Global State Deadlock Detection Algorithms

Distributed Systems

Project Report

Submitted By-

Sarvat Ali, 2018201010

Priya Upadhyay, 2018202012

Himani Gupta, 2018202014

1. Introduction

Global state detection-based deadlock detection algorithms exploit the following facts: (i) a consistent snapshot of a distributed system can be obtained without freezing the underlying computation, and (ii) 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. Examples of these types of algorithms include the Bracha–Toueg , Kshemkalyani–Singhal and Wang et al. We will be discussing Bracha–Toueg and Kshemkalyani–Singhal algorithms in this report.

2. Bracha–Toueg Algorithm

Bracha-Toueg algorithm is a **two phase algorithm**, first phase is the *notification* phase, in which processes/nodes are notified that deadlock detection algorithm has started, and a *granting* phase, in which free processes simulate granting of requests. Deadlocked nodes are the nodes that are not made free by executing the *granting* phase.

Bracha-Toueg algorithm is initiated by a process that suspects it is deadlocked, starts a Lai-Yang snapshot to compute the wait-for graph. It performs static analysis on the wait-for-graph to detect deadlock. The wait-for-graph captures dependencies between processes(sent/received requests).

Each node *u* in the wait-for-graph is a process which can take a local snapshot, to determine the sent/received requests that are yet to be fulfilled. it maintains two lists Out_u and In_u , and a $requests_u$ variable to store the count of grants it requires to become unblocked.

Out_u= {*v*: *u* has sent a request to *v* which is not yet granted}

In_u = {*v*: *u* has received a request from *v* which is not yet granted}

$requests_u$ = number of requests *u* has made to other processes

Whenever a request is granted to *u* from a process in Out_u , $requests_u$ is decremented by 1. If $requests_u$ becomes 0 i.e, the process at node *u* has no pending requests to other processes, and it can grant requests of processes in In_u. If at the end of the algorithm $request_u > 0$ then the process at node *u* is deadlocked.

We need to know when our algorithm will end, Bracha-Toueg algorithm is designed in such a way that there is no need for another termination detection algorithm. We will understand that as we go through the algorithm:

Each node can perform two functions: $Notify_u()$ and $Grant_u()$.

Initially, $notified_u$ = false and $free_u$ = false at all nodes, these two ensure that $Notify_u()$ and $Grant_u()$ are executed at most once.

Nodes can send each other 4 types of messages:

<notify> : *u* sends this message to all the processes in Out_u .

<grant> : when *u* becomes unblocked, it sends this message to all the processes in In_u.

<done> : *u* sends it to its parent(which sent <notify> to it), when it's $Notify_u()$ is finished

<ack> : as an acknowledgement of <grant>, *u* sends it to the process which sent <grant> to it.

The pseudocode for $Notify_u()$:

```
$notified_u \leftarrow true$;
send ⟨**notify**⟩ to all $w \in Out_u$;
**if** $requests_u = 0$ **then**
    perform procedure $Grant_u$;
**end if**
await ⟨**done**⟩ from all $w \in Out_u$;
```

And the pseudocode for $Grant_u()$:

```
$free_u \leftarrow true$;
send ⟨**grant**⟩ to all $w \in In_u$;
await ⟨**ack**⟩ from all $w \in In_u$;
```

u On receiving **<notify>** from *v* does the following:

```
**if** $notified_u = false$ **then**
    perform procedure $Notify_u$;
**end if**
send ⟨**done**⟩ to $v$;
```

On receiving **<grant>** it does the following:

```
if $requests_u > 0$ then
    $requests_u \leftarrow requests_u - 1$;
    if $requests_u = 0$ then
        perform procedure $Grant_u$;
    end if
end if
send $\langle$ack$\rangle$ to $v$;
```

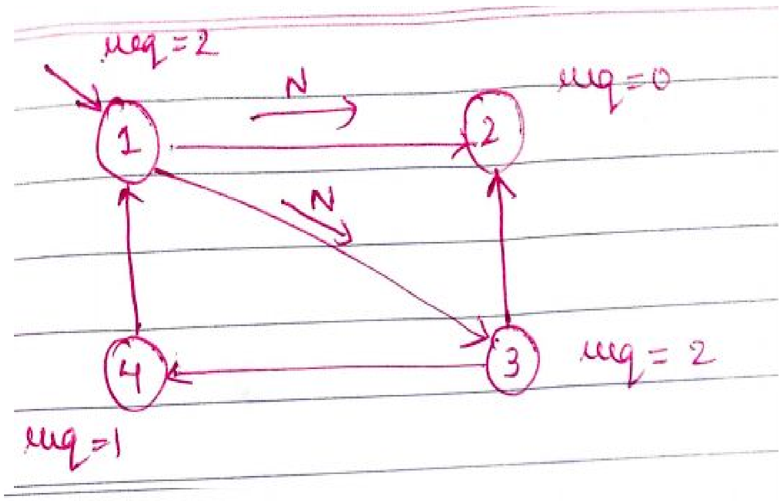
Some noteworthy points:

- When u receives **<notify>** from v it executes *Notify*$_u$*()* and if *notified*=true, i.e, some other node in the graph notified it already, therefore it sends **<done>** to v.
- When u receives **<grant>** from v it executes *Grant*$_u$*()* and if *requests*$_u$ does not become zero, u immediately sends **<ack>** to v.
- While awaiting **<done>**, **<ack>** messages, u can process **<notify>** and **<grant>** messages.
- If initiator receives **<done>** from all the processes in *Out*$_u$ it means it's *Notify*$_u$*()* has completed execution i.e, algorithm can be terminated after checking whether initiator node variable *free* whether it is true or false, if false it is concluded that initiator is deadlocked.

