International Journal of Innovative Computing, Information and Control
Volume **13**, Number **2**, April **2017**

ICIC International ©2017 ISSN 1349-4198
pp. 703–710

AN EFFICIENT DEADLOCK DETECTION AND RESOLUTION ALGORITHM FOR GENERALIZED DEADLOCKS

Wei Lu, Chengkai Yu, Weiwei Xing, Xiaoping Che and Yong Yang

School of Software Engineering
Beijing Jiaotong University
No. 3, Shangyuancun, Haidian District, Beijing 100044, P. R. China
{ luwei; wwxing; xpche; 12112088; 14121684 }@bjtu.edu.cn

Received September 2016; revised January 2017

Abstract. *In existing deadlock detection and resolution algorithms, only a few of them can handle deadlocks in generalized request models. Most of these algorithms diffuse probes and collect dependency information in the replies. In this paper, we propose an efficient algorithm to detect and resolve generalized deadlocks. In this algorithm, replies are sent after a process has received all the messages from its predecessors. This mechanism reduces both the quantity and total size of the messages. The message number of our proposed algorithm is $O(e+n)$. And the experiment shows that our algorithm has better performance in concurrent executions.*

Keywords: Generalized deadlock, Deadlock detection, Deadlock resolution, Distributed system, Wait-for graph

1. Introduction.

In distributed systems, a process may request resources from other processes. Deadlock is defined as a state in which two or more processes wait for the other to finish. Processes involved in a deadlock wait for requested resources to be granted infinitely. Thus, deadlock prevents processes from releasing resources and obstructs the progress of program. Therefore, deadlocks should be resolved promptly in distributed systems.

Deadlock can be modeled by a directed graph called wait-for graph (WFG) [11]. Each process in a distributed system is defined as a vertex in the WFG. And dependences are defined as directed edges in the corresponding WFG. According to the relationship between edges, WFG can be classified into many request models, such as *AND model* [12], and *OR model* [17]. In this paper, we focus on a more generalized model called *P out-of Q model* [3]. P out of Q model also can be called *generalized model* [15]. In this model, a process which makes requests for Q resources remains blocked until it receives any P out of Q resources.

In this paper, we introduce a new centralized algorithm to detect and resolve deadlocks in distributed systems. Our algorithm diffuses probe messages and collects request conditions in the WFG. We count probe messages which come from the predecessors. However, our algorithm is different from other algorithms in the following aspects.

1) The proposed algorithm only needs $e+n$ messages in a single execution, while others need $e+2n$, $2e$, $4e$, etc.
2) Our algorithm reduces the number of request conditions that are sent repeatedly in concurrent executions.
3) Our algorithm can adapt to more complicated network environment where messages may not be received as the order of sending them.

Many deadlock detection algorithms have been proposed to solve AND and OR models [1, 6, 13, 14]. However, only a few algorithms can detect generalized distributed deadlocks [10, 15, 16]. Most of the mentioned algorithms collect request conditions to build a WFG through propagating messages, then evaluate and reduce the request conditions to determine whether the system has a deadlock. In [8], Kshemkalyani and Singhal propose a one-phase algorithm which takes $2d + 2$ time units and needs $2e$ messages. In Chen et al.'s algorithm [5], the initiator collects request conditions from the other processes and expands the WFG when receiving a message until the WFG becomes steady. The initiator is the process that initiates the detection algorithm when potential deadlock exists. Their algorithm needs only $2n$ messages, and the delay is $2d$ time units. In Lee's algorithm [9], the request conditions are dispatched separately to the successors and sent to the initiator at leaf nodes in the induced spanning tree, whereas in [8] they are sent to the predecessors. Lee et al. decrease the message size and delay to $O(d)$ and $d + 2$ respectively. And it only needs less than $2e$ messages in a single execution. These algorithms send request conditions or the identifier set of dependent processes multiple times except Chen et al.'s algorithm in [5]. In [15], Srinivasan and Rajaram propose a new algorithm which sends request condition only one time, although the message size is $O(n)$. What is more, it reduces the message number to $e + 2n$.

The rest of our paper is arranged as follows. We describe the underlying system models and give the definition of deadlocks in Section 2.1. Then, we explain the procedure of the proposed algorithm for both a single execution and concurrent executions in Section 2. After that, we compare the performance between existing algorithms and ours experimentally and theoretically in Section 3. Section 4 concludes this paper and proposes some future works.

2. **The Proposed Algorithm.**

In this section, we give a description of the proposed deadlock detection algorithm. Firstly, we describe a single algorithm instance following with the formal description of it. Then we discuss concurrent executions of our proposed algorithm.

2.1. **System model.**

Figure 1 is the event trace diagram of our system model. There are three processes. When P_2 requests R_1 from P_1, it sends a REQUEST to P_1 and P_1 grants the resource to P_2. P_1 sends a RELEASE after it has released R_1. These REQUEST and RELEASE messages are computational messages which are sent within normal executions. In addition, CANCEL also is computational message. These processes request resources from each other and become deadlocked. P_3 initiates deadlock detection by sending a PROBE to P_1. P_1 diffuses PROBE to its successors and responds REPORT to P_3. These

