Distributed Deadlock Detection and Resolution

10IT60R12
Soumadip Biswas
School of Information technology
Indian Institute of Technology
Kharagpur, India

10IT60R03
Ishant Choukse
School of Information technology
Indian Institute of Technology
Kharagpur, India

10IT60R13
Tanmay Saha
School of Information technology
Indian Institute of Technology
Kharagpur, India

10IT60R07
Vaibhav Neharkar
School of Information technology
Indian Institute of Technology
Kharagpur, India

1. Introduction

Deadlock, as the name suggests, is the condition where the system ceases to advance from the state, it has presently gotten itself into. The necessary and sufficient conditions for deadlock are mutual exclusion, no preemption, hold-and-wait and circular wait. Many deadlock detection schemes have been proposed over the years and many have already been put into use. The most applicable deadlock detection scheme that is employed is the edge chasing scheme, originally proposed by Chandy et al. However, deadlock detection in a distributed system is a very challenging task due to many reasons, some are as mentioned below

1. Taking snapshots of the whole system in distributive systems at any instance of time is not an easy task.
2. Communication delays sometimes might lead to a known problem of *Phantom Deadlocks*.
3. Legacy distributed deadlock detection schemes are of high message complexity.(e.g. flooding)

In the following paper we have studied four different deadlock detection schemes, provided on the models, single resource request model, *p* out of *q* resource request model and OR-model.

2. Paper work mapping

a. Soumadip Biswas
   → An O(n) distributed deadlock resolution algorithm
b. Tanmay Saha
   → A Low Communication Cost Algorithm for Distributed Deadlock
c. Ishant Choukse
   → An Optimal, Distributed Deadlock Detection and Resolution Algorithm for Generalized Model in Distributed Systems.
d. Vaibhav Neharkar
   → An Efficient Distributed Algorithm for Detection of Knots and Cycles in a Distributed Graph

3. Overall organization

The rest of the document contains study of the papers mentioned in the section 2. Next in section 4.1 describes “An O(n) distributed deadlock resolution algorithm”, section 4.2 describes “A Low Communication Cost Algorithm for Distributed Deadlock”, section 4.3 describes “An Optimal, Distributed Deadlock Detection and Resolution Algorithm for Generalized Model in Distributed Systems”, section 4.4 describes “An Efficient Distributed Algorithm for Detection of Knots and Cycles in a Distributed Graph”. Next in the document there is a conclusion part which actually is a gist of this paper and in section 6 the references are noted down.

4. Work done

4.1 Soumadip: “An O(n) distributed deadlock resolution algorithm” [1]

4.1.1 Key Idea

The main idea of this algorithm is to utilize every message sent by the system for detecting a deadlock I the system, rather unnecessarily sending more and more messages and increasing the message load of the system as a whole. This algorithm uses two kinds of messages namely forward and backward message, and then ensures that no node in the system will receive more than 2 messages in deadlock detection procedure. Then it tries to figure out the first successor of every node which having a greater id and also in the wait-for path; and similarly finds the first greater predecessor in that path. Now it is evident that if there is a cycle and every node in that cycle having unique ids then the 2^{nd} highest id node will end up having the same id as a successor as well as a predecessor which is greater than itself; using this truth the following algorithm rather the pseudo-code states how it is possible to find a deadlock in the system and to resolute it at the same time and with O (n) complexity.

4.1.2 Assumptions:

a. Single resource request model
b. FIFO channel
c. Each node has unique id

4.1.3 Algorithm:

→ Whenever a node is blocked while requesting for a resource, it sends query with its id both ways (i.e. node on which this node is waiting and node which is waiting on this node) for checking whether there is deadlock and wait for response

→ The receiving nodes of the previous query will be returned whenever there is a node of higher id in the path of the query with the greater node.

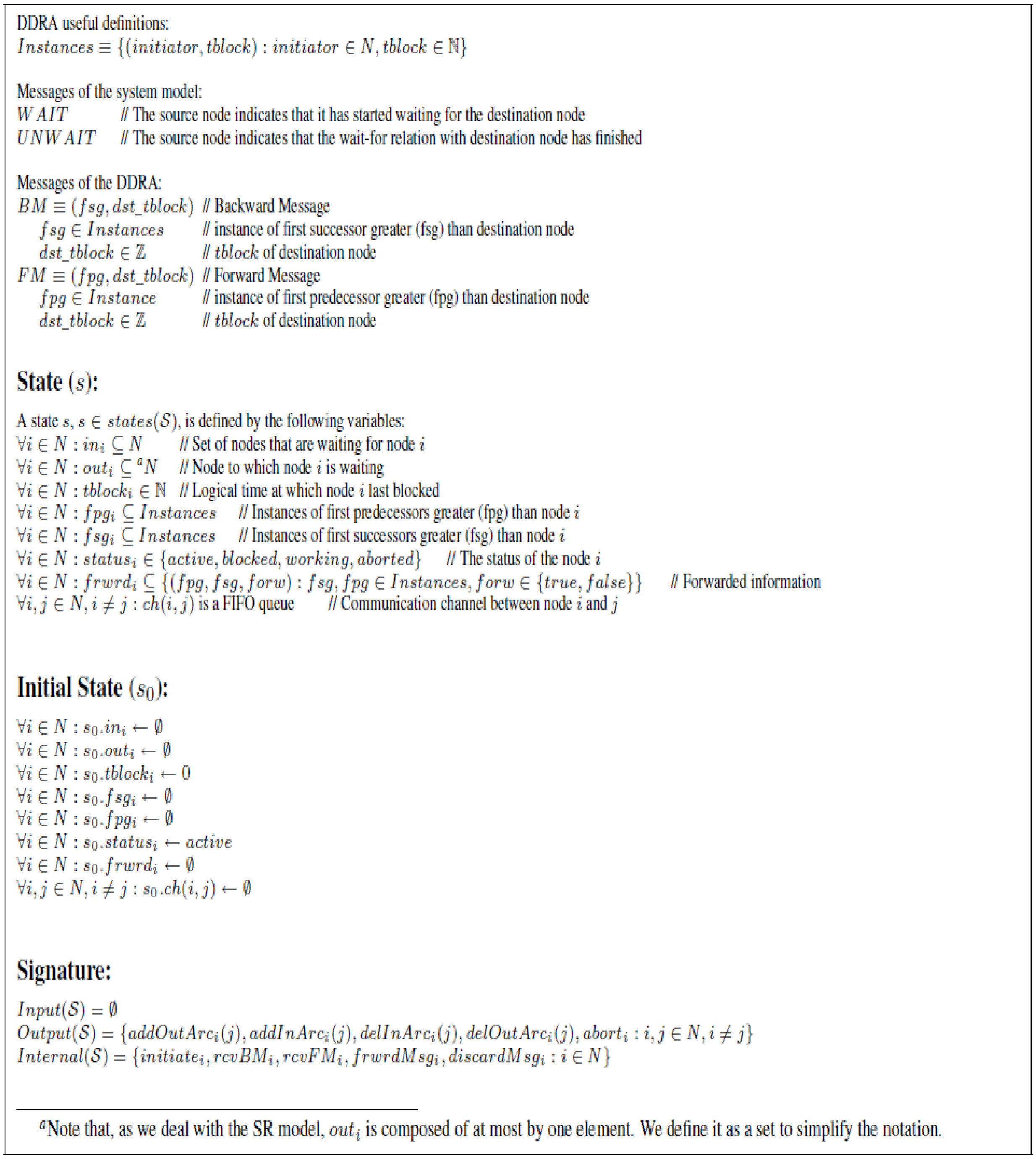
→ On receiving both the reply if it is found that both the replies are indicating same id then a signal will be sent to the node with that id to abort

→ On receiving the signal to abort that node aborts and this indeed resolute the deadlock.

❖ Following we will see one implementation of the algorithm which shows a message complexity of O (n).

4.1.4 Pseudo Code:

Message formats, states, initial states and signature of the system



DDRA useful definitions:
$Instances \equiv \{(initiator, tblock) : initiator \in N, tblock \in \mathbb{N}\}$

Messages of the system model:
$WAIT$ // The source node indicates that it has started waiting for the destination node
$UNWAIT$ // The source node indicates that the wait-for relation with destination node has finished

Messages of the DDRA:
$BM \equiv (fsg, dst\_tblock)$ // Backward Message
$fsg \in Instances$ // instance of first successor greater (fsg) than destination node
$dst\_tblock \in \mathbb{Z}$ // $tblock$ of destination node
$FM \equiv (fpg, dst\_tblock)$ // Forward Message
$fpg \in Instance$ // instance of first predecessor greater (fpg) than destination node
$dst\_tblock \in \mathbb{Z}$ // $tblock$ of destination node

State (s):

A state s, $s \in states(\mathcal{S})$, is defined by the following variables:
$\forall i \in N : in_i \subseteq N$ // Set of nodes that are waiting for node i
$\forall i \in N : out_i \subseteq {}^a N$ // Node to which node i is waiting
$\forall i \in N : tblock_i \in \mathbb{N}$ // Logical time at which node i last blocked
$\forall i \in N : fpg_i \subseteq Instances$ // Instances of first predecessors greater (fpg) than node i
$\forall i \in N : fsg_i \subseteq Instances$ // Instances of first successors greater (fsg) than node i
$\forall i \in N : status_i \in \{active, blocked, working, aborted\}$ // The status of the node i
$\forall i \in N : frwrd_i \subseteq \{(fpg, fsg, forw) : fsg, fpg \in Instances, forw \in \{true, false\}\}$ // Forwarded information
$\forall i, j \in N, i \neq j : ch(i, j)$ is a FIFO queue // Communication channel between node i and j

Initial State (s_0):

$\forall i \in N : s_0.in_i \leftarrow \emptyset$
$\forall i \in N : s_0.out_i \leftarrow \emptyset$
$\forall i \in N : s_0.tblock_i \leftarrow 0$
$\forall i \in N : s_0.fsg_i \leftarrow \emptyset$
$\forall i \in N : s_0.fpg_i \leftarrow \emptyset$
$\forall i \in N : s_0.status_i \leftarrow active$
$\forall i \in N : s_0.frwrd_i \leftarrow \emptyset$
$\forall i, j \in N, i \neq j : s_0.ch(i, j) \leftarrow \emptyset$

