, p_n$ that communicates by message passing over the communication network. 1/2/2
- Without loss of generality we assume that each process is running on a different processor.
- The processors do not share a common global memory and communicate solely by passing messages over the communication network. 1/2/2

- There is no physical global clock in the system to which processes have instantaneous access.
- The communication medium may deliver messages out of order, messages may be lost garbled or duplicated due to timeout and retransmission, processors may fail and communication links may go down.
- We make the following assumptions:
 - The systems have only reusable resources.
 - Processes are allowed to make only exclusive access to resources.
 - There is only one copy of each resource.

1/11/21

V.V. / m!

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

sho be
resources
ache

us p source
acquire
karu m cheta

Wait-For-Graph (WFG)

U.V.T.P

→ wait-for graph

- The state of the system can be modeled by directed graph, called a wait for graph (WFG).
- In a 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 exists a directed cycle or knot in the WFG.

- Figure 1 shows a WFG, where process P_{11} of site 1 has an edge to process P_{21} of site 1 and P_{32} of site 2 is waiting for a resource which is currently held by process P_{21} .
- At the same time process P_{32} is waiting on process P_{33} to release a resource.
- If P_{21} is waiting on process P_{11} , then processes P_{11} , P_{32} and P_{21} form a cycle and all the four processes are involved in a deadlock depending upon the request model.

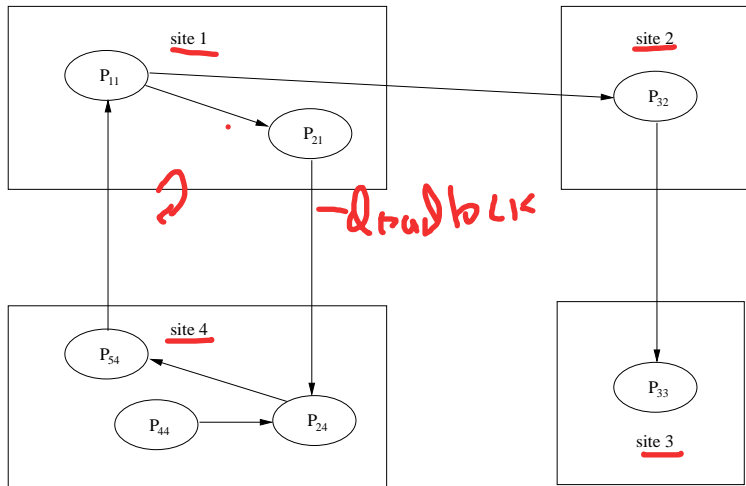


Figure 1: An Example of a WFG

Deadlock Handling Strategies

imp

link in OS

- There are three strategies for handling deadlocks, viz., deadlock prevention, deadlock avoidance, and deadlock detection.
- Handling of deadlock becomes highly complicated in distributed systems because no site has accurate knowledge of the current state of the system and because every inter-site communication involves a finite and unpredictable delay.
- ① • Deadlock prevention is commonly achieved either by having a process acquire all the needed resources simultaneously before it begins executing or by preempting a process which holds the needed resource.
- This approach is highly inefficient and impractical in distributed systems.

Bankers Algo
(practically not possible)

- in deadlock avoidance approach to distributed systems, a resource is granted to a process if the resulting global system state is safe (note that a global state includes all the processes and resources of the distributed system).
- However, due to several problems, deadlock avoidance is impractical in distributed systems.
- Deadlock detection requires examination of the status of process-resource interactions for presence of cyclic wait.
- Deadlock detection in distributed systems seems to be the best approach to handle deadlocks in distributed systems.

deadlock help improve knowledge

Issues in Deadlock Detection

- Deadlock handling using the approach of deadlock detection entails addressing two basic issues: First, detection of existing deadlocks and second resolution of detected deadlocks.
← two steps ✓
- Detection of deadlocks involves addressing two issues:
Maintenance of the WFG and searching of the WFG for the presence of cycles (or knots).
✓ ✓ ✓ ✓ ✓

Correctness Criteria: A deadlock detection algorithm must satisfy the following two conditions:

(i) Progress (No undetected deadlocks):

- The algorithm must detect all existing deadlocks in finite time.

- In other words, after all wait-for dependencies for a deadlock have formed, the algorithm should not wait for any more events to occur to detect the deadlock.