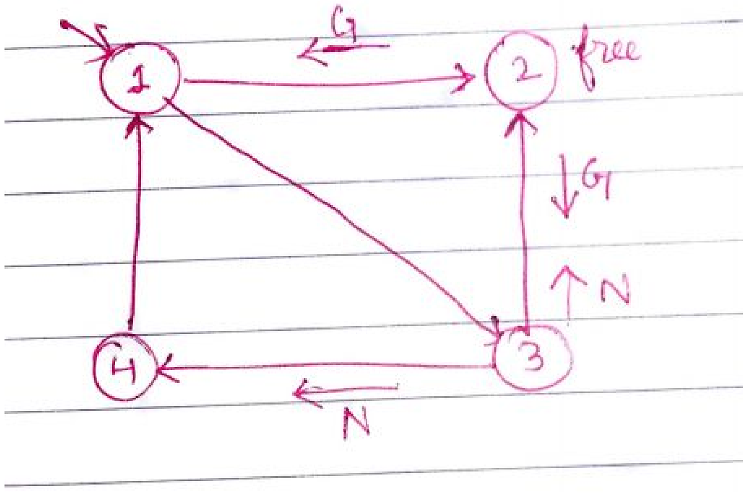
It can be argued that even if initiator u has received **<done>** from all nodes, some messages may still be in transit i.e, the algorithm should not terminate solely on this condition. On analyzing the trees formed by messages sent/received, we figure that initiator u will receive **<done>** from other nodes after they have finished sending/receiving messages from/to other nodes. Two types of trees are observed, one is tree T rooted at initiator node, formed by **<notify>**/**<done>** messages, if node w received first **<notify>**

from node *v*, then *v* is parent of *w*, and another is tree T_v rooted at each unblocked process *v*, formed by **<grant>/<ack>** messages, if node *w* receives **<grant>** from node *v*, then *v* is parent of *w*. A non-initiator *v* that is the root of a tree T_v only sends a done to its parent in T when all **<grant>** sent by nodes in T_v have been acknowledged. This implies that when the initiator completes its *Notify*$_u$*()* call, not only all **<notify>**'s but also all **<grant>**'s in the network have been acknowledged. So at that moment there are no messages in transit or pending messages.

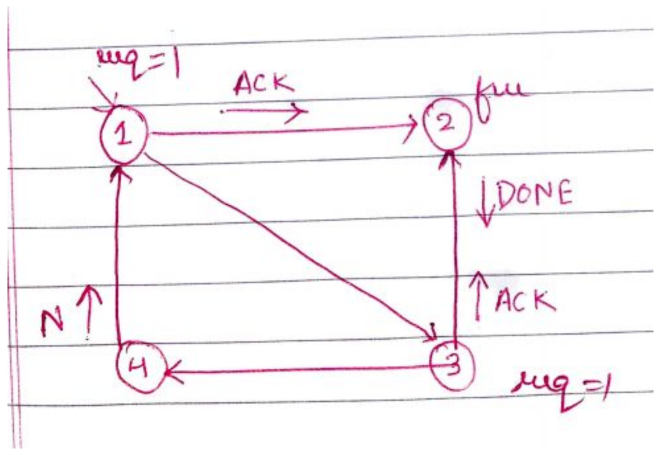
Take following graph for example-



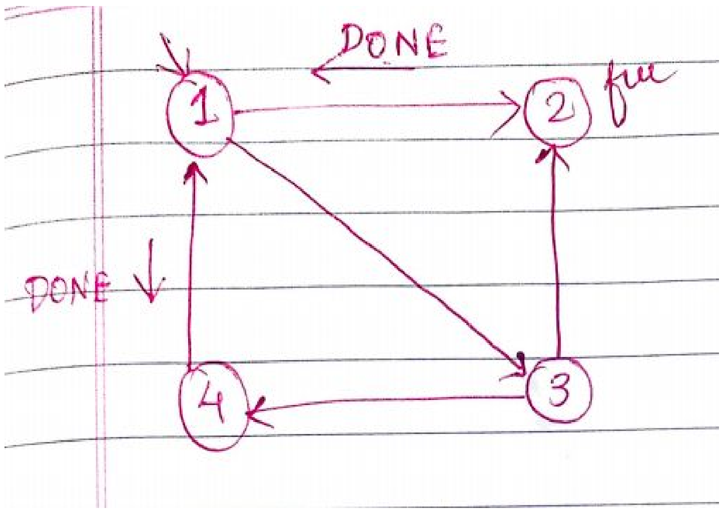
Here 1 is the initiator. So it starts by executing Notify() and sends <notify> messages to its outgoing edges,i.e, 2 and 3.



Node 2 upon receiving <notify> msg executes Notify() and finds that its requests=0, there it executes Grant(), it sets free = True and sends <grant> messages to node 1 and node 3. Node 3 upon receiving <notify> executes Notify() and sends <notify> messages to node 2 and node 4.