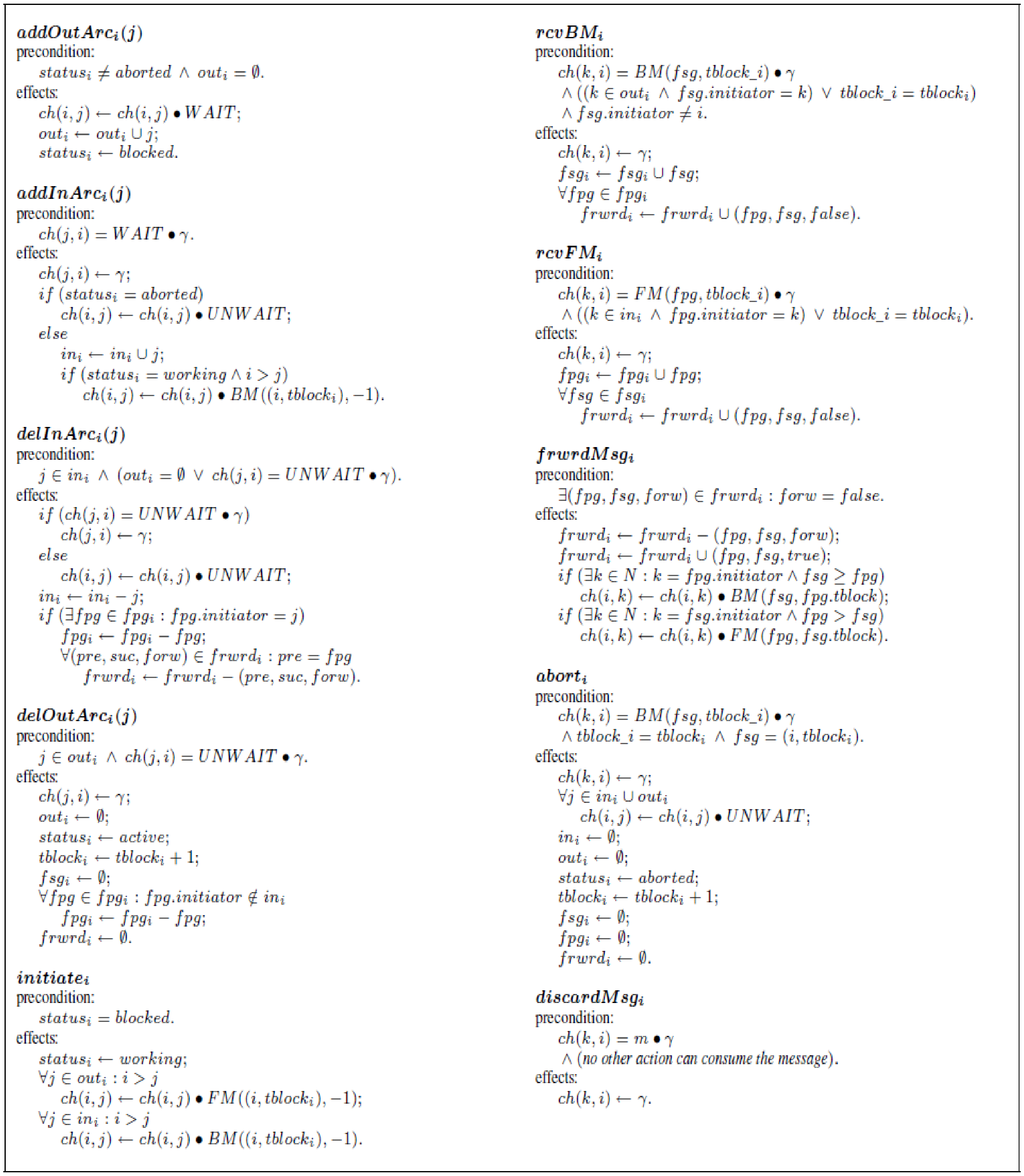
Signature:

$Input(\mathcal{S}) = \emptyset$
$Output(\mathcal{S}) = \{addOutArc_i(j), addInArc_i(j), delInArc_i(j), delOutArc_i(j), abort_i : i, j \in N, i \neq j\}$
$Internal(\mathcal{S}) = \{initiate_i, rcvBM_i, rcvFM_i, frwrdMsg_i, discardMsg_i : i \in N\}$

[a]Note that, as we deal with the SR model, out_i is composed of at most by one element. We define it as a set to simplify the notation.

Figure 1: State, initial state (s0) and signature of the system S. [1]

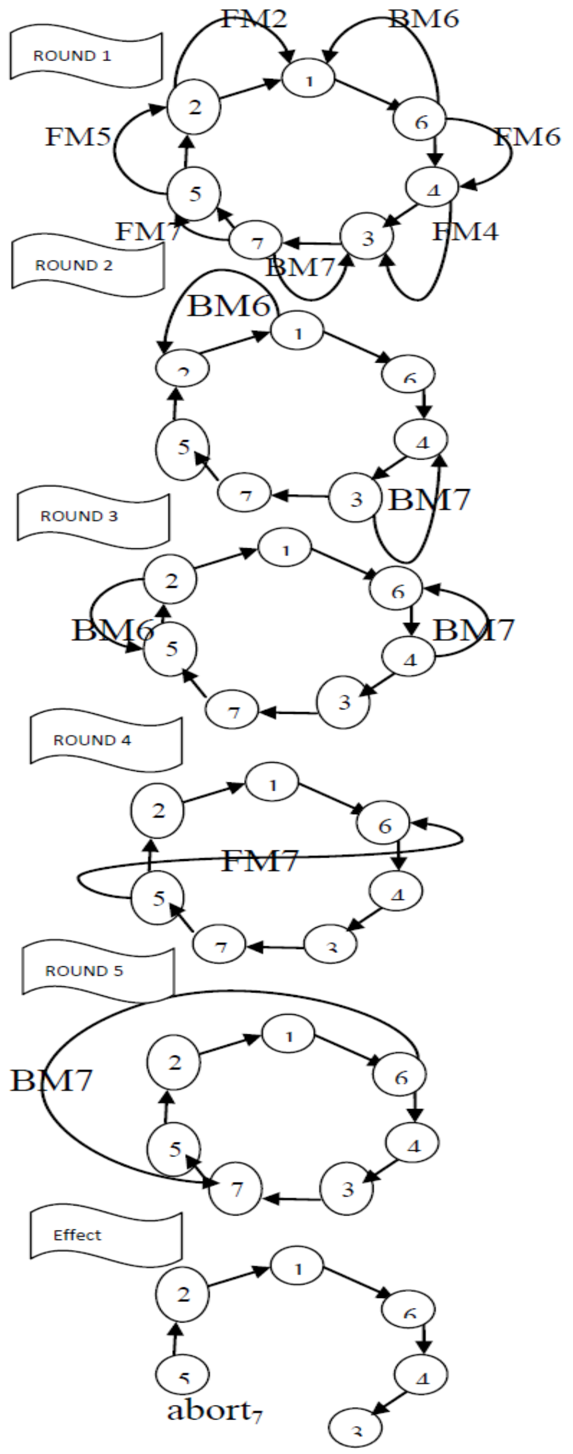
Actions of the System



```
$addOutArc_i(j)$
precondition:
    $status_i \neq aborted \ \wedge \ out_i = \emptyset$.
effects:
    $ch(i,j) \leftarrow ch(i,j) \bullet WAIT$;
    $out_i \leftarrow out_i \cup j$;
    $status_i \leftarrow blocked$.

$addInArc_i(j)$
precondition:
    $ch(j,i) = WAIT \bullet \gamma$.
effects:
    $ch(j,i) \leftarrow \gamma$;
    $if\ (status_i = aborted)$
        $ch(i,j) \leftarrow ch(i,j) \bullet UNWAIT$;
    $else$
        $in_i \leftarrow in_i \cup j$;
        $if\ (status_i = working \wedge i > j)$
            $ch(i,j) \leftarrow ch(i,j) \bullet BM((i, tblock_i), -1)$.

$delInArc_i(j)$
precondition:
    $j \in in_i \ \wedge \ (out_i = \emptyset \ \vee \ ch(j,i) = UNWAIT \bullet \gamma)$.
effects:
    $if\ (ch(j,i) = UNWAIT \bullet \gamma)$
        $ch(j,i) \leftarrow \gamma$;
    $else$
        $ch(i,j) \leftarrow ch(i,j) \bullet UNWAIT$;
    $in_i \leftarrow in_i - j$;
    $if\ (\exists fpg \in fpg_i : fpg.initiator = j)$
        $fpg_i \leftarrow fpg_i - fpg$;
        $\forall (pre, suc, forw) \in frwrd_i : pre = fpg$
            $frwrd_i \leftarrow frwrd_i - (pre, suc, forw)$.

$delOutArc_i(j)$
precondition:
    $j \in out_i \ \wedge \ ch(j,i) = UNWAIT \bullet \gamma$.
effects:
    $ch(j,i) \leftarrow \gamma$;
    $out_i \leftarrow \emptyset$;
    $status_i \leftarrow active$;
    $tblock_i \leftarrow tblock_i + 1$;
    $fsg_i \leftarrow \emptyset$;
    $\forall fpg \in fpg_i : fpg.initiator \notin in_i$
        $fpg_i \leftarrow fpg_i - fpg$;
    $frwrd_i \leftarrow \emptyset$.

$initiate_i$
precondition:
    $status_i = blocked$.
effects:
    $status_i \leftarrow working$;
    $\forall j \in out_i : i > j$
        $ch(i,j) \leftarrow ch(i,j) \bullet FM((i, tblock_i), -1)$;
    $\forall j \in in_i : i > j$
        $ch(i,j) \leftarrow ch(i,j) \bullet BM((i, tblock_i), -1)$.

$rcvBM_i$
precondition:
    $ch(k,i) = BM(fsg, tblock\_i) \bullet \gamma$
    $\wedge ((k \in out_i \ \wedge \ fsg.initiator = k) \ \vee \ tblock\_i = tblock_i)$
    $\wedge fsg.initiator \neq i$.
effects:
    $ch(k,i) \leftarrow \gamma$;
    $fsg_i \leftarrow fsg_i \cup fsg$;
    $\forall fpg \in fpg_i$
        $frwrd_i \leftarrow frwrd_i \cup (fpg, fsg, false)$.

$rcvFM_i$
precondition:
    $ch(k,i) = FM(fpg, tblock\_i) \bullet \gamma$
    $\wedge ((k \in in_i \ \wedge \ fpg.initiator = k) \ \vee \ tblock\_i = tblock_i)$.
effects:
    $ch(k,i) \leftarrow \gamma$;
    $fpg_i \leftarrow fpg_i \cup fpg$;
    $\forall fsg \in fsg_i$
        $frwrd_i \leftarrow frwrd_i \cup (fpg, fsg, false)$.

$frwrdMsg_i$
precondition:
    $\exists (fpg, fsg, forw) \in frwrd_i : forw = false$.
effects:
    $frwrd_i \leftarrow frwrd_i - (fpg, fsg, forw)$;
    $frwrd_i \leftarrow frwrd_i \cup (fpg, fsg, true)$;
    $if\ (\exists k \in N : k = fpg.initiator \wedge fsg \geq fpg)$
        $ch(i,k) \leftarrow ch(i,k) \bullet BM(fsg, fpg.tblock)$;
    $if\ (\exists k \in N : k = fsg.initiator \wedge fpg > fsg)$
        $ch(i,k) \leftarrow ch(i,k) \bullet FM(fpg, fsg.tblock)$.

$abort_i$
precondition:
    $ch(k,i) = BM(fsg, tblock\_i) \bullet \gamma$
    $\wedge tblock\_i = tblock_i \ \wedge \ fsg = (i, tblock_i)$.
effects:
    $ch(k,i) \leftarrow \gamma$;
    $\forall j \in in_i \cup out_i$
        $ch(i,j) \leftarrow ch(i,j) \bullet UNWAIT$;
    $in_i \leftarrow \emptyset$;
    $out_i \leftarrow \emptyset$;
    $status_i \leftarrow aborted$;
    $tblock_i \leftarrow tblock_i + 1$;
    $fsg_i \leftarrow \emptyset$;
    $fpg_i \leftarrow \emptyset$;
    $frwrd_i \leftarrow \emptyset$.

$discardMsg_i$
precondition:
    $ch(k,i) = m \bullet \gamma$
    $\wedge$ (no other action can consume the message).
effects:
    $ch(k,i) \leftarrow \gamma$.
```

Figure 2: Actions of the system S. [1]

4.1.5 Example:



	fpg	fsg	frwrd
1:	2	6	2,6,F
2:	5	-	-
3:	4	7	4,7,F
4:	6	-	-
5:	7	-	-
6:	-	-	-
7:	-	-	-

	fpg	fsg	frwrd
1:	2	6	2,6,T
2:	5	6	5,6,F
3:	4	7	4,7,T
4:	6	7	6,7,F
5:	7	-	-
6:	-	-	-
7:	-	-	-

	fpg	fsg	frwrd
1:	2	6	2,6,T
2:	5	6	5,6,T
3:	4	7	4,7,T
4:	6	7	6,7,T
5:	7	6	7,6,F
6:	-	7	-
7:	-	-	-

	fpg	fsg	frwrd
1:	2	6	2,6,T
2:	5	6	5,6,T
3:	4	7	4,7,T
4:	6	7	6,7,T
5:	7	6	7,6,T
6:	7	7	7,7,F
7:	-	-	-

	fpg	fsg	frwrd
1:	2	6	2,6,T
2:	5	6	5,6,T
3:	4	7	4,7,T
4:	6	7	6,7,T
5:	7	6	7,6,T
6:	7	7	7,7,F
7:	-	7	-

Finally node 7 receives its own BM and it aborts, thus deadlock resolute.