L page photo 1

(ii) Safety (No false deadlocks):

- The algorithm should not report deadlocks which do not exist (called *phantom* or *false deadlocks*).

Resolution of a Detected Deadlock

- ✓ Deadlock resolution involves breaking existing wait-for dependencies between the processes to resolve the deadlock. ✓
- It involves rolling back one or more deadlocked processes and assigning their resources to blocked processes so that they can resume execution. ✓

Models of Deadlocks

Distributed systems allow several kinds of resource requests.

The Single Resource Model

- In the single resource model, a process can have at most one outstanding request for only one unit of a resource.
- Since the maximum out-degree of a node in a WFG for the single resource model can be 1, the presence of a cycle in the WFG shall indicate that there is a deadlock.

The AND Model

- In the AND model, a process can request for more than one resource simultaneously and the request is satisfied only after all the requested resources are granted to the process.
- The out degree of a node in the WFG for AND model can be more than 1.
- The presence of a cycle in the WFG indicates a deadlock in the AND model.
- Since in the single-resource model, a process can have at most one outstanding request, the AND model is more general than the single-resource model.

- Consider the example WFG described in the Figure 1.
- P_{11} has two outstanding resource requests. In case of the AND model, P_{11} shall become active from idle state only after both the resources are granted.
- There is a cycle $P_{11} \rightarrow P_{21} \rightarrow P_{24} \rightarrow P_{54} \rightarrow P_{11}$ which corresponds to a deadlock situation.
- That is, a process may not be a part of a cycle, it can still be deadlocked. Consider process P_{44} in Figure 1.
- It is not a part of any cycle but is still deadlocked as it is dependent on P_{24} which is deadlocked.

The OR Model

- In the OR model, a process can make a request for numerous resources simultaneously and the request is satisfied if any one of the requested resources is granted.
- Presence of a cycle in the WFG of an OR model does not imply a deadlock in the OR model.
- Consider example in Figure 1: If all nodes are OR nodes, then process P_{11} is not deadlocked because once process P_{32} releases its resources, P_{32} shall become active as one of its requests is satisfied.
- After P_{32} finishes execution and releases its resources, process P_{11} can continue with its processing.
- In the OR model, the presence of a knot indicates a deadlock.

The AND-OR Model

AND-OR model

- A generalization of the previous two models (OR model and AND model) is the AND-OR model.
- In the AND-OR model, a request may specify any combination of *and* and *or* in the resource request.
- For example, in the AND-OR model, a request for multiple resources can be of the form *x and (y or z)*.
- To detect the presence of deadlocks in such a model, there is no familiar construct of graph theory using WFG.
- Since a deadlock is a stable property, a deadlock in the AND-OR model can be detected by repeated application of the test for OR-model deadlock.

The $\binom{p}{q}$ Model

$\forall CP \checkmark$

(resources theke
 $\binom{p}{q}$ veso jae hai)

- The $\binom{p}{q}$ model (called the P-out-of-Q model) allows a request to obtain any k available resources from a pool of n resources.
- It has the same in expressive power as the AND-OR model.
- However, $\binom{p}{q}$ model lends itself to a much more compact formation of a request.
- Every request in the $\binom{p}{q}$ model can be expressed in the AND-OR model and vice-versa.
- Note that AND requests for p resources can be stated as $\binom{p}{p}$ and OR requests for p resources can be stated as $\binom{p}{1}$.

✓ 1249

- ests.
made

Knapp's Classification

Distributed deadlock detection algorithms can be divided into four classes:

- path-pushing
- edge-chasing
- diffusion computation
- global state detection.

Path-Pushing 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 neighboring 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.

Edge-Chasing Algorithms

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

Diffusing Computations Based Algorithms

- In *diffusion computation* based distributed deadlock detection algorithms, deadlock detection computation is diffused through the WFG of the system.
- These algorithms make use of *echo algorithms* to detect deadlocks.
- This computation is superimposed on the underlying distributed computation. If this 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.

- When a blocked process receives first query message for a particular deadlock detection initiation, it does not send a reply message until it has received a reply message for every query it sent.
- For all subsequent queries for this deadlock detection initiation, it immediately sends back a reply message.
- The initiator of a deadlock detection detects a deadlock when it receives reply for every query it had sent out.

Global State Detection Based Algorithms

- 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 and
- 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.