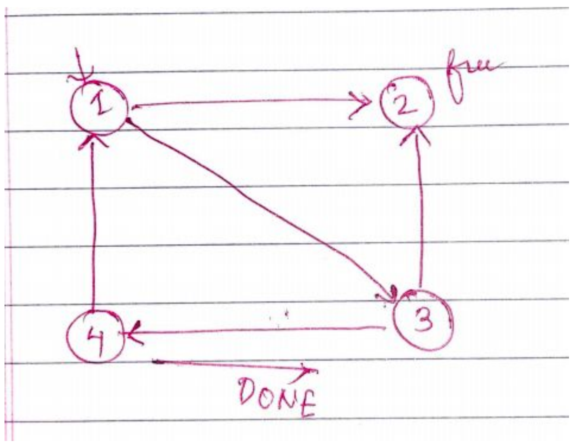


Node 1 received a <grant> message from node 2, request is decremented by 1 and is 1 now, <ack> sent back to node 2. Node 3 also decrements its request which now becomes 1, <ack> sent back to node 2. Node 2 received a <notify> from node 3, but since it has already executed Notify() i.e, notified = True, it sends <done> back to node 3. Node 4 received <notify> from node 3, executes Notify() and sends <notify> to outgoing edge to node 1.

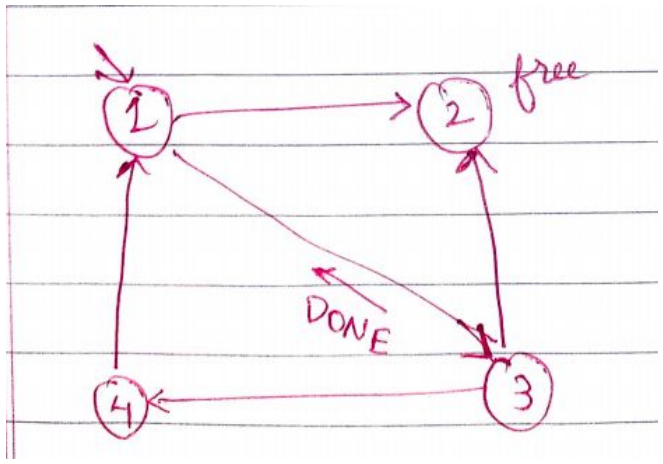


Node 2 completes its execution of Notify() and sends back <done> to node 1

Node 1 received <notify> from node 4, but since its notified=True, it sends <done> to 4.

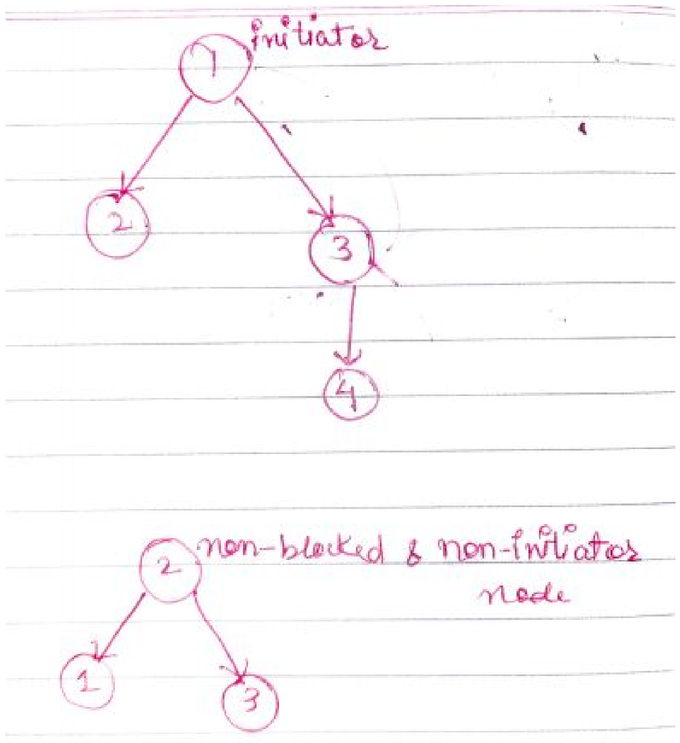


When node 4 recives <done> from node 1 its execution of Notify() is complete and it sends back <done>to 3.



When node 3 recives <done> from node 4 its execution of Notify() is complete and it sends back <done>to 1. By this time initiator 1 has received <done> from both the nodes in its Out set. It checks free variable, which is still false, because it never got a chance to execute the *granting* phase, ie., Node 1 is deadlocked!

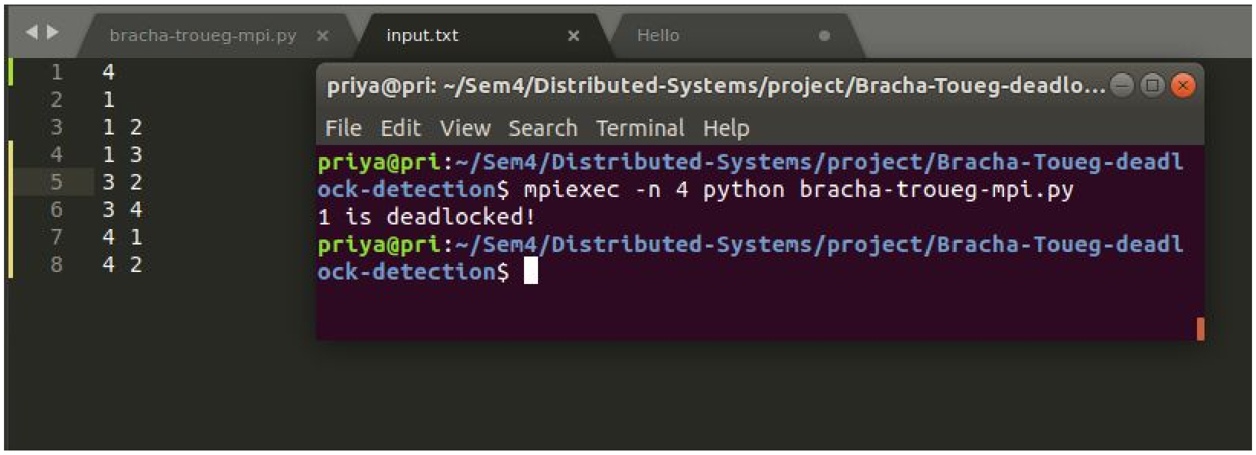
Trees for the above example



-First tree is notification phase tree

-Second tree is granting phase tree

Output for the above graph from our code:



```
4
1
1 2
1 3
3 2
3 4
4 1
4 2
```

```
priya@pri:~/Sem4/Distributed-Systems/project/Bracha-Toueg-deadl
ock-detection$ mpiexec -n 4 python bracha-troueg-mpi.py
1 is deadlocked!
priya@pri:~/Sem4/Distributed-Systems/project/Bracha-Toueg-deadl
ock-detection$
```