Figure 4.1.3: One example of execution

4.1.6 Complexity:

As it can be seen from the pseudo-code itself and also from the example stated above that for detecting a deadlock in the system this algorithm will pass only 2 messages per node in the cycle, but for the greatest node in the system as no other node can send any message to this node. Thus in the worst case if there is ‘n’ no of nodes in the system and all of them are involved in a deadlock cycle then, the system will require 2*(n-1) messages to find that there is a deadlock in the system and also to detect the greatest id node which need to be aborted and will take another 1 message as a signal to abort.
Thus total message complexity is: [2*(n - 1) + 1] = (2*n - 1). → This is indeed O (n).

4.2 Tanmay: “A Low Communication Cost Algorithm for Distributed Deadlock” [2]

4.2.1 Deadlock detection and recovery with minimized message complexity

Many deadlock detection schemes have been proposed over the years. However, the most applicable deadlock detection scheme that is employed is the edge chasing scheme, originally proposed by Chandy et al. [2]. In this scheme the main logic implemented is that a certain probe is made to circulate through the nodes, and deadlock, if any, is detected in the process. Here, in this paper, the main focus is one the Single-unit Request model (SR model). SR model refers to those where only a single unit of resource has been requested for. Again, this algorithm makes its inferences based on the wait-for-graph (WFG). In all major algorithms of this kind, after a deadlock has been detected, another round of probe-passing is executed, wherein the previously sent probes, which have now become obsolete, are cleared up. However, this increases the message complexity of the algorithm and it would be advantageous if a method could be deployed wherein the previously stored stale message are cleared up right away, instead of waiting for another round to get cleared up. Again, one major advantage of this paper is that the method of selecting the victim process, the one which has to abort processing, is not based on any pre-defined condition, instead it is dynamically selected. Following mentioned are certain characteristics of the paper:

i. It allows multiple initiations of the algorithm.

ii. A special message, *recovery*, is used to clean up the previously stored stale probes. However, this does not require the deadlock to be resolved.

iii. The *victim* is not defined ‘a priori’, and thus resistant to starvation.

iv. Decreases the communication cost to the orders of $O\ (n \log n)$ in the worst cases.

v. The deadlock detection latency in the worst case is $O\ (n \log n)$.

4.2.2 System model

The system model describes the state of the system while the algorithm initiates its execution. Here, the system is a distributed system, where all the different processes are executed independently in their respective machines. Even, the resource managers consume resources and thus qualify the criterion of being termed as processes, and hence are represented as nodes. Here, we assume a universally unique identifier for all the nodes. And no assumption on any kind of shared memory has been made. It is assumed that a connected reliable FIFO channel communication exists between all the nodes, even if the connection is a logical one. Messages get delivered eventually and the message delay between any two nodes is finite, although the delays between nodes might vary on the pair of nodes in question. A node is said to be blocked if its request for some resource has not been granted by the serving node. As it has already been stated that it is a single request system, hence here a node waits for at most one node.

In this algorithm, each node has the ability to initiate the algorithm. And when a process has to abort, it relinquishes all its resources and the resources return to their original states as if the node did not even exist. Even the pending request, if the case be, is also removed. In this algorithm, a special arc is being used to represent the request for resource from one node to another, known as the *Asynchronous Wait-For-Graph (AWFG)*. Here an edge for resource from *i* to *j* is made into an arc so that the information regarding which node is waiting for which is known to all. To achieve this e_{ij} is used to create an arc from *i* to *j* to represent that *i* waits for a resource available in *j*. So the arc $< i,\ e_{ij} >$ can clearly be interpreted as a request for a resource has been made by node *i* to *j*, and that *i* knows about it. Similarly, for $<e_{ij}, j>$ it is obvious that *j* knows about a request pending from *i* for a resource in *j*.

The model mainly comprises of a set of arcs which models the wait-for-graph, a set of aborted processes and a *version* to control the order of arc formations. And then there are a set of preconditions, which if satisfied triggers a set of actions, and these actions in turn triggers a state change.

A particular node in question can be in one of the following states, which is represented by the variable $status_i$:

i. Active: If it is not waiting for any resource on any other node.

ii. Blocked: If it is waiting for a resource to be granted, after the request has been made.

iii. Deadlock: If a certain node decides to break the cycle, after a deadlock is detected.

iv. *level#x:* If a certain node starts the instance at level *x*. Level is generally used to signify the round in which the algorithm is, however with respect to that particular node in question.

v. *level#x_received:* If a certain node receives a probe with a smaller level than its current status.

A node at a certain level shall send probes with that level as its type. Certain other messages that are being passed are: *ant(i)*, which retrieves the immmediate predecessors of the node; *incoming(i)*, which returns the number of arcs which end at the node *i*; and, *outgoing(i)*, which returns the number of arcs going out from the node *i*.

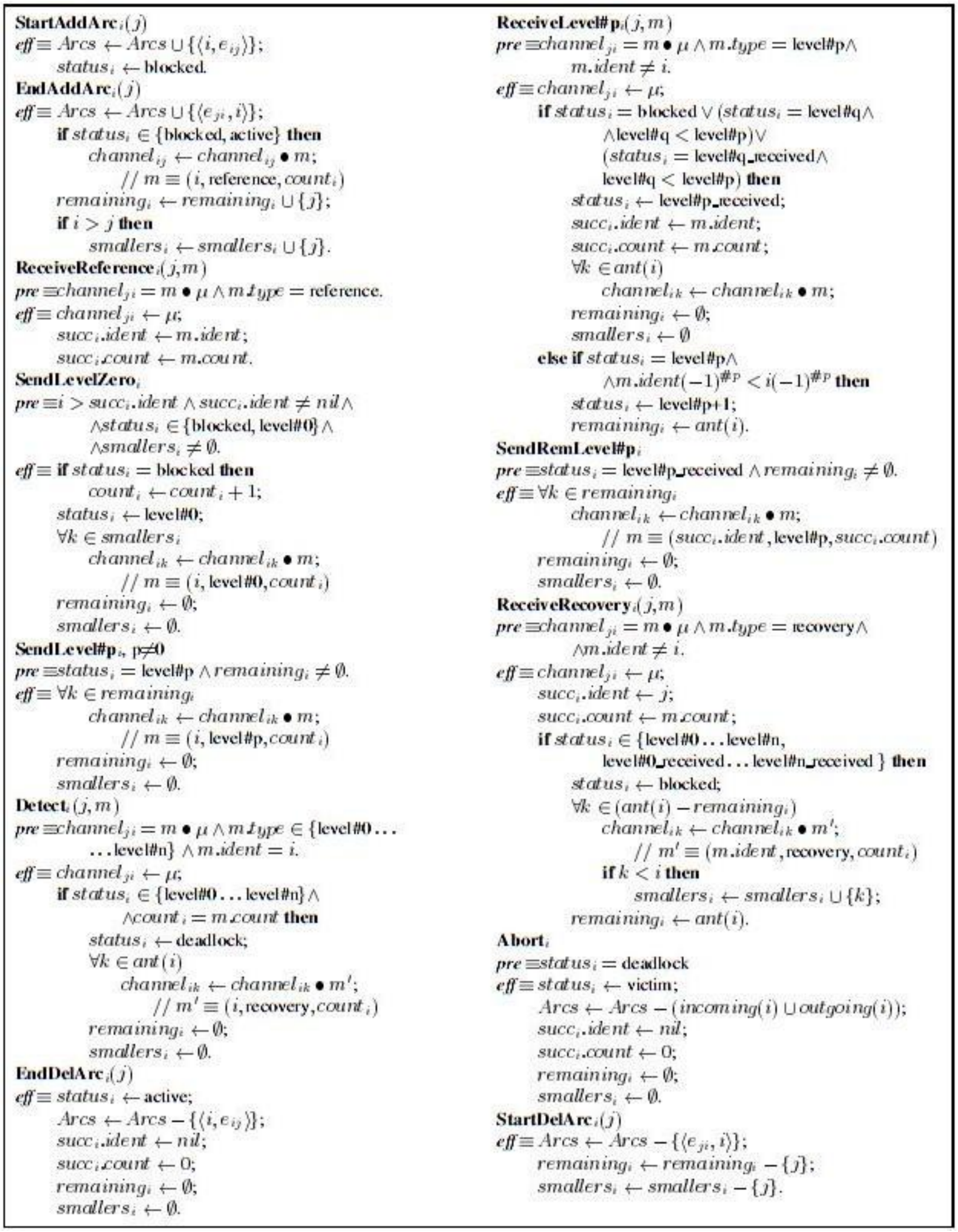
A variable $succ_i.ident$ is used to identifies the successors to the node *i* and $succ_i.count$ maintains the count of the successors available.

4.2.3 The algorithm

Here first the wait-for-graph is improvised by changing the edge into arcs. This is achieved by replacing an edge by two arcs, using *StartAddArc*$_i$*(j)* and *EndAddArc*$_j$*(i).* This would signify that a request is pending from *i* to *j*, and that it is known to both the ends. As *StartAddArc*$_i$*(j)* makes it known to the node *i* about a request pending on node *j*, the *EndAddArc*$_j$*(i)* does just the opposite and sends a*reference* reply back to the node waiting for its reply, here *i*.

Now, on receipt of the *reference* reply it depends on the node whether to initiate an instance of the algorithm. For that it first checks if the node to which it is about to probe has an identifier smaller than its own. Then, it puts it status as *blocked* and initiates the algorithm at *level#0*. And then it just keeps a count as to how many instances of the algorithm has been initiated known to it. However, there is a precondition for the initiator, and it is that the initiator identifier must be larger than its immediate neighbors. And as the paper puts in, there will certainly be at least one node which satisfies the condition.