~~Mitchell~~ and Merritt's Algorithm for the Single-Resource Model

imp

- Belongs to the class of edge-chasing algorithms where probes are sent in opposite direction of the edges of WFG.
- When a probe initiated by a process comes back to it, the process declares deadlock.
- Only one process in a cycle detects the deadlock. This simplifies the deadlock resolution – this process can abort itself to resolve the deadlock.

$label = (count, p_id)$

- Each node of the WFG has two local variables, called labels:

- 1 a private label, which is unique to the node at all times, though it is not constant, and
- 2 a public label, which can be read by other processes and which may not be unique.

- Each process is represented as u/v where u and v are the public and private labels, respectively.

- Initially, private and public labels are equal for each process.

- A global WFG is maintained and it defines the entire state of the system.

$inc \hookrightarrow \text{increase count}$

$label = (count, pid)$

- The algorithm is defined by the four state transitions shown in Figure 2, where $z = inc(u, v)$, and $inc(u, v)$ yields a unique label greater than both u and v labels that are not shown do not change.

$z = \max(u, v) + 1$

- Block creates an edge in the WFG.

- Two messages are needed, one resource request and one message back to the blocked process to inform it of the public label of the process it is waiting for.

- Activate denotes that a process has acquired the resource from the process it was waiting for.

- Transmit propagates larger labels in the opposite direction of the edges by sending a probe message.

$count \quad pid$

Condition

- 1)
 - Whenever a process receives a probe which is less than its public label, then it simply ignores that probe.
 - Detect means that the probe with the private label of some process has returned to it, indicating a deadlock.
 - The above algorithm can be easily extended to include priorities where whenever a deadlock occurs, the lowest priority process gets aborted.

Message Complexity:

If we assume that a deadlock persists long enough to be detected, the worst case complexity of the algorithm is $s(s-1)/2$ Transmit steps, where s is the number of processes in the cycle.

Chandy-Misra-Haas Algorithm for the AND Model

4th part of site - no deadlock

- Chandy-Misra-Haas's distributed deadlock detection algorithm for AND model is based on edge-chasing.
- The algorithm uses a special message called *probe*, which is a triplet (i, j, k) , denoting that it belongs to a deadlock detection initiated for process P_i and it is being sent by the home site of process P_j to the home site of process P_k .
- A probe message travels along the edges of the global WFG graph, and a deadlock is detected when a probe message returns to the process that initiated it.

$\leftarrow (i, j, i)$

we give this

- A process P_j is said to be *dependent* on another process P_k if there exists a sequence of processes $P_j, P_{i1}, P_{i2}, \dots, P_{im}, P_k$ such that each process except P_k in the sequence is blocked and each process, except the P_j , holds a resource for which the previous process in the sequence is waiting.

- Process P_j is said to be *locally dependent* upon process P_k if P_j is dependent upon P_k and both the processes are on the same site.

Data Structures

- Each process P_i maintains a boolean array, *dependent_i*, where *dependent_i(j)* is true only if P_i knows that P_j is dependent on it.
- Initially, *dependent_i(j)* is false for all i and j .

Karna's algorithm - *dependent*

Algorithm

The following algorithm determines if a blocked process is deadlocked:

- if P_i is locally dependent on itself then declare a deadlock else for all P_j and P_k such that

- P_i is locally dependent upon P_j , and
- P_j is waiting on P_k , and
- P_j and P_k are on different sites, send a probe (i, j, k) to the home site of P_k



- On the receipt of a probe (i, j, k) , the site takes the following actions. if

- P_k is blocked, and
- $dependent_k(i)$ is false, and
- P_k has not replied to all requests P_j ,

— sending

— replying

Who the galo to p ndant

then

begin

$\text{dependent}_k(i) = \text{true};$

if $k=i$

then declare that P_i is deadlocked
else for all P_m and P_n such that

(a') P_k 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 a probe (i, m, n) to the home site
of P_n

end.

$k \leftarrow i$

you are dependent on me


$k \rightarrow m \rightarrow n$

1 mr

✓

1 mr

$k \rightarrow m \rightarrow n$

- 
- A probe message is continuously circulated along the edges of the global WFG graph and a deadlock is detected when a probe message returns to its initiating process.

Performance Analysis

- One probe message (per deadlock detection initiation) is sent on every edge of the WFG which that two sites.
- Thus, the algorithm exchanges at most $m(n - 1)/2$ messages to detect a deadlock that involves m processes and that spans over n sites.
- The size of messages is fixed and is very small (only 3 integer words). (i, j, k)
- Delay in detecting a deadlock is $O(n)$.