Complexity Analysis:

Message Complexity: 4 messages per edge in the WFG, <notify>, <grant>, <done>, <ack>.

Time Complexity: 4d hops, where d = diameter of the WFG.

2d to send <notify>/<grant> to the farthest node in the graph.

Another 2d to receive <done>/<ack> from the same.

3. Kshemkalyani-Singhal algorithm

This algorithm consists of a single phase. The phase comprises diffusion of FLOOD messages outward from the initiator process along the edges(outward sweep) of WFG and then diffusion of ECHO messages inward to the initiator process(inward sweep). In the outward sweep, the algorithm maintains a consistent snapshot of distributed WFG. In inward sweep, a reduction procedure that simulates unblocking of the processes whose requests can be granted is applied to the recorded distributed WFG to determine whether a deadlock exists. If the requests on which a process is blocked can be granted, then that node in the WFG can be reduced. All the processes that cannot be reduced after the reduction procedure are deadlocked. Also, termination detection algorithm is used to detect the end of the algorithm.

Distributed WFG is recorded using FLOOD messages during the outward sweep and is examined for deadlocks using ECHO messages during the inward sweep. The initiator records its local state and sends FLOOD messages along its outgoing wait-for edges at the time it blocks.
 At the time a node *i* receives the first FLOOD message along an existing incoming wait-for edge, it records its local state (out_i, p_i, t_block_i, and this particular incoming wait-for edge). If the node happens to be blocked at this time, it sends FLOOD's along its outgoing wait-for edges to ensure that all nodes in the teachability set of the initiator participate in recording the WFG in the outward sweep.

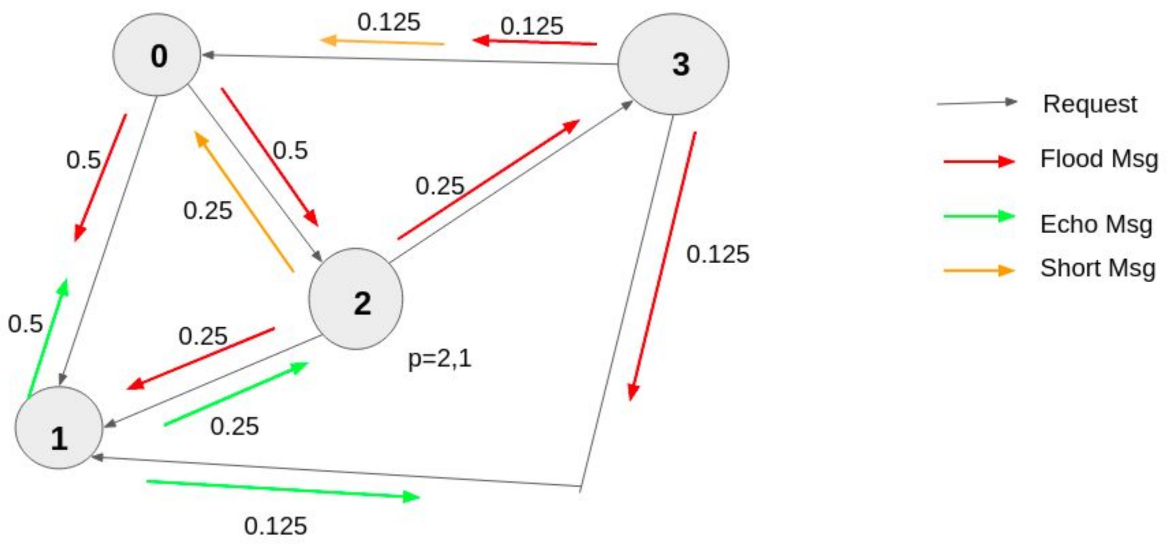


Fig 1.

For example, in Fig. 1, when node **2** receives FLOOD from node **0**, it sends FLOOD's to nodes **1** and **3**. If the node happens to be active at this time, (i.e., it does not have any outgoing wait-for edges). then it initiates reduction of the incoming wait-for edge by returning an ECHO message on it. For example, node **1** returns an ECHO to node **2** in response to a FLOOD from it. Note that such an active node can initiate Induction (by sending back an ECHO in response to a FLOOD along an incoming wait-for edge) even before the states of all other incoming wait-for edges have been recorded in the WFG snapshot at that node. For example, node **1** in Fig. 1 starts reduction after receiving a FLOOD from **0** even before it has received FLOOD's from **2** and **3**.