Now that this much is done, the node sends a probe to its neighbor with *level#0*. The neighbor or the successor in this case, sends it over to all the channels leading to its own successors. The successors which have identifiers smaller than its own identifier are stored in the array *smallers*$_i$, corresponding to the node *i*. Now, if the node detects that the probe with a certain level, which it has recently received, is equal to its own status and that it has no other channels left to be probed, it immediately can infer that a deadlock is probable. This is because in that case it would mean that after it had initiated the algorithm the probe went around in a loop and came all the way back to its initiator. But in case it receives a probe with a higher level than its own status, it initiates the algorithm at the level it just received, and also initializes its own status to that level. However, if a node receives a probe with a lower level, compared to its own level it simply discards the probe.



```
StartAddArc$_i(j)$
eff $\equiv$ $Arcs \leftarrow Arcs \cup \{\langle i, e_{ij}\rangle\}$;
    $status_i \leftarrow$ blocked.
EndAddArc$_i(j)$
eff $\equiv$ $Arcs \leftarrow Arcs \cup \{\langle e_{ji}, i\rangle\}$;
    if $status_i \in$ {blocked, active} then
        $channel_{ij} \leftarrow channel_{ij} \bullet m$;
            // $m \equiv (i, \text{reference}, count_i)$
    $remaining_i \leftarrow remaining_i \cup \{j\}$;
    if $i > j$ then
        $smallers_i \leftarrow smallers_i \cup \{j\}$.
ReceiveReference$_i(j, m)$
pre $\equiv$ $channel_{ji} = m \bullet \mu \wedge m.type =$ reference.
eff $\equiv$ $channel_{ji} \leftarrow \mu$;
    $succ_i.ident \leftarrow m.ident$;
    $succ_i.count \leftarrow m.count$.
SendLevelZero$_i$
pre $\equiv$ $i > succ_i.ident \wedge succ_i.ident \neq nil \wedge$
        $\wedge status_i \in$ {blocked, level#0} $\wedge$
        $\wedge smallers_i \neq \emptyset$.
eff $\equiv$ if $status_i =$ blocked then
        $count_i \leftarrow count_i + 1$;
    $status_i \leftarrow$ level#0;
    $\forall k \in smallers_i$
        $channel_{ik} \leftarrow channel_{ik} \bullet m$;
            // $m \equiv (i, \text{level\#0}, count_i)$
    $remaining_i \leftarrow \emptyset$;
    $smallers_i \leftarrow \emptyset$.
SendLevel#p$_i$, p≠0
pre $\equiv$ $status_i =$ level#p $\wedge remaining_i \neq \emptyset$.
eff $\equiv$ $\forall k \in remaining_i$
        $channel_{ik} \leftarrow channel_{ik} \bullet m$;
            // $m \equiv (i, \text{level\#p}, count_i)$
    $remaining_i \leftarrow \emptyset$;
    $smallers_i \leftarrow \emptyset$.
Detect$_i(j, m)$
pre $\equiv$ $channel_{ji} = m \bullet \mu \wedge m.type \in$ {level#0 . . .
        . . . level#n} $\wedge m.ident = i$.
eff $\equiv$ $channel_{ji} \leftarrow \mu$;
    if $status_i \in$ {level#0 . . . level#n} $\wedge$
            $\wedge count_i = m.count$ then
        $status_i \leftarrow$ deadlock;
        $\forall k \in ant(i)$
            $channel_{ik} \leftarrow channel_{ik} \bullet m'$;
                // $m' \equiv (i, \text{recovery}, count_i)$
        $remaining_i \leftarrow \emptyset$;
        $smallers_i \leftarrow \emptyset$.
EndDelArc$_i(j)$
eff $\equiv$ $status_i \leftarrow$ active;
    $Arcs \leftarrow Arcs - \{\langle i, e_{ij}\rangle\}$;
    $succ_i.ident \leftarrow nil$;
    $succ_i.count \leftarrow 0$;
    $remaining_i \leftarrow \emptyset$;
    $smallers_i \leftarrow \emptyset$.
ReceiveLevel#p$_i(j, m)$
pre $\equiv$ $channel_{ji} = m \bullet \mu \wedge m.type =$ level#p $\wedge$
        $m.ident \neq i$.
eff $\equiv$ $channel_{ji} \leftarrow \mu$;
    if $status_i =$ blocked $\vee$ ($status_i =$ level#q $\wedge$
            $\wedge$ level#q < level#p) $\vee$
            ($status_i =$ level#q_received $\wedge$
            level#q < level#p) then
        $status_i \leftarrow$ level#p_received;
        $succ_i.ident \leftarrow m.ident$;
        $succ_i.count \leftarrow m.count$;
        $\forall k \in ant(i)$
            $channel_{ik} \leftarrow channel_{ik} \bullet m$;
        $remaining_i \leftarrow \emptyset$;
        $smallers_i \leftarrow \emptyset$
    else if $status_i =$ level#p $\wedge$
            $\wedge m.ident(-1)^{\#p} < i(-1)^{\#p}$ then
        $status_i \leftarrow$ level#p+1;
        $remaining_i \leftarrow ant(i)$.
SendRemLevel#p$_i$
pre $\equiv$ $status_i =$ level#p_received $\wedge remaining_i \neq \emptyset$.
eff $\equiv$ $\forall k \in remaining_i$
        $channel_{ik} \leftarrow channel_{ik} \bullet m$;
            // $m \equiv (succ_i.ident, \text{level\#p}, succ_i.count)$
    $remaining_i \leftarrow \emptyset$;
    $smallers_i \leftarrow \emptyset$.
ReceiveRecovery$_i(j, m)$
pre $\equiv$ $channel_{ji} = m \bullet \mu \wedge m.type =$ recovery $\wedge$
        $\wedge m.ident \neq i$.
eff $\equiv$ $channel_{ji} \leftarrow \mu$;
    $succ_i.ident \leftarrow j$;
    $succ_i.count \leftarrow m.count$;
    if $status_i \in$ {level#0 . . . level#n,
            level#0_received . . . level#n_received } then
        $status_i \leftarrow$ blocked;
        $\forall k \in (ant(i) - remaining_i)$
            $channel_{ik} \leftarrow channel_{ik} \bullet m'$;
                // $m' \equiv (m.ident, \text{recovery}, count_i)$
            if $k < i$ then
                $smallers_i \leftarrow smallers_i \cup \{k\}$;
        $remaining_i \leftarrow ant(i)$.
Abort$_i$
pre $\equiv$ $status_i =$ deadlock
eff $\equiv$ $status_i \leftarrow$ victim;
    $Arcs \leftarrow Arcs - (incoming(i) \cup outgoing(i))$;
    $succ_i.ident \leftarrow nil$;
    $succ_i.count \leftarrow 0$;
    $remaining_i \leftarrow \emptyset$;
    $smallers_i \leftarrow \emptyset$.
StartDelArc$_i(j)$
eff $\equiv$ $Arcs \leftarrow Arcs - \{\langle e_{ji}, i\rangle\}$;
    $remaining_i \leftarrow remaining_i - \{j\}$;
    $smallers_i \leftarrow smallers_i - \{j\}$.
```

Figure 4.2.1: The pseudo code for the algorithm [2]

4.2.4 Example:

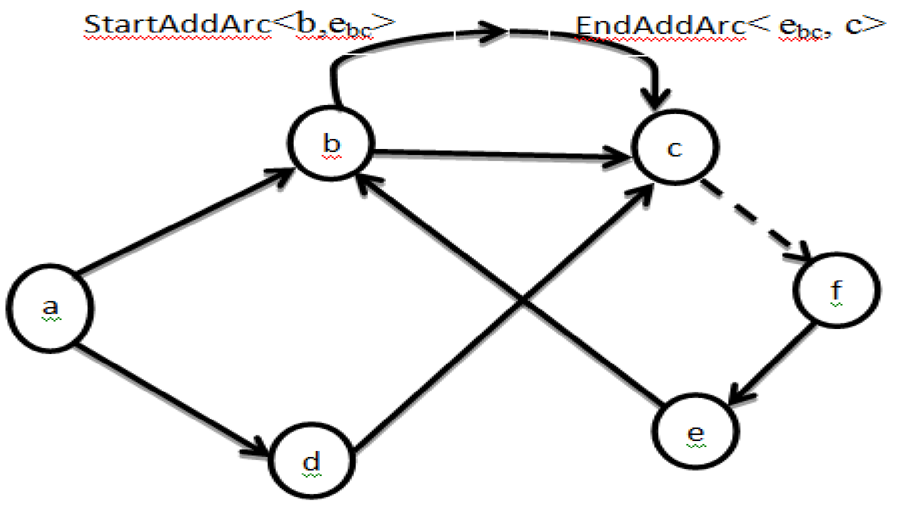


Fig. 4.2.2 a wait-for-graph

Above shown is a wait-for-graph, and the edges shown are the requests made and pending for those nodes, considering the literals as the identifiers and the order in them is maintained by their alphabetical order.
Now say *b* wishes to initiate the algorithm, considering there was no edge from *c* to *f*.

1. It would first add an arc from *b* to *c* and would store that information in *b*.
2. Initialize its level to *level#0*, and would even put its status as *blocked*. Then it would put its identifier on the probe and initiate the algorithm.
3. Store *c* in the list of $smallers_b$, which basically stores the list of all the nodes whose identifiers are smaller than *b*.
4. *e* would reply back immediately with a *reference*, and put status as *active* and thus the $EndAddArc_c(b)$ would be called and this would end in all the nodes running smoothly.

However, considering the edge from *c* to *f* exists and again *b* initiates the algorithm.

1. It would first add an arc from *b* to *c* and would store that information in *b*.
2. Initialize its level to *level#0*, and would even put its status as *blocked*.
3. Store *c* in the list of $smallers_b$.
4. Now *c* initiates another instance of the algorithm, and sends the probe to *f* with
   m. ident = c
   $succ_{b.}$ *ident = f*
   level#0
   This would make *c* as the initiator for *f*.
5. And then *f* would send the probe to other nodes, i.e. *e*.
6. *e* on receipt shall forward it to *b*, its only successor.
7. And as soon as *b* receives the probe back it and finds its status as *blocked* it immediately changes its status to *deadlock*.
8. And later when the abort process is executed this node is chosen as victim and it is aborted.

Now, considering the multiple initializations, say nodes *c* and *a* initiate the algorithm at the same time, and that the edge from *c* to *f* exists.

1. Nodes *c* and *a* add arcs to *f* and *d*.
2. The nodes *f* and *d* reply with type *reference*, and then start the algorithm instances with their statuses as *blocked*.
3. Then these nodes forward the probe to their respective successors, *e* and *c*.
4. *c* on receipt of the probe with *level#0* reinitiates the algorithm with *level#1* and the same procedure continues.
5. And if the probe with the same level and initiator identity returns to the initiator the deadlock is detected.

4.2.5 Complexity calculation

As we can see from above, for a case where all the nodes are at *level#p* there is only one initiator. And for *level#p-1* there has to be two initiators and that too they will have to go around for another round to get to same level of the probe. And thus for *level#p-2* there has to be three initiators. Thus, in order to reach *level#p* the minimum number of nodes required is in successively as: 1, 2, 3, 5…, which is a Fibonacci series.