Chandy-Misra-Haas Algorithm for the OR Model

Chandy-Misra-Haas distributed deadlock detection algorithm for OR model is based on the approach of diffusion-computation.

- A blocked process determines if it is deadlocked by initiating a diffusion computation.
- Two types of messages are used in a diffusion computation:
- $query(i, j, k)$ and $reply(i, j, k)$, denoting that they belong to a diffusion computation initiated by a process P_i and are being sent from process P_j to process P_k .

- 1. A blocked process initiates deadlock detection by sending query messages to all processes in its dependent set.
- 2. If an active process receives a query or reply message, it discards it.
- 3. When a blocked process P_k receives a query(i, j, k) message, it takes the following actions:
 - 1. If this is the first query message received by P_k for the deadlock detection initiated by P_i (called the *engaging query*), then it propagates the query to all the processes in its dependent set and sets a local variable $num_k(i)$ to the number of query messages sent.
 - 2. If this is not the *engaging query*, then P_k returns a reply message to it immediately provided P_k has been continuously blocked since it received the corresponding *engaging query*. Otherwise, it discards the query.

Not first query = not engaging query

- Process P_k maintains a boolean variable $wait_k(i)$ that denotes the fact that it has been continuously blocked since it received the last engaging query from process P_i .
- When a blocked process P_k receives a $reply(i, j, k)$ message, it decrements $num_k(i)$ only if $wait_k(i)$ holds.
- A process sends a reply message in response to an engaging query only after it has received a reply to every query message it had sent out for this engaging query.
- The initiator process detects a deadlock when it receives reply messages to all the query messages it had sent out.

Algorithm

The algorithm works as follows:

Initiate a diffusion computation for a blocked process P_i :

send query(i, i, j) to all processes P_j in the dependent

DS_i of P_i ;

$num_i(i) := |DS_i|$; $wait_i(i) := true$;

DS = Dependent Set

When a blocked process P_k receives a query(i, j, k):

if this is the engaging query for process P_i

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

$num_k(i) := num_k(i) - 1$; // decrement

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.

non-active

it's initiator

- In practice, several diffusion computations may be initiated for a process (A diffusion computation is initiated every time the process gets blocked), but, at any time only one diffusion computation is current for any process.
- However, messages for outdated diffusion computations may still be in transit
- The current diffusion computation can be distinguished from outdated ones by using sequence numbers.

Performance Analysis

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

Kshemkalyani-Singhal Algorithm for P-out-of-Q Model

- Kshemkalyani-Singhal algorithm detects deadlocks in the P-out-of-Q model is based on the global state detection approach.
- It is a single phase algorithm, which consists of a fan-out sweep of messages outwards from an initiator process and a fan-in sweep of messages inwards to the initiator process.
- A sweep is a traversal of the WFG in which all messages are sent in the direction of the WFG edges (outward sweep) or all messages are sent against the direction of the WFG edges (inward sweep).

- In the outward sweep, the algorithm records a snapshot of a distributed WFG.
- In the inward sweep, the recorded distributed WFG is reduced to determine if the initiator is deadlocked.
- Both the outward and the inward sweeps are executed concurrently in the algorithm.
- Complications are introduced because the two sweeps can overlap in time at a process, i.e., the reduction of the WFG at a process can begin before the WFG at that process has been completely recorded.

Reason of complication.

System Model

11/11/17

- The system has n nodes, and every pair of nodes is connected by a logical channel.
- Events are assigned timestamps using Lamport's clocks.
- The computation messages can be either REQUEST, REPLY or CANCEL messages.
- To execute a p-out-of-q request, an active node i sends REQUESTs to other nodes and remains blocked until it receives sufficient number of REPLY messages.

1/1/17

- When node i blocks on node j , node j becomes a successor of node i and node i becomes a predecessor of node j in the WFG.
- A **REPLY** message denotes the granting of a request.
- A node i unblocks when p out of its q requests have been granted.
- When a node unblocks, it sends **CANCEL** messages to withdraw the remaining $q - p$ requests it had sent.
- Sending and receiving of **REQUEST**, **REPLY**, and **CANCEL** messages are *computation events*.
- The sending and receiving of deadlock detection algorithm messages are *algorithmic or control events*.