ECHO messages perform reduction of the nodes and edges in the WFG by simulating the granting of requests in the inward sweep. A node gets reduced at the time it has received *p* ECHO's. When a node is reduced, it sends ECHO's along all the incoming wait-for edges incident on it in the WFG snapshot to continue the progress of the inward sweep. These ECHO's in turn may reduce other nodes. The initiator node detects a deadlock if it is not reduced when the deadlock detection algorithm terminates. The nodes in the WFG snapshot that have not been reduced are deadlocked. The order in which reduction of the nodes and edges of the WFG is performed does not alter the final result.
In general, WFG reduction can begin at a nonleaf node before recording of the WFG has been completed at that node. This happens when ECHO's arrive and begin reduction at a nonleaf node before FLOOD's have arrived along all incoming wait-for edges and recorded the complete local WFG at that node. For example. node **1** in Fig. I starts reduction (by sending an ECHO to node **0**), even before FLOOD from **2** has arrived at **1**. When a FLOOD on an incoming wait-for edge arrives at a node that is already reduced, the node simply returns an ECHO along that wait-for edge. For example. in Fig. 1 when a FLOOD from node **2** arrives at node **1**, node D returns an ECHO to **2**. Thus, the two activities of recording the WFG snapshot and reducing the nodes and edges in the WFG snapshot are done

concurrently in a single phase, and no serialization is imposed between the two activitie. Since reduction is applied to an incompletely recorded WFG at nodes, the local snapshot at each node has to be judiciously manipulated to give the effect that reduction is initiated after WFG recording has been completed. The reduction operations at any node can be permuted with the WFG recording operations that follow them at that node, without affecting the result.

Termination Detection:

A termination detection technique based on weights detects the termination of the algorithm by using SHORT messages (in addition to FLOOD's and ECHO's). A weight of I 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 nonleaf node along an existing WFG edge, the weight of the received FLOOD is distributed among the FLOOD's sent along outgoing wait-for edges at that node. Weights in all subsequent FLOOD's arriving along existing WFG edges at a nonleaf node that is not yet reduced are returned to the initiator through SHORT messages. For example, in Fig. 1, node **3** receives a FLOOD from **2**, it sends a SHORT to the initiator node **0**.

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. For example. in Fig. I, when node **2** receives an ECHO from node **1**. it sends a SHORT to the initiator node **0**.

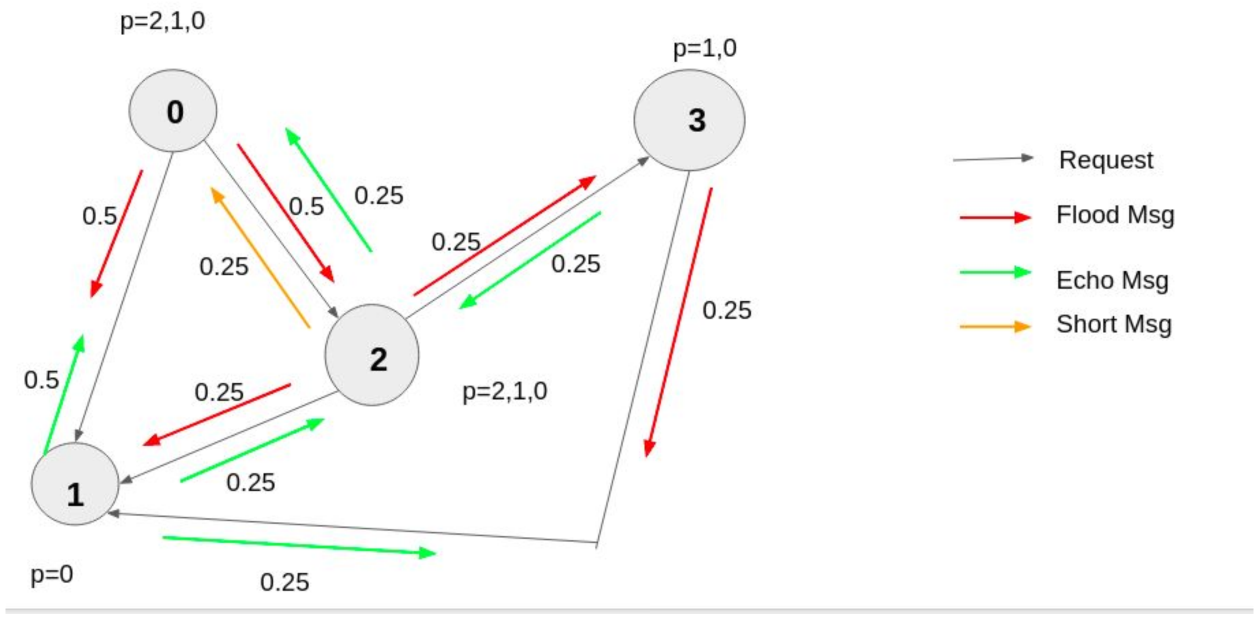


Fig 2.

When an ECHO that arrives at a node reduces that node, the weight of the ECHO is distributed among the ECHO's that are sent by that node along the incoming edges in its WFG snapshot. For example, in Fig. 2, at the time node **2** gets reduced (after receiving ECHO's from nodes **1** and **3**), it sends ECHO to node **0**.

When an ECHO arrives at a reduced node, its weight is sent directly to the initiator through a SHORT message. For example, in Fig. 2, when an ECHO from node **3** arrives at node **2** after node **2** has been reduced (by receiving ECHO's from node **1**), node **2** sends a SHORT to initiator node **0**.