The paper has proven that the order in case of Fibonacci series is *O (log n)*. And then the number of messages per round is *n*. Thus the worst case complexity of the algorithm is *O (n log n)*.

4.3 Ishant: "An Optimal, Distributed Deadlock Detection and Resolution Algorithm for Generalized Model in Distributed Systems" [3]

4.3.1 Introduction:

In this part of the paper we discuss a generalized deadlock detection algorithm proposed by S. Srinivasan, Rajan Vidya, and Ramasamy Rajaram. Distributed system deadlock detection algorithms can be classified into three categories, depends on which resource request model is used, AND, OR or P out of Q model. In AND model the process requests for n resources and for successful execution of process all resources are needed. In OR model the process requests for n resources and allocation of any one resource is sufficient for execution. But in case of P out of Q model process requests for Q resources out of which P resources are required for execution. So we can say that AND and OR model are the special cases of P out of Q model and because of these P out of Q model is also referred as a generalized deadlock model. In order to detect deadlock in AND and OR model detection of cycle and knot in WFG is sufficient condition. But in case of generalized deadlock detection algorithm cycle is necessary but not sufficient condition and knot is sufficient but not necessary condition.

The algorithm which we discuss in this part is a distributed single phase generalized deadlock detection algorithm which gives the consistent distributed snapshot of global wait-for-graph (WFG).Which have worst case time complexity of 2d time units and the message complexity of 2e, where d is the diameter and e is the number of edges.[1]

4.3.2 Preliminaries

In this algorithm following assumptions about the system will be made –

1. System consists of N process and each process is identified by a unique id.
2. System have no shared memory, each process communicate by message passing using reliable communication link.
3. Communication link is FIFO, messages are delivered according to order of their sending events.
4. Lamport’s logical clock is used to attach timestamp to events.
5. System is fault free and there is no duplication of messages.
6. Events are classified as computational events and control event. Computational event send message such as REQUEST, REPLY, CANCEL and ACK and control event sends message such as CALL and REPORT.
7. In generalized resource request model process request resources is expressed by predicates using logical AND and OR. In this algorithm request of resources are shown in the form of function f_i which was calculated when reply of requests are received and then return true if P resources out of Q are available else return false .

4.3.3 Distributed Deadlock Detection Algorithm

In this algorithm any node i can initiates the algorithm for deadlock detection if it blocks on P out of Q requests. The result of algorithm is the consistent distributed snapshot of global wait-for-graph (WFG). In this algorithm multiple nodes can initiates the algorithm but for simplicity in this paper we discuss the whole process for a single initiator.

4.3.4 Algorithm Description

We can divide the whole algorithm in two phases forward phase and reduction phase. In forward phase initiator sends CALL messages to its immediate successor which in turn propagates the CALL messages to its entire successor and this process is continues until all nodes which are reachable from initiator are covered. Each node stores the identity of node from which it first receive the CALL message, sending node is act as a father of receiving node. After the completion of forward phase a directed spanning tree of the wait for graph is formed.

In this algorithm nodes are categorized as active node and blocked node. Active nodes are those nodes whose requests for P out of Q resources are fulfilled and block nodes are those whose requests are not fulfilled. All leaf node of spanning tree have no dependency on any other node hence they initiated the reduction phase by sending the REPORT message to its father and get reduced. All active nodes immediately sends the replies of its CALL messages in the form of REPORT message and get reduced form spanning tree whereas all blocked nodes send REPORT message to all its wait by edges except the father until it get REPORT message from all its successor. The algorithm gets terminated when all the nodes get REPORT message for their CALL messages. And finally we get WFG which consists of only blocked nodes.

4.3.4.1 Data Structure Used

An array LS_i of N record is used to store snapshot

In this algorithm each process (node) i have following data structure to store local state

t_block_i	To store the timestamp at which process i is last blocked.
IN_i	An array of tuples <k, t_block_k> used to store the process which sends request to i for resources at timestamp t_block_k
OUT_i	An array of process id's which store process-id of the process to whom i send requests for resources
p_i	An integer to store number of resources needed to avoid deadlock
q_i	An integer to store number of requests send
$father_i$	An integer store the process-id of father of i
n_i	An integer which stores the number of tuples in OUT_i

4.3.4.2 Messages Used

CALL message contain three arguments sender identity, initiator identity and timestamp at which init initiated snapshot.

REPORT message contain six arguments sender's father identity, sender identity, timestamp at which init initiated snapshot, number of resource request left, number of request sends, array of victims.

4.3.4.3 Algorithm[3]

```
Initiate a snapshot      /*executed by process i to detect deadlock*/
init ← i
Lsi t .← ti
Lsi . out ←outi
Lsi . in ← □
Lsi . p ← pi
Lsi . q← qi
Lsi . fatheri ← i
send call ( i , i , ti ) to each in j in outi

on sending call( i , init , ti ) /*executed by process i when it blocks on a p out of q request model*/.
For each node j of outi   nodes on which i blocks
send call ( i , i , ti ) to j

on receiving call( j , init , ti ) /* executed by process i on receiving call from j*/
Lsi out ← outi
Lsi . in ← { j }
Lsi . t← ti
Lsi . p ← pi
Lsi . q← qi
Lsi . n ← ni

case(i)   /*node i is a unvisited intermediate node*/
if ( fatheri =udef ^ pi ) →
fatheri ← j
send call ( i , init , ti ) to each j in outi /* i is blocked*/
case(ii)   /*node i is a blocked visited intermediate node*/
if( fatheri = def ) ^ ( pi )→
send report ( fatheri , i , ti , pi ,| ini |, victim ) to j
/* i is blocked visited intermediate node*/


case(iii)  /*node i is a reduced visited intermediate node*/
if( fatheri = def ) ^(¬ pi ) →
send report ( fatheri , i , ti ,0,0, ∅□) to j
```

```
case(iv)  /*node i is a leaf node hence no first edge*/
if($\neg p_i \wedge \neg q_i$)
send report ( $father_i$ , $i$ , $t_i$ ,0,0,$\emptyset$□) to $j$ /* i is unblocked leaf node*/
on receiving report( $father_i$ , $i$ , $t_i$ , $p_j$ ,| $in_j$ |, $victim$ )
/*process i on receiving a report from unblocked node j*/
if($\neg p_j$ ) then → /*report from a unblocked node*/
$p_i \leftarrow p_i$ -1
$n_i \leftarrow n_i$ -1 /*process i on receiving a report from blocked node j*/
else
$n_i \leftarrow n_i$ -1
if (| $in_j$ |>=| $in_i$ |)→
$victim \leftarrow victim \cup \{ j \}$
if (n=0) then →
send report( $father_i$ ,$i$, $t_i$ , $p_i$ ,| $in_j$ |, $victim$ ) to $father_i$
/*node with highest in array degree is chosen as victim*/
else
send report( $father_i$ ,$i$, $t_i$ , $p_i$ ,| $in_j$ |, $victim$ ) to other in array nodes
else
$victim \leftarrow victim \cup \{ i \}$
if (n=0) then →
send report( $father_i$ ,$i$, $t_i$ , $p_i$ ,| $in_i$ |, $victim$ ) to $father_i$
else
send report( $father_i$ ,$i$, $t_i$ , $p_i$ ,| $in_i$ |, $victim$ to other in array nodes
```

4.3.5 Example Execution

In this section we give an example of deadlock detection using above algorithm. As show I n figure whole system consist of ten nodes from A to J, where A is a block node on P out of Q request which initiates the algorithm and act as an initiator. To detect the deadlock following steps will be followed-

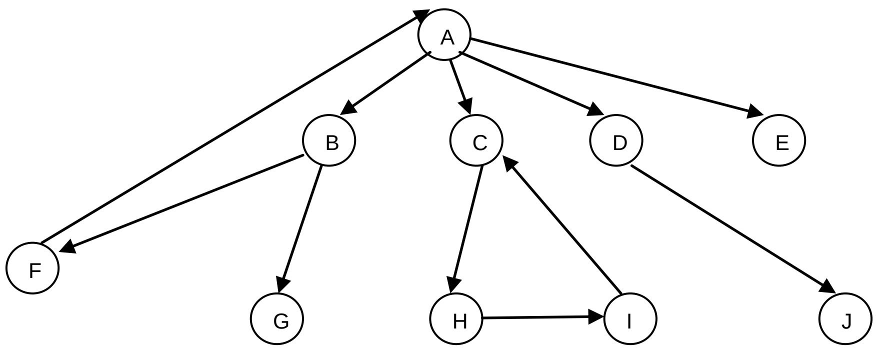


Figure 4.3.1: shows the WFG for system

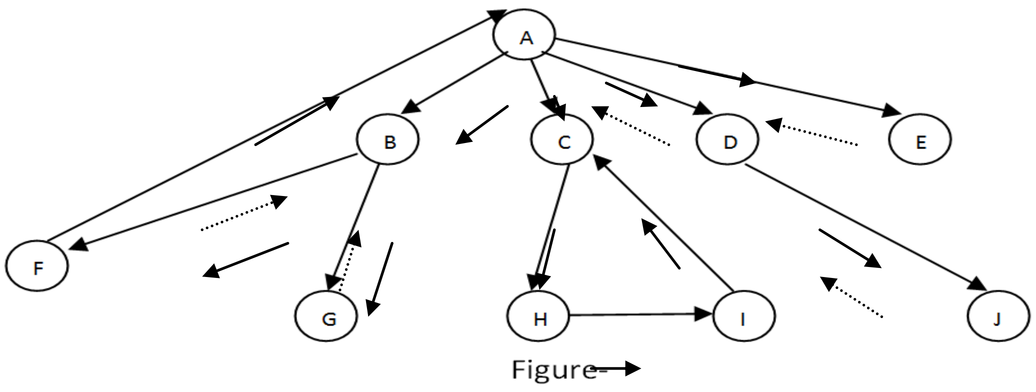


Figure 4.3.2: Shows the diffusion of CALL message and Propagation of REPORT message

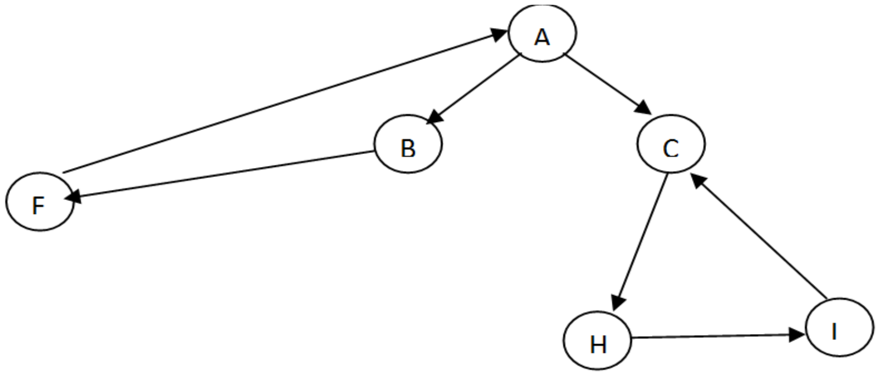


Figure 4.3.3: Snapshot

4.3.5.1 Initialization

```
$init \leftarrow A$
$LsA.\ t \leftarrow tA$
$LsA\ .\ out \leftarrow \{B,C,D,E\}$
$LsA.\ in \leftarrow \emptyset$
$LsA.\ p \leftarrow 4$
$LsA\ .\ q \leftarrow 4$
$LsA.\ fatherA \leftarrow \mathrm{A}$
$Lsi\ .\ n \leftarrow 4$
CALL(A,A,$t_A$) to B
CALL(A,A,$t_A$) to C
CALL(A,A,$t_A$) to D
CALL(A,A,$t_A$) to E
```