Data Structures

A node i has the following local variables:

$wait_i$: boolean ($:= false$); /*records the current status.*/

t_i : integer ($:= 0$); /*denotes the current time.*/

t_block_i : real; /*denotes the local time when i blocked last.*/

$in(i)$: set of nodes whose requests are outstanding at node i .

$out(i)$: set of nodes on which node i is waiting.

p_i : integer ($:= 0$); /*the number of replies required for unblocking.*/

w_i : real ($:= 1.0$); /*keeps weight to detect the termination of the algorithm.*/

Computation Events

REQUEST_SEND(i)

*/*Executed by node i when it blocks on a p-out-of-q request.*/*

For every node j on which i is blocked do

$out(i) \leftarrow out(i) \cup \{j\};$

send REQUEST(i) to j ;

set p_i to the number of replies needed;

$t_block_i := t_i;$

$wait_i \leftarrow true;$

✓/! waiting.

REQUEST_RECEIVE(j)

*/*Executed by node i when it receives a request made by j .*/*

$in(i) \leftarrow in(i) \cup \{j\};$

REPLY_SEND(j)

*/*Executed by node i when it replies to a request by j .*/*

$in(i) \leftarrow in(i) - \{j\};$

send REPLY(i) to j .

REPLY_RECEIVE(*j*)

*/*Executed by node *i* when it receives a reply from *j* to its request.*/*

if valid reply for the current request

then begin

$out(i) \leftarrow out(i) - \{j\};$

$p_i \leftarrow p_i - 1;$

$p_i = 0 \rightarrow$

$\{wait_i \leftarrow false;$

$\forall k \in out(i), \text{ send CANCEL}(i) \text{ to } k;$

$out(i) \leftarrow \emptyset.\}$

end

CANCEL_RECEIVE(*j*)

*/*Executed by node *i* when it receives a cancel from *j*.*/*

if $j \in in(i)$ then $in(i) \leftarrow in(i) - \{j\}.$

Informal Description of the Algorithm

- When a node *init* blocks on a P -out-of- Q request, it initiates the deadlock detection algorithm.
- The algorithm records part a of the WFG that is reachable from *init* (henceforth, called the *init*'s WFG) in a distributed snapshot.
- The distributed WFG is recorded using FLOOD messages in the outward sweep and recorded WFG is examined for deadlocks using ECHO messages in the inward sweep.

1)
2)
3)

- 1) To detect a deadlock, the initiator *init* records its local state and sends FLOOD messages along all of its outward dependencies.
- 2) When node *i* receives the first FLOOD message along an existing inward dependency, it records its local state.
- 3) If node *i* is blocked at this time, it sends out FLOOD messages along all of its outward dependencies to continue the recording of the WFG in the outward sweep.
- 4) If node *i* is active at this time, then it initiates reduction of the WFG by returning an ECHO message along the incoming dependency even before the states of all incoming dependencies have been recorded in the WFG snapshot at the leaf node.

1 hr

- ECHO messages perform reduction of the recorded WFG by simulating the granting of requests in the inward sweep.
- A node i in the WFG is reduced if it receives ECHOs along p_i out of its q_i outgoing edges indicating that p_i of its requests can be granted.
- An edge is reduced if an ECHO is received on the edge indicating that the request it represents can be granted.
- The nodes that can be reduced do not form a deadlock whereas the nodes that cannot be reduced are deadlocked.
- Node $init$ detects the deadlock if it is not reduced when the deadlock detection algorithm terminates.

1 hr

The Problem of Termination Detection

- The algorithm requires a termination detection technique so that the initiator can determine that it will not receive any more ECHO messages.
- The algorithm uses a termination detection technique based on weights in conjunction with SHORT messages to detect the termination of the algorithm.
- A weight of 1.0 at the initiator node, when the algorithm is initiated, is distributed among all FLOOD messages sent out by the initiator.
- When the first FLOOD is received at a non-leaf node, the weight of the received FLOOD is distributed among the FLOODs sent out along outward edges at that node to expand the WFG further.

1m

- Since any subsequent FLOOD arriving at a non-leaf node does not expand the WFG further, its weight is returned to the initiator in a SHORT message.
- When a FLOOD is received at a leaf node, its weight is piggybacked to the ECHO sent by the leaf node to reduce the WFG.
- When an ECHO that arrives at a node unblocks the node, the weight of the ECHO is distributed among the ECHOs that are sent by that node along the incoming edges in its WFG snapshot.
- When an ECHO arriving at a node does not unblock the node, its weight is sent directly to the initiator in a SHORT message.