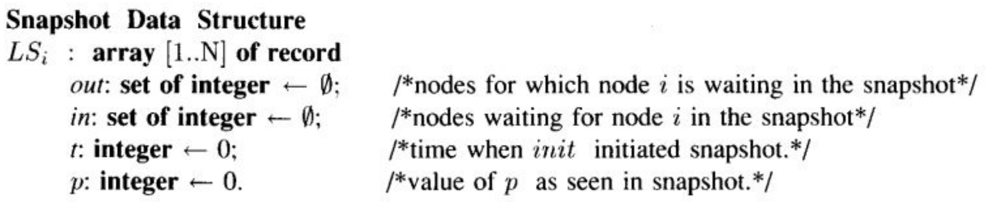
The algorithm maintains an invariant that the sum of the weights in FLOOD, ECHO, and SHORT messages plus the weight at the initiator (received in SHORT and ECHO messages) is always 1. The algorithm terminates when the weight at the initiator becomes 1, signifying that all WFG recording and reduction activity has completed.

The Algorithm:

FLOOD, ECHO, and SHORT control messages use weights for termination detection. The weight in a message is a real number from 0 to 1. An event at which a FLOOD, ECHO, or SHORT message is sent or received is a control event. A node *i* stores the local snapshot to detect deadlocks in a data structure LS_i , which is an array of records. Record $LS_i[init]$ stores a snapshot at node *i* corresponding to deadlock detection initiation by the initiator node *init*.In addition, the initiator has a variable in which it collects returned weights.

```
  $wt_{init}$: real ←1.0;                    /*weight to detect termination
                                          of deadlock detection algorithm.*/
```

For the sake of notational convenience, the initiator's identity is dropped when referring to LS_i , that is, $LS_i[init]$ is denoted by LS_i.



```
Snapshot Data Structure
$LS_i$ : array [1..N] of record
    out: set of integer ← ∅;     /*nodes for which node $i$ is waiting in the snapshot*/
    in: set of integer ← ∅;      /*nodes waiting for node $i$ in the snapshot*/
    t: integer ← 0;              /*time when $init$ initiated snapshot.*/
    p: integer ← 0.              /*value of $p$ as seen in snapshot.*/
```

```
initiate a SNAPSHOT
/*Executed by process $i$ to detect whether it is deadlocked. */
$init \leftarrow i$;
$wt_{init} \leftarrow 0$;
$LS_i.t \leftarrow t_i$;
$LS_i.out \leftarrow out_i$;
$LS_i.in \leftarrow \emptyset$;
$LS_i.p \leftarrow p_i$;
send $FLOOD(i, i, t_i, 1/|out_i|)$ to each $j$ in $out_i$.

receive FLOOD($k$, $init$, $t\_init$, $w$)
/*Executed by process $i$ on receiving a FLOOD message from $k$.*/
[
 /*Valid FLOOD for a new snapshot. */
$LS_i.t < t\_init \bigwedge k \in in_i \longrightarrow$
        $LS_i.out \leftarrow out_i$;
        $LS_i.in \leftarrow \{k\}$;
        $LS_i.t \leftarrow t\_init$;
        $LS_i.p \leftarrow p_i$;
        $p_i > 0 \longrightarrow$
            send FLOOD($i, init, t\_init, w/|out_i|$) to each $j \in out_i$;
        $p_i = 0 \longrightarrow$
            send $ECHO(i, init, t\_init, w)$ to $k$;
□
/* FLOOD for a new snapshot is received
along an edge that has ceased to exist. A reply
has already been sent back to the sender of the FLOOD.*/
$LS_i.t < t\_init \bigwedge k \notin in_i \longrightarrow$
        send $ECHO(i, init, t\_init, w)$ to $k$.
□
/* FLOOD for a current snapshot is received along
an edge that has ceased to exist. A reply has already been sent back to the
sender of the FLOOD.*/

$LS_i.t = t\_init \bigwedge k \notin in_i \longrightarrow$
    send $ECHO(i, init, t\_init, w)$ to $k$.
□
/*Valid FLOOD for current snapshot.*/
$LS_i.t = t\_init \bigwedge k \in in_i \longrightarrow$
    $LS_i.in \leftarrow LS_i.in \bigcup \{k\}$;
    $LS_i.p = 0 \longrightarrow$
      send $ECHO(i, init, t\_init, w)$ to $k$;
    $LS_i.p > 0 \longrightarrow$
      send $SHORT(init, t\_init, w)$ to $init$.
□
/*Outdated FLOOD. */

$LS_i.t > t\_init \longrightarrow$ discard the flood message.
]

receive ECHO($j$, $init$, $t\_init$, $w$)
/*Executed by process $i$ on receiving an ECHO from $j$ for a current snapshot
for which $LS_i.t = t\_init$. ECHO for an outdated snapshot ($LS_i.t > t\_init$)
is discarded. */
$LS_i.t = t\_init \longrightarrow$
    $LS_i.out \leftarrow LS_i.out - \{j\}$;
    $LS_i.p = 0 \longrightarrow$
```

```
**Algorithm (continued)**
        send $SHORT(init, t\_init, w)$ to $init$.
    $LS_i.p > 0 \longrightarrow$
      $LS_i.p \leftarrow LS_i.p - 1;$
      $LS_i.p = 0 \longrightarrow$
        $init = i \longrightarrow$ declare not deadlocked; exit.
        send $ECHO(i, init, t\_init, w/|LS_i.in|)$ to all $k \in LS_i.in$;
      $LS_i.p \neq 0 \longrightarrow$
        send $SHORT(init, t\_init, w)$ to $init$.

receive SHORT($init, t\_init, w$)
/*Executed by process $i$ (which is always $init$) on receiving a
SHORT for which $t\_init = t\_block_i$. SHORT for an outdated
snapshot ($t\_init < t\_block_i$) is discarded.*/
$t\_init = t\_block_i \bigwedge LS_i.p = 0 \longrightarrow$
        discard the message.
$t\_init = t\_block_i \bigwedge LS_i.p > 0 \longrightarrow$
    $wt_{init} \leftarrow wt_{init} + w;$
    $wt_{init} = 1 \longrightarrow$ declare deadlock and abort.
```