4.3.5.2 Forwarding Phase

1. On receiving CALL message at B from A

$father_B = A$
B sends CALL message to F and G
CALL(B,A,t_A) to F
CALL(B,A,t_A) to G

2. On receiving CALL message at C from A

$father_C = A$
C sends CALL message to H
CALL(C,A,t_A) to H

3. On receiving CALL message at D from A

$father_D = A$
D sends CALL message to J
CALL(D,A,t_A) to J

4. On receiving CALL message at F from B

$father_F = B$
F sends CALL message to A
CALL(F,A,t_A) to A

5. On receiving CALL message at H from C

$father_H = C$
H sends CALL message to I
CALL(H,A,t_A) to I

6. On receiving CALL message at I from H

$father_I$=H
I sends CALL message to C
CALL(I,A,t_A) to C

4.3.5.3 Reduction Phase

1. E send report (*A* , *E* , *ti* ,0,0,$^{\emptyset}$□) message to A and get reduced because E is a leaf node.
   1.1 On receiving report message fron E
       p_A = 4-1 and n_A = 4-1
2. J send report (D , *J* , *ti* ,0,0,$^{\emptyset}$□) message to D and get reduced because J is a leaf node.
   2.1 On receiving report message fron E to j
       p_D = 1-1 and n_D = 1-1
   2.2 In step 2.1 value of p_D=0 hence node D also send report (A , *D* , *ti* ,0,0,$^{\emptyset}$□) message to A
       2.2.1 On receiving report message fron D to A
             p_A =3-1 and n_A=3-1
3. G send report (B ,*G* , *ti* ,0,0,$^{\emptyset}$□) message to B and get reduced because G is a leaf node.
   3.1 On receiving report message from G to B

p_B = 2-1 and n_B =2-1

4. Similarly node C send REPORT message to I and I send REPORT message to C and H send report message to C and C send REPORT message to A but they all are not reduced because their p value is not equal to zero.
5. Similarly in case of E, G and B they all are not reduced.
6. Thus these blocked nodes are declared as deadlock node.

4.3.6 Deadlock Resolution

For deadlock resolution a special argument victim is send with REPORT message. Before sending report message each node compares its own degree of IN array with degree of IN array of other node and the node which has higher IN degree is add to victim array, if it is already not added and then send it to the successive nodes. At the time of deadlock resolution initiator and the first element of victim array are removed.

4.3.7 Performance Analysis

In this algorithm each node send one CALL message and one REPORT message at each edge of spanning tree so the message complexity of algorithm is 2e where e is the number of edges of spanning tree. Similarly each node traverse two times in algorithm hence time complexity is 2d where d is the diameter of system. And the message size is O (1) because it uses fixed size message format.

4.4 Vaibhav: "An Efficient Distributed Algorithm for Detection of Knots and Cycles in a Distributed Graph" [4]

4.4.1 Introduction

A knot in the graph means every node in the graph is reachable from every other node in that graph and no node outside that graph is reachable from any node in that graph. Thus, a node in a directed graph belongs to a knot if and only if the node can be reached from all the nodes that are reachable from it [4]. Detecting a knot is very important problem in distributed system because it tells us that whether there exists a deadlock in the system. Having a knot is necessary and sufficient condition for existence of deadlock in OR model.

In this paper we are going to discuss about a knot detection algorithm which was proposed by D. Manivannan (Member, IEEE) and Mukesh Singhal (Fellow, IEEE) in paper [4]. In this paper they proposed the algorithm which detects the knot in the system with message complexity of 2e and time delay of 2(d+1) where e is the no of edges in the graph and d is diameter of the graph. The main advantage of this algorithm is that it not only detects the weather node is in the knot but also detects all the nodes involved it the knot with that node. Also if the node is not in the knot but in cycle then it will detect all the nodes which are in cycle with that node. This information about the nodes which are involved in the knot, can be used to resolve the deadlock.

The rest of the paper is organised as follows: In section 4.4.2 we will discuss about the model used in this algorithm, then in 4.4.3 we will discuss the actual algorithm with pseudo code, then in 4.4.4 we give an example of the algorithm and finally in section 4.4.5 we conclude with comparison with some other algorithms.

4.4.2 System Model

The model used in this algorithm includes following:

1. The distributed system can be assumed as graph in which nodes are the nodes in the system and edges are logical link in the between the nodes. For the simplicity of exposition they have assumed that each node is running only one process, hence we can use terms nodes, sites and processes interchangeably.
2. Channels are bidirectional.
3. Each node is having unique id.
4. System is Reliable and asynchronous.
5. There is no failure during the execution of the algorithm.
6. For simplicity, we assume that only one process initiates the knot detection. If multiple processes initiate knot detection, the initiations can be distinguished by initiator id and a sequence number associated with the initiation [4].

4.4.3 The Algorithm

To determine if the initiator is involved in a knot, it only needs to check if every node that is reachable from the initiator node is on a cycle that passes through the initiator [4].

In this section we refer initiator node as i. The pseudo code of algorithm for initiator (Process Pi) and for other node (Process k) is given separately in fig. 1 and fig. 2 respectively. Each algorithm contains repetitive five alternative guarded commands. The guarded command has following form:

Guard $\rightarrow$ Command

If the condition in the guard is true then corresponding command (action) is triggered. Also each action is atomic. If several Actions are enabled at an instant, then only one of these enabled Actions is picked randomly and executed at a time. We assume that each enabled action is given a fair chance for execution [4].

4.4.3.1 States:

It has following variables:

Out_i: It is set of nodes. Initially containing all nodes is directly reachable (immediate successor) from i.

$Seen_i$: It is set containing order pairs of the nodes e.g. if there is order pair (j, k) means that there is path from i to j and edge between j and k. It also means that both k and j have seen the detect knot message and, when k received the detect knot message from j, k did not know yet if it is on a cycle with the initiator [4]. Initially, Seeni is empty.

$Incycle_i$: It is set of nodes, initially empty. It contains nodes which are in cycle with i.

$Cyclenodes_i$: It is set of nodes, initially empty. At the end of algorithm if I is in knot then it contains all the nodes which are in the knot with i. Otherwise it contain all nodes in cycle with I if I is involved in the cycle.

$done_i$: It is Boolean variable, initialised to false. At the end when node receives acknowledgement from its all immediate successor it is set to true.

For all the nodes k other than initiator, all variables mentioned above are same in addition to the following:

$hasseen_k$: It is Boolean variable, initialised to false. It is set to true when node receives the first detect_knot message

$parent_k$: It is initialised 0 means node is not having any parent. It is set to id of node from which it receives the first detect_knot message.

4.4.3.2 Types of Messages:

detect_knot (i, j): In this i is the id of the initiator node and j is the id of the propagator node.

seen_ack (k): The node k sends this message to node which had sent message detect_knot to k that it had already received detect_knot message.

cycle_ack (k): This message is sent by node in order to know the node k that it is in cycle with sender.

parent_ack (Seenj, Incyclej): This message is sent by j to its parent after j had received the acknowledgement from all its successor.

Initial Action:

Variables Out, Incycle, Cyclenodes, done, seen, parent, hasseen of all nodes is initialised to their respective initial values.

When node i want to check whether it is in the knot it will send detect_knot (i, i) message to all its immediate successor i.e. all nodes in its Out_i set.

4.4.3.3 Explanation of the Algorithm

When node k receives message detect_knot (i, j) message it takes one of the following action (from Actions of Pk):

1. If node does not have seen detect_knot (i, j) message before, it will set its parent to j and propagates the detect_knot (i, k) to all nodes in Out_k. It should be noted that the edges along which the first detect knot message is received by each node form a directed spanning tree (DST) rooted at the initiator [4].

2. If node has already seen detect_knot (i, j) message and $Incycle_k$ =Ø then it will send seen_ack (k) to node j.

3. If node has already seen detect_knot (i, j) message and $Incycle_k$!=Ø then it will send cycle_ack (k) to j.

4. If node which received message detect_knot (i, j) is i then it will send cycle_ack (i) to j so that j come to know it is in cycle with i (Action 4 of Pi).

When node j receives message seen_ack (k) message it takes following action:

1. After receiving seen_ack (k) message, j includes order pair (j, k) in its set Seenj. So that if, later, k is found to be on a cycle with the initiator, the initiator can conclude that j is also on a cycle with the initiator [4].

When node j receives message cycle_ack (k) message it takes following action:

1. After receiving this acknowledgement j node add k to its Incyclej set (Action 4 of Pk).

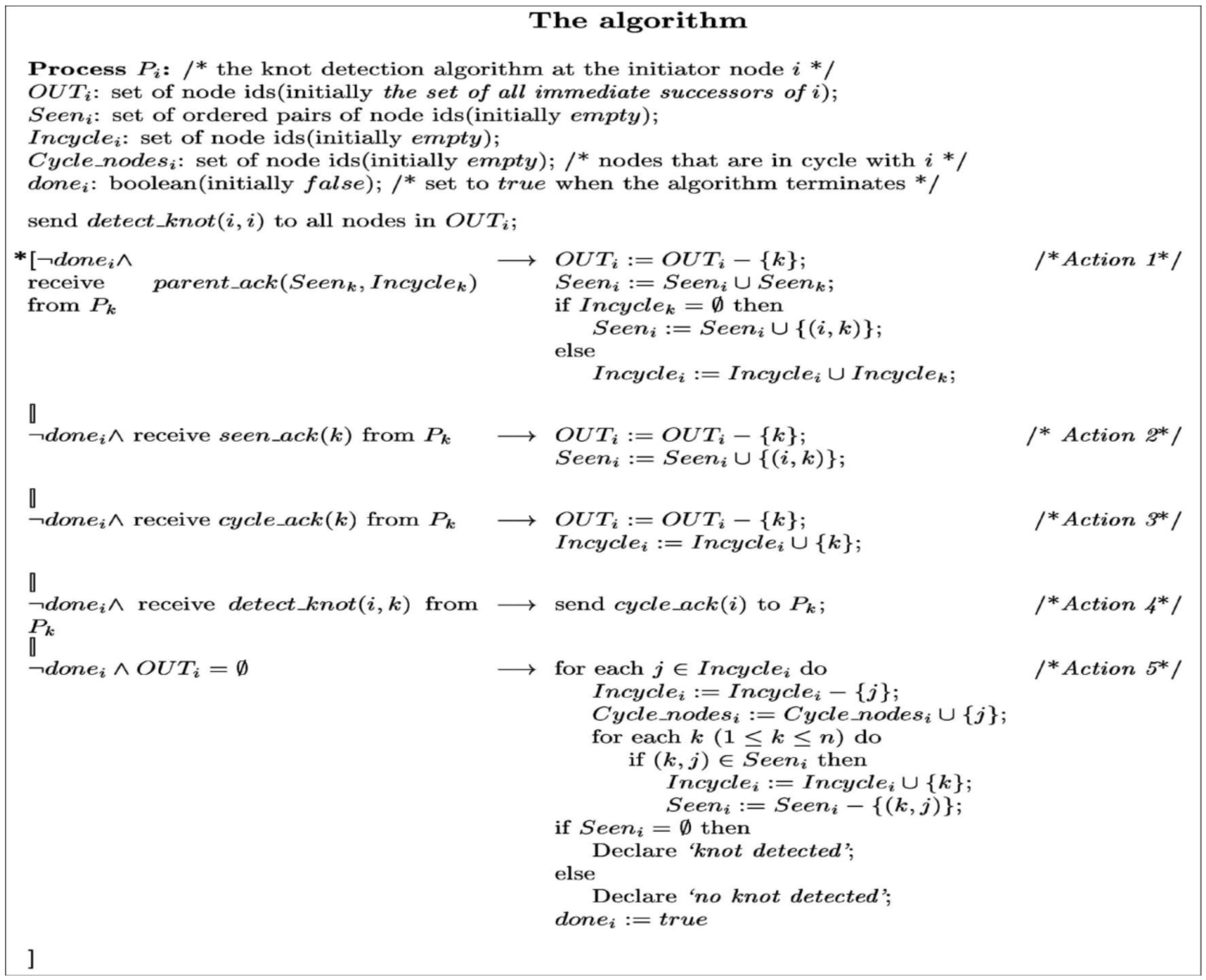
When node j receives message parent_ack (Seenk, Incyclek) message it takes following action:

1. When j receives a parent_ack (Seenk, Incyclek) message from one of its children in the DST, it updates its variables Seenj and Incyclej (Action 2 of Pk) [4].