The following invariant holds in an execution of the algorithm:

- the sum of the weights in FLOOD, ECHO, and SHORT messages plus the weight at the initiator (received in SHORT and ECHO messages) is always 1.0.
- The algorithm terminates when the weight at the initiator becomes 1.0, signifying that all WFG recording and reduction activity has completed.
- FLOOD, ECHO, and SHORT messages carry weights for termination detection. Variable w , a real number in the range $[0, 1]$, denotes the weight in a message.

The Algorithm

- A node i stores the local snapshot for snapshots *initiated* by other nodes in a data structure LS_i (Local Snapshot), which is an array of records.

LS_i : array $[1..n]$ of record;

- A record has several fields to record snapshot related information and is defined below for an initiator *init*:

$LS_i[init].out$: set of integers ($:= \emptyset$); /*nodes on which i is waiting in the snapshot.*/

$LS_i[init].in$: set of integers ($:= \emptyset$); /*nodes waiting on i in the snapshot.*/

$LS_i[init].t$: integer ($:= 0$); /*time when *init* initiated snapshot.*/

$LS_i[init].s$: boolean ($:= false$); /*local blocked state as seen by snapshot.*/

$LS_i[init].p$: integer; /*value of p_i as seen in snapshot.*/

The deadlock detection algorithm is defined by the following procedures. The procedures are executed atomically.

SNAPSHOT_INITIATE

/* Executed by node i to detect whether it is deadlocked. */

$init \leftarrow i;$

$w_i \leftarrow 0;$

$LS_i[init].t \leftarrow t_i;$

$LS_i[init].out \leftarrow out(i);$

$LS_i[init].s \leftarrow true;$

$LS_i[init].in \leftarrow \emptyset;$

$LS_i[init].p \leftarrow p_i;$

send $FLOOD(i, i, t_i, 1/|out(i)|)$ to each j in $out(i)$. /*

$1/|out(i)|$ is the fraction of weight sent in a FLOOD message. */

FLOOD_RECEIVE($j, init, t_init, w$)

*/*Executed by node i on receiving a FLOOD message from j . */*

*LS _{i} [$init$]. $t < t_init \wedge j \in in(i) \rightarrow$ */*Valid FLOOD for a new snapshot.**
**/*

LS _{i} [$init$]. $out \leftarrow out(i)$;

LS _{i} [$init$]. $in \leftarrow \{j\}$;

LS _{i} [$init$]. $t \leftarrow t_init$;

LS _{i} [$init$]. $s \leftarrow wait_i$;

*wait _{i} = true \rightarrow */* Node is blocked. */**

LS _{i} [$init$]. $p \leftarrow p_i$;

***send** FLOOD($i, init, t_init, w/|out(i)|$) to each $k \in out(i)$;*

*wait _{i} = false \rightarrow */* Node is active. */**

LS _{i} [$init$]. $p \leftarrow 0$;

***send** ECHO($i, init, t_init, w$) to j ;*

LS _{i} [$init$]. $in \leftarrow LS_i[init].in - \{j\}$.



$LS_i[init].t < t_{init} \wedge j \notin in(i) \rightarrow$ /* Invalid FLOOD for a new snapshot. */
send *ECHO*(*i*, *init*, *t_init*, *w*) to *j*.

□

$LS_i[init].t = t_{init} \wedge j \notin in(i) \rightarrow$ /* Invalid FLOOD for current snapshot. */
send *ECHO*(*i*, *init*, *t_init*, *w*) to *j*.

□

$LS_i[init].t = t_{init} \wedge j \in in(i) \rightarrow$ /*Valid FLOOD for current snapshot. */
 $LS_i[init].s = false \rightarrow$
send *ECHO*(*i*, *init*, *t_init*, *w*) to *j*;
 $LS_i[init].s = true \rightarrow$
 $LS_i[init].in \leftarrow LS_i[init].in \cup \{j\};$
send *SHORT*(*init*, *t_init*, *w*) to *init*.

$LS_i[init].t > t_{init} \rightarrow$ discard the FLOOD message. /*Out-dated FLOOD. */

ECHO_RECEIVE($j, init, t_{init}, w$)

*/*Executed by node i on receiving an ECHO from j . */*

*/*Echo for out-dated snapshot. */*

$LS_i[init].t > t_{init} \rightarrow$ discard the ECHO message.