Performance:

Time Complexity:

- 2*diameter of WFG
- To record farthest edge in WFG it requires one diameter to traverse and another diameter is traversed during reduction of WFG

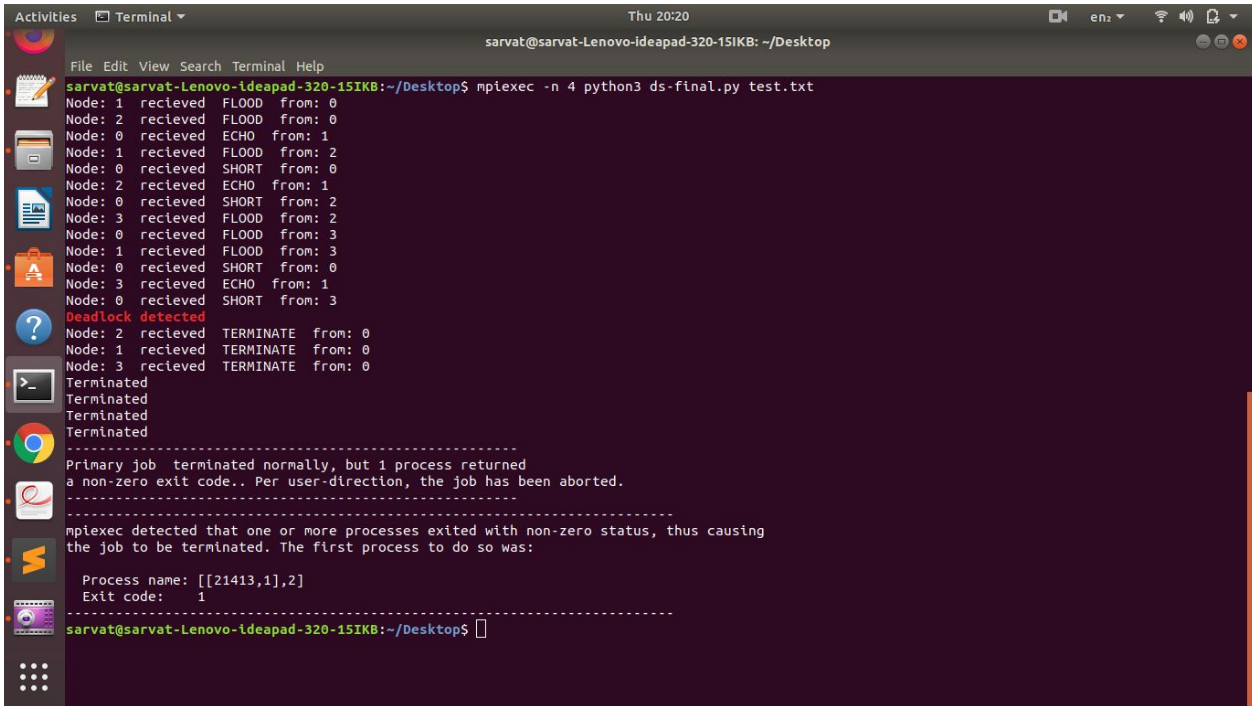
Message Complexity

- Flood messages = o(e)
- Echo messages= o(e)
- Flood Messages converted to Short Messages = o(e-n+l)-number of edges to leaf nodes(le)
- n-l as internal node uses Flood message to expand deadlock.
- e-(n-l) -le as Flood messages to leaf nodes(le) always returns Echo message.
- Echo Messages converted to Short Messages = o(e-n+l)
- e-(n-l) as when non leaf node reduces it returns Echo messages and not Short messages

- Hence total=o(4e-2n+l-le). Many messages are concurrent
- e is number of edges in WFG, l is number of leaf nodes(p=0),n is total number of nodes in system.le is number of edges to leaf nodes.

Output:

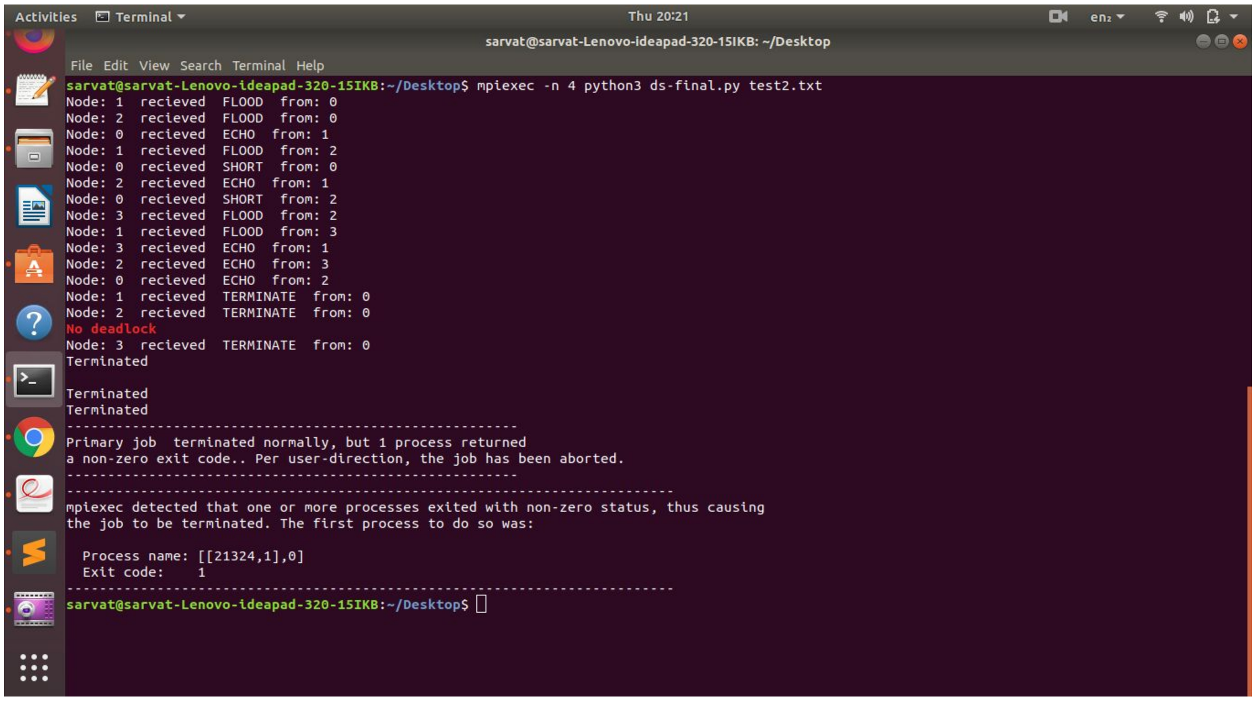
For Example 1(Fig 1), we received the following output:


Activities
Terminal
Thu 20:20
en
sarvat@sarvat-Lenovo-ideapad-320-15IKB: ~/Desktop
File Edit View Search Terminal Help

```
sarvat@sarvat-Lenovo-ideapad-320-15IKB:~/Desktop$ mpiexec -n 4 python3 ds-final.py test.txt
Node: 1  recieved  FLOOD  from: 0
Node: 2  recieved  FLOOD  from: 0
Node: 0  recieved  ECHO  from: 1
Node: 1  recieved  FLOOD  from: 2
Node: 0  recieved  SHORT  from: 0
Node: 2  recieved  ECHO  from: 1
Node: 0  recieved  SHORT  from: 2
Node: 3  recieved  FLOOD  from: 2
Node: 0  recieved  FLOOD  from: 3
Node: 1  recieved  FLOOD  from: 3
Node: 0  recieved  SHORT  from: 0
Node: 3  recieved  ECHO  from: 1
Node: 0  recieved  SHORT  from: 3
Deadlock detected
Node: 2  recieved  TERMINATE  from: 0
Node: 1  recieved  TERMINATE  from: 0
Node: 3  recieved  TERMINATE  from: 0
Terminated
Terminated
Terminated
Terminated
-------------------------------------------------------
Primary job  terminated normally, but 1 process returned
a non-zero exit code.. Per user-direction, the job has been aborted.
-------------------------------------------------------
--------------------------------------------------------------------------
mpiexec detected that one or more processes exited with non-zero status, thus causing
the job to be terminated. The first process to do so was:

  Process name: [[21413,1],2]
  Exit code:    1
--------------------------------------------------------------------------
sarvat@sarvat-Lenovo-ideapad-320-15IKB:~/Desktop$ 
```

For Example 2(Fig 2), we received the following output:


Activities
Terminal
Thu 20:21
sarvat@sarvat-Lenovo-ideapad-320-15IKB: ~/Desktop
File Edit View Search Terminal Help

```
sarvat@sarvat-Lenovo-ideapad-320-15IKB:~/Desktop$ mpiexec -n 4 python3 ds-final.py test2.txt
Node: 1  recieved  FLOOD  from: 0
Node: 2  recieved  FLOOD  from: 0
Node: 0  recieved  ECHO  from: 1
Node: 1  recieved  FLOOD  from: 2
Node: 0  recieved  SHORT  from: 0
Node: 2  recieved  ECHO  from: 1
Node: 0  recieved  SHORT  from: 2
Node: 3  recieved  FLOOD  from: 2
Node: 1  recieved  FLOOD  from: 3
Node: 3  recieved  ECHO  from: 1
Node: 2  recieved  ECHO  from: 3
Node: 0  recieved  ECHO  from: 2
Node: 1  recieved  TERMINATE  from: 0
Node: 2  recieved  TERMINATE  from: 0
No deadlock
Node: 3  recieved  TERMINATE  from: 0
Terminated

Terminated
Terminated
-------------------------------------------------------
Primary job  terminated normally, but 1 process returned
a non-zero exit code.. Per user-direction, the job has been aborted.
-------------------------------------------------------
--------------------------------------------------------------------------
mpiexec detected that one or more processes exited with non-zero status, thus causing
the job to be terminated. The first process to do so was:

  Process name: [[21324,1],0]
  Exit code:    1
--------------------------------------------------------------------------
sarvat@sarvat-Lenovo-ideapad-320-15IKB:~/Desktop$
```

4. Comparison of the two algorithms

Criteria	**Bracha-Toueg**	**Kshemkalyani-Singhal**
Phase	2	2
Delay	4*diameter of WFG	2*diameter of WFG
Messages	4*edges	4*edges-2*nodes+leaf nodes-edges to leaf nodes

5. Github Repo

https://github.com/himani19/DS_Project.git

6. References

1. Book- Distributed Algorithms: An Intuitive Approach (MIT Press) by Wan Fokkink

2. http://web.cs.iastate.edu/~borzoo/teaching/15/CAS769/lectures/week3.pdf
3. https://cse.iitkgp.ac.in/~pallab/Distributed_Systems_2016_17/ds_2016_17_lec11.pdf
4. https://www.cs.uic.edu/~ajayk/Chapter10.pdf