After a node j receives parent_ack from all its successor, if any one of them is forming a cycle with i then j will include itself in Incyclej sends parent_ack to its parent (Action 5 of Pk).

After i receive parent_ack from its all successor it will initiate Action 5 Pi. Note that, after node i has received an acknowledgment from all its immediate successors, for each node j that is reachable from i, either j 2 Incyclei or (k; j) 2 Seeni for some node k, reachable from i [4]. After execution Action 5 if Seeni is empty then i declares knot detected and nodes in the set Cyclenodesi are in the knot with i.

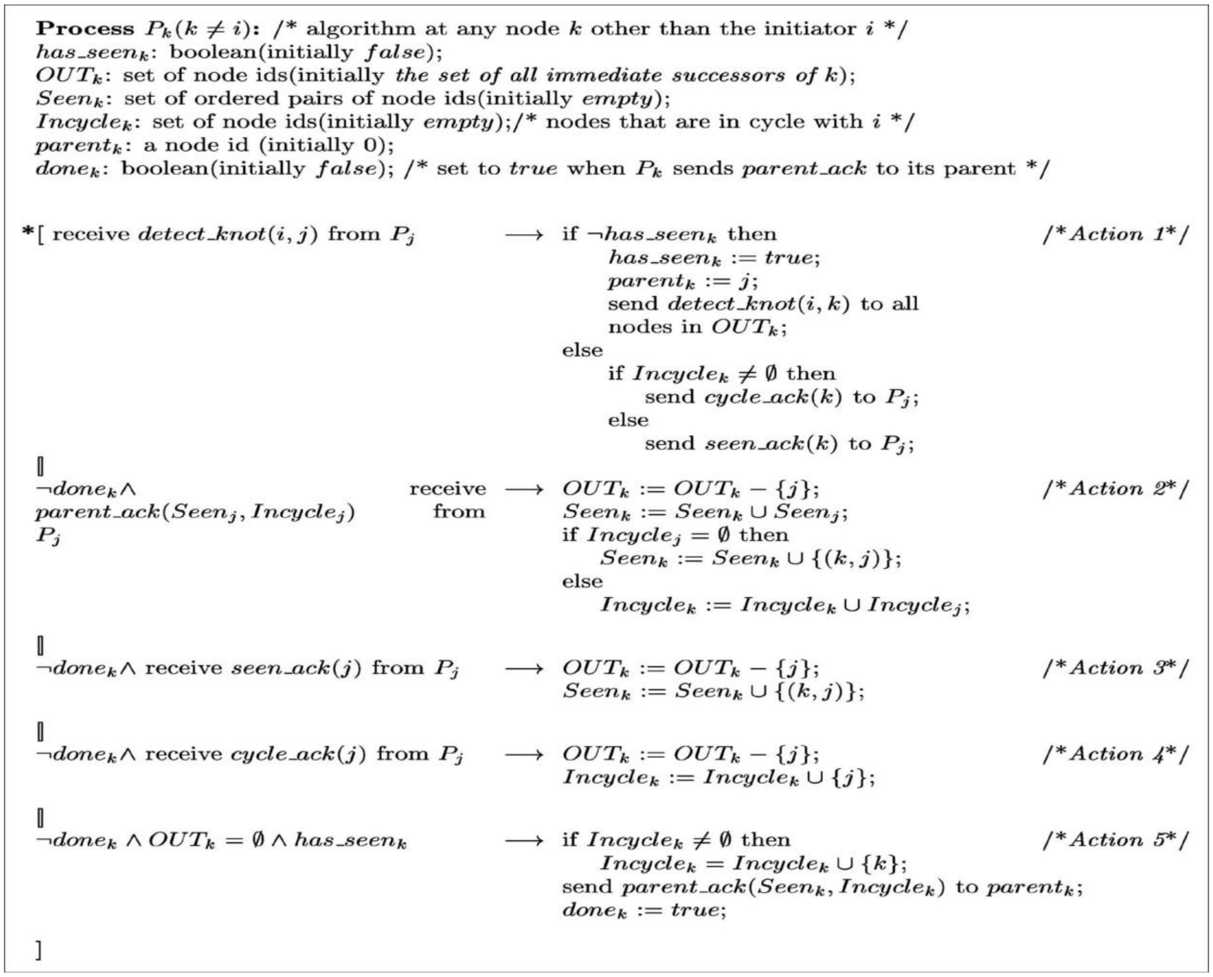
Also we can see from above algorithm that the node does not send a message to the nodes which are not reachable from it.



The algorithm

```
Process $P_i$: /* the knot detection algorithm at the initiator node $i$ */
$OUT_i$: set of node ids(initially the set of all immediate successors of $i$);
$Seen_i$: set of ordered pairs of node ids(initially empty);
$Incycle_i$: set of node ids(initially empty);
$Cycle\_nodes_i$: set of node ids(initially empty); /* nodes that are in cycle with $i$ */
$done_i$: boolean(initially $false$); /* set to $true$ when the algorithm terminates */

send $detect\_knot(i, i)$ to all nodes in $OUT_i$;

*[$\neg done_i \wedge$                              $\longrightarrow$  $OUT_i := OUT_i - \{k\}$;                    /*Action 1*/
 receive $parent\_ack(Seen_k, Incycle_k)$              $Seen_i := Seen_i \cup Seen_k$;
 from $P_k$                                            if $Incycle_k = \emptyset$ then
                                                           $Seen_i := Seen_i \cup \{(i, k)\}$;
                                                       else
                                                           $Incycle_i := Incycle_i \cup Incycle_k$;

 ▯
 $\neg done_i \wedge$ receive $seen\_ack(k)$ from $P_k$   $\longrightarrow$  $OUT_i := OUT_i - \{k\}$;        /* Action 2*/
                                                       $Seen_i := Seen_i \cup \{(i, k)\}$;

 ▯
 $\neg done_i \wedge$ receive $cycle\_ack(k)$ from $P_k$  $\longrightarrow$  $OUT_i := OUT_i - \{k\}$;        /*Action 3*/
                                                       $Incycle_i := Incycle_i \cup \{k\}$;

 ▯
 $\neg done_i \wedge$ receive $detect\_knot(i, k)$ from  $\longrightarrow$  send $cycle\_ack(i)$ to $P_k$;   /*Action 4*/
 $P_k$
 ▯
 $\neg done_i \wedge OUT_i = \emptyset$              $\longrightarrow$  for each $j \in Incycle_i$ do         /*Action 5*/
                                                           $Incycle_i := Incycle_i - \{j\}$;
                                                           $Cycle\_nodes_i := Cycle\_nodes_i \cup \{j\}$;
                                                           for each $k$ $(1 \leq k \leq n)$ do
                                                               if $(k, j) \in Seen_i$ then
                                                                   $Incycle_i := Incycle_i \cup \{k\}$;
                                                                   $Seen_i := Seen_i - \{(k, j)\}$;
                                                       if $Seen_i = \emptyset$ then
                                                           Declare 'knot detected';
                                                       else
                                                           Declare 'no knot detected';
                                                       $done_i := true$

 ]
```

Figure. 4.4.1 [4]



```
Process $P_k (k \neq i)$: /* algorithm at any node $k$ other than the initiator $i$ */
$has\_seen_k$: boolean(initially $false$);
$OUT_k$: set of node ids(initially the set of all immediate successors of $k$);
$Seen_k$: set of ordered pairs of node ids(initially $empty$);
$Incycle_k$: set of node ids(initially $empty$);/* nodes that are in cycle with $i$ */
$parent_k$: a node id (initially 0);
$done_k$: boolean(initially $false$); /* set to $true$ when $P_k$ sends $parent\_ack$ to its parent */

*[ receive $detect\_knot(i, j)$ from $P_j$  $\longrightarrow$ if $\neg has\_seen_k$ then                /*Action 1*/
                                                   $has\_seen_k := true$;
                                                   $parent_k := j$;
                                                   send $detect\_knot(i, k)$ to all
                                                   nodes in $OUT_k$;
                                               else
                                                   if $Incycle_k \neq \emptyset$ then
                                                       send $cycle\_ack(k)$ to $P_j$;
                                                   else
                                                       send $seen\_ack(k)$ to $P_j$;
 ⫿
 $\neg done_k \wedge$ receive                   $\longrightarrow$ $OUT_k := OUT_k - \{j\}$;                  /*Action 2*/
 $parent\_ack(Seen_j, Incycle_j)$ from         $Seen_k := Seen_k \cup Seen_j$;
 $P_j$                                          if $Incycle_j = \emptyset$ then
                                                   $Seen_k := Seen_k \cup \{(k, j)\}$;
                                               else
                                                   $Incycle_k := Incycle_k \cup Incycle_j$;
 ⫿
 $\neg done_k \wedge$ receive $seen\_ack(j)$ from $P_j$  $\longrightarrow$ $OUT_k := OUT_k - \{j\}$;     /*Action 3*/
                                               $Seen_k := Seen_k \cup \{(k, j)\}$;
 ⫿
 $\neg done_k \wedge$ receive $cycle\_ack(j)$ from $P_j$  $\longrightarrow$ $OUT_k := OUT_k - \{j\}$;    /*Action 4*/
                                               $Incycle_k := Incycle_k \cup \{j\}$;
 ⫿
 $\neg done_k \wedge OUT_k = \emptyset \wedge has\_seen_k$  $\longrightarrow$ if $Incycle_k \neq \emptyset$ then   /*Action 5*/
                                                   $Incycle_k = Incycle_k \cup \{k\}$;
                                               send $parent\_ack(Seen_k, Incycle_k)$ to $parent_k$;
                                               $done_k := true$;
 ]
```

Figure. 4.4.2 [4]

4.4.4 Example Execution Of the Algorithm

In This section we are giving one example to illustrate the working of algorithm. The graph of the system as shown in fig.3.It consists of nodes P1, P2, P3, P4, P5 and P6, the nodes A and B represents other subgraph which may consists of knot in itself. From the figure it is clear that there P1 to P6 nodes form a knot.