$LS_i[init].t < t_{init} \rightarrow$ cannot happen. */*ECHO for unseen snapshot. */*

$LS_i[init].t = t_{init} \rightarrow$ */*ECHO for current snapshot. */*

$LS_i[init].out \leftarrow LS_i[init].out - \{j\};$

$LS_i[init].s = false \rightarrow$ **send** *SHORT*($init, t_{init}, w$) to $init$.

$LS_i[init].s = true \rightarrow$

$LS_i[init].p \leftarrow LS_i[init].p - 1;$

$LS_i[init].p = 0 \rightarrow$ */* getting reduced */*

$LS_i[init].s \leftarrow false;$

$init = i \rightarrow$ declare not deadlocked; exit. •

send *ECHO*($i, init, t_{init}, w / |LS_i[init].in|$) to all

$k \in LS_i[init].in;$

$LS_i[init].p \neq 0 \rightarrow$ •

send *SHORT*($init, t_{init}, w$) to $init$.

SHORT_RECEIVE(*init*, t_{init} , *w*)

*/**Executed by node *i* (which is always *init*) on receiving a SHORT. **/*

[
*/**SHORT for out-dated snapshot. **/*
 $t_{init} < t_{block_i} \rightarrow$ discard the message.

□

*/**SHORT for uninitiated snapshot. **/*
 $t_{init} > t_{block_i} \rightarrow$ not possible.

□

*/**SHORT for currently initiated snapshot. **/*
 $t_{init} = t_{block_i} \wedge LS_i[init].s = false \rightarrow$ discard. */* init is*
active. **/*

$t_{init} = t_{block_i} \wedge LS_i[init].s = true \rightarrow$
 $w_i \leftarrow w_i + w;$
 $w_i = 1 \rightarrow$ **declare a deadlock.**

]

An Example

V. V. 2.1

- We now illustrate the operation of the algorithm with the help of an example shown in Figures 3 and 4.
- Figure 3 shows initiation of deadlock detection by node A and Figure 4 shows the state after node D is reduced.
- The notation x/y beside a node in the figures indicates that the node is blocked and needs replies to x out of the y outstanding requests to unblock.

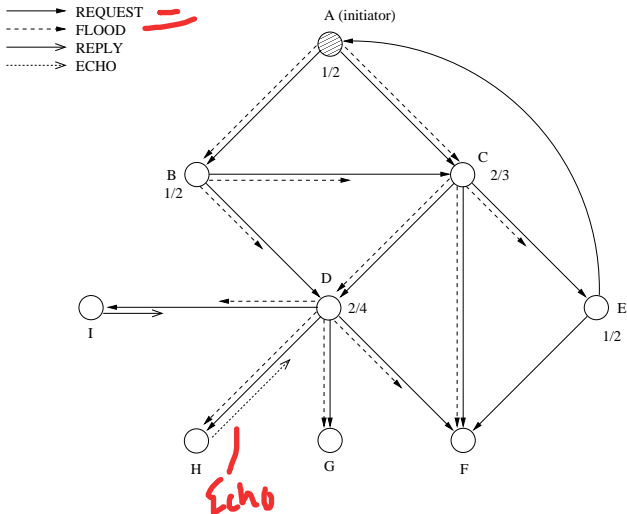


Figure 3: An Example-run of the Algorithm.

- In Figure 3, node A sends out FLOOD messages to nodes B and C. When node C receives FLOOD from node A, it sends FLOODs to nodes D, E, and F.
- If the node happens to be active when it receives a FLOOD message, it initiates reduction of the incoming wait-for edge by returning an ECHO message on it.
- For example, in Figure 3, node H returns an ECHO to node D in response to a FLOOD from it.
- Note that node can initiate reduction even before the states of all other incoming wait-for edges have been recorded in the WFG snapshot at that node.

depends on node

- For example, node F in Figure 3 starts reduction after receiving a FLOOD from C even before it has received FLOODs from D and E.
- Note that when a node receives a FLOOD, it need not have an incoming wait-for edge from the node that sent the FLOOD because it may have already sent back a REPLY to the node.
- In this case, the node returns an ECHO in response to the FLOOD.
- For example, in Figure 3, when node I receives a FLOOD from node D, it returns an ECHO to node D.

1 m

- ✓ ECHO messages perform reduction of the nodes and edges in the WFG by simulating the granting of requests in the inward sweep.
- A node that is waiting a p -out-of- q request, gets reduced after it has received p ECHOs.
- When a node is reduced, it sends ECHOs along all the incoming wait-for edges incident on it in the WFG snapshot to continue the progress of the inward sweep.
- ✓ In general, WFG reduction can begin at a non-leaf node before recording of the WFG has been completed at that node.

1 m

✓ This happens when ECHOs arrive and begin reduction at a non-leaf node before FLOODs have arrived along all incoming wait-for edges and recorded the complete local WFG at that node.

• For example, node D in Figure 3 starts reduction (by sending an ECHO to node C) after it receives ECHOs from H and G, even before FLOOD from B has arrived at D.

inp ✓ When a FLOOD on an incoming wait-for edge arrives at a node which is already reduced, the node simply returns an ECHO along that wait-for edge.

✓ • For example, in Figure 4, when a FLOOD from node B arrives at node D, node D returns an ECHO to B.

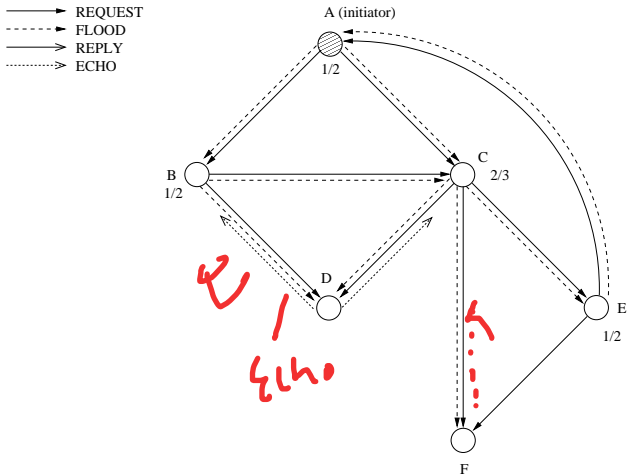


Figure 4: An Example-run of the Algorithm (continued).

- In Figure 3, node C receives a FLOOD from node A followed by a FLOOD from node B.
- When node C receives a FLOOD from B, it sends a SHORT to the initiator node A.
- When a FLOOD is received at a leaf node, its weight is returned in the ECHO message sent by the leaf node to the sender of the FLOOD.
- Note that an ECHO is like a reply in the simulated unblocking of processes.
- When an ECHO arriving at a node does not reduce the node, its weight is sent directly to the initiator through a SHORT message.

ΣΙΤΟΥΚΟ minimum donkey, 147

- For example, in Figure 3, when node D receives an ECHO from node H, it sends a SHORT to the initiator node A.
- When an ECHO that arrives at a node reduces that node, the weight of the ECHO is distributed among the ECHOs that are sent by that node along the incoming edges in its WFG snapshot.
- For example, in Figure 4, at the time node C gets reduced (after receiving ECHOs from nodes D and F), it sends ECHOs to nodes A and B. (When node A receives an ECHO from node C, it is reduced and it declares no deadlock.)
- When an ECHO arrives at a reduced node, its weight is sent directly to the initiator through a SHORT message.

- ✓ 1. For example, in Figure 4, when an ECHO from node E arrives at node C after node C has been reduced (by receiving ECHOs from nodes D and F), node C sends a SHORT to initiator node A. ✓ IMW

Correctness

Proving the correctness of the algorithm involves showing that it satisfies the following conditions:

1. The execution of the algorithm terminates.
2. The entire WFG reachable from the initiator is recorded in a consistent distributed snapshot in the outward sweep. ✓
3. In the inward sweep, ECHO messages correctly reduce the recorded snapshot of the WFG. IMW ✓

- ✓ The algorithm is initiated within a timeout period after a node blocks on a P-out-of-Q request. clock
- ✓ On the termination of the algorithm, only all the nodes that are not reduced, are deadlocked. V.V.I. P

Complexity Analysis

- The algorithm has a message complexity of $4e - 2n + 2l$ and a time complexity¹ of $2d$ hops, where e is the number of edges, n the number of nodes, l the number of leaf nodes, and d the diameter of the WFG.
- This gives the best time complexity that can be achieved by an algorithm that reduces a distributed WFG to detect generalized deadlocks in distributed systems.

¹Time complexity denotes the delay in detecting a deadlock after its detection has been initiated.