Suppose if P1 wants check for knot, it will initiate the algorithm by sending detect_knot messages to its successor in turn to propagate to its successor and so on. The values of variables initially are shown in fig.4. As we can see there is no incoming edge into A or B so nodes P1 to P6 do not any message to them.

The following is one scenario of propagation of the knot detection messages and acknowledgments.

1. P1 sends detect_knot (1, 1) to P2 and P3.
2. P2 receives detect_knot (1, 1) from P1 and sends detect_knot (1, 2) to P4 and P5.
3. P5 receives detect_knot (1, 2) from P2 and sends detect_knot (1, 5) to P6.
4. P4 receives detect_knot (1, 2) from P2 and sends detect_knot (1, 4) to P5 and P6.
5. P3 receives detect_knot (1, 1) from P1 and sends detect_knot (1, 3) to P6.
6. P6 receives detect_knot (1, 3) from P3 and sends detect_knot (1, 6) to P1.
7. P1 receives detect_knot (1, 6) from P6 and sends cycle_ack (1) to P6.
8. P6 receives detect_knot (1, 4) from P4 and sends seen_ack (6) to P4.
9. P6 receives detect_knot (1, 5) from P4 and sends seen_ack (6) to P5.
10. P5 receives detect_knot (1, 4) from P4 and sends seen_ack (5) to P4.
11. P6 receives cycle_ack (1) from P1 and add 1 to its Incycle.
12. P4 receives Seen (5) from P5 and add order pair (4, 5) to its Seen set.
13. P5 receives Seen (6) from P6 and add order pair (5, 6) to its Seen set.
14. P4 receives Seen (6) from P6 and add order pair (4, 6) to its Seen set.
15. P5 sends parent_ack ({5, 6}, {}) to P2.
16. P6 adds itself to Incycle and sends parent_ack ({}, {1, 6}) to P3.
17. P2 receives parent_ack({5, 6}, {}) from P5 adds (5, 6) and (2, 5) to its Seen
18. P4 sends parent_ack ({(4, 6), (4, 5)}, {}) to P2.
19. P3 receives parent_ack ({}, {1, 6}) from P6 adds 1, 6 to Incycle.
20. P2 receives parent_ack ({(4, 6), (4, 5)}, {}) from P2 and adds (4, 6), (4, 5) and (2, 4) to Seen.
21. P3 adds itself to Incycle sends parent_ack ({}, {1, 3, 6}) to P1.
22. P1 receives parent_ack ({}, {1, 3, 6}) from P3 adds 1, 3 and 6 to Incycle.
23. P2 sends parent_ack ({(5, 6), (2, 5), (4, 6), (4, 5), (2, 4)}, {}) to P1.
24. P1 receives parent_ack ({(5, 6), (2, 5), (4, 6), (4, 5), (2, 4)}, {}) from P2 adds (5, 6), (2, 5), (4, 6), (4, 5), (2, 4) and (1, 2) to Seen.

The contents of local variables after step 24 are as shown in fig. 5.

As now P1 has received acknowledgement from all its successor it will initiate action 5 and for each j in Incycle, P1 deletes j from Incycle, adds j to Cyclenodes and then for each k such that (k, j) is in Seen, it will remove (k, j) from Seen and add k to Incycle. After end of for loop eventually Seen = Ø and Cyclenodes = {1, 2, 3, 4, 5, 6} and P1 declares detection of knot and nodes in Cyclenodes are the nodes which are in knot with P1.

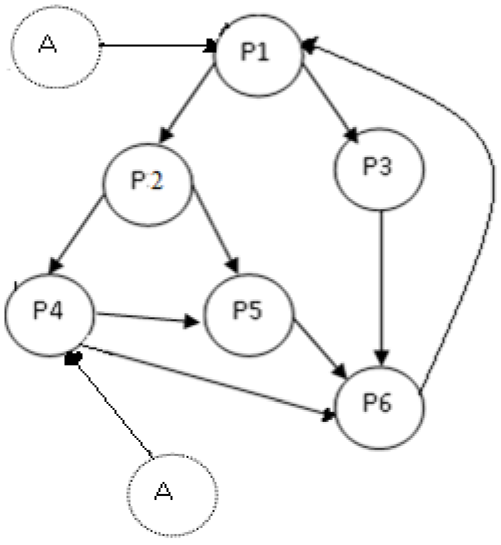


Figure. 4.4.3

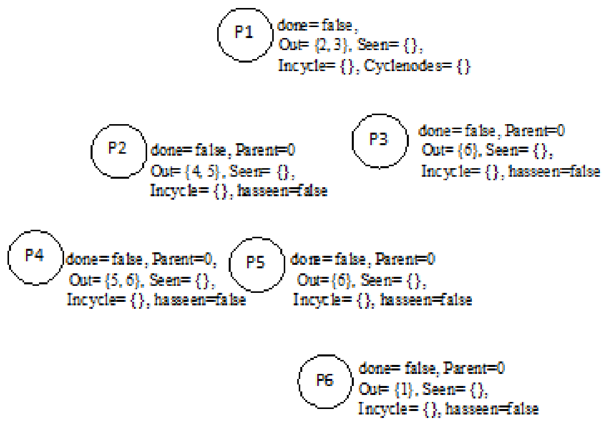


Figure. 4.4.4

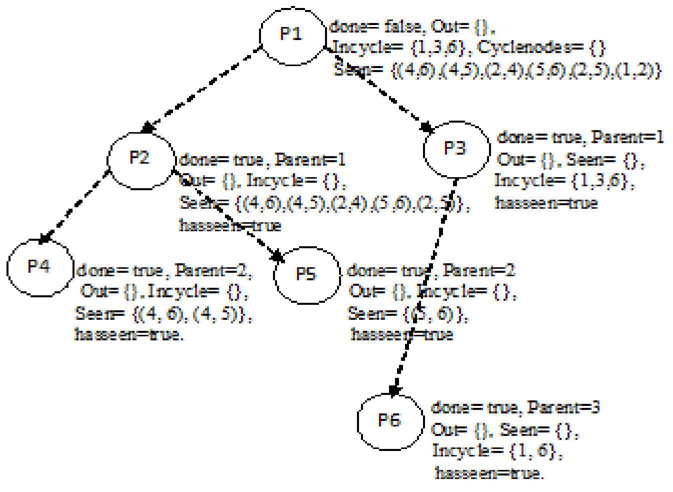


Figure. 4.4.5

4.4.5 Discussion:

In this section we compare algorithm discussed above with previous deadlock detection algorithm.

1. Mishra and Chandy's [7] algorithm requires message complexity of 4e to detect the knot which is much higher than above algorithm having 2e, also Mishra and Chandy's algorithm only says whether node is involved in deadlock and do not give information about which are other nodes in the knot.

2. Cidon's knot detection algorithm [6] requires message complexity of O(n log(n) + e) where as above algorithm requires it as 2e where e is the number of edges in the subgraph consisting of the nodes reachable from the initiator.

3. Boukerche and Tropper [5] algorithm finds knot in the graph with message complexity of 2e which is same as above algorithm but like others it also does not say anything about which other nodes are involved in the knot. This information can be useful for deadlock resolution.

Thus knot detection algorithm is important problem in distributed system, such as store-and-forward networks, distributed simulation, and distributed databases. In this paper we have studied the algorithm which requires message complexity of 2e (where e is the number of edges in the subgraph consisting of the nodes reachable from the initiator) to detect whether there is the knot in the system and if there is knot then it will also detect which are the other nodes involved in the knot with the initiator. If there is no knot and initiator is in the cycle then it will give which other nodes are in the cycle with initiator. This information will be helpful in case of AND request model to determine which nodes are in the cycle.

5. Conclusion

Thus deadlock detection and also resolution algorithm, is important problem in distributed system. In this paper we had studied 4 different deadlock detection algorithms [1], [2], [3], [4]. In this conclusion part we are comparing them by their message complexity (no of message transferred for detecting the deadlock), time delay (no of message hops require to detect the deadlock), and also putting advantage of each of them.

Ref No.	Message Complexity	Time Delay	Advantage
[1]	2n – 1 = O (n), n is the no of nodes in the cycle/system	O (n)	Utilizes all the messages sent by all the nodes in a controlled manner and its quite efficient as per its message complexity and latency is concerned.
[2]	O (n log n)		No extra cycle is required to break deadlock. Communication cost very low of the worst case order of O (n log n).
[3]	2e, e is the no of edges	2d, d is the diameter of the system	This algorithm woks for the p-out-of-q resource request model and also gives a strategy to resolve it from information transferred for detection only.
[4]	2e (e is the number of edges in the reachable part of the distributed graph)	2 (d+1) (d is diameter)	This algorithm not only detects the knot in the subgraph but also give which all nodes are involved in the knot, whose information will be helpful for deadlock resolution.

References

[1] Prieto, M.; Villadangos, J.; Farina, F.; Cordoba, A.; , "An O(n) distributed deadlock resolution algorithm," Parallel, Distributed, and Network-Based Processing, 2006. PDP 2006. 14th Euromicro International Conference on , vol., no., pp. 8 pp., 15-17 Feb. 2006
DOI: 10.1109/PDP.2006.22
http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1613253&isnumber=33865

[2] Cordoba, A.; Farina, F.; Garitagoitia, J.R.; de Mendivil, J.R.G.; Villadangos, J.; , "A low communication cost algorithm for distributed deadlock detection and resolution," Parallel, Distributed and Network-Based Processing, 2003. Proceedings. Eleventh Euromicro Conference on , vol., no., pp. 235- 242, 5-7 Feb. 2003
DOI: 10.1109/EMPDP.2003.1183594
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1183594&isnumber=26558

[3] Srinivasan, S.;Vidya, Rajan; Rajaram, Ramasamy; ”An Optimal, Distributed Deadlock Detection and Resolution Algorithm for Generalized Model in Distributed Systems”; Book series: “Communications in Computer and Information Science”,2009; Publisher: Springer Berlin Heidelberg
DOI: 10.1007/978-3-642-03547-0
URL: http://www.springerlink.com/content/r6606138n9220302

[4] Manivannan, D.; Singhal, M.; , "An efficient distributed algorithm for detection of knots and cycles in a distributed graph," Parallel and Distributed Systems, IEEE Transactions on , vol.14, no.10, pp. 961-972, Oct. 2003
DOI: 10.1109/TPDS.2003.1239865
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1239865&isnumber=27797

[5] A. Boukerche and C. Tropper, “A Distributed Graph Algorithm for the Detection of Local Cycles and Knots,” IEEE Trans. Parallel and Distributed Systems, vol. 9, no. 8, pp. 748-757, Aug. 1998.

[6] I. Cidon, “An Efficient Distributed Knot Detection Algorithm,” IEEE Trans. Software Eng., vol. 15, no. 5, pp. 644-649, May 1989.

[7] J. Misra and K. Chandy, “A Distributed Graph Algorithm: Knot Detection,” ACM Trans. Programming Languages and Systems, pp. 678-686, 1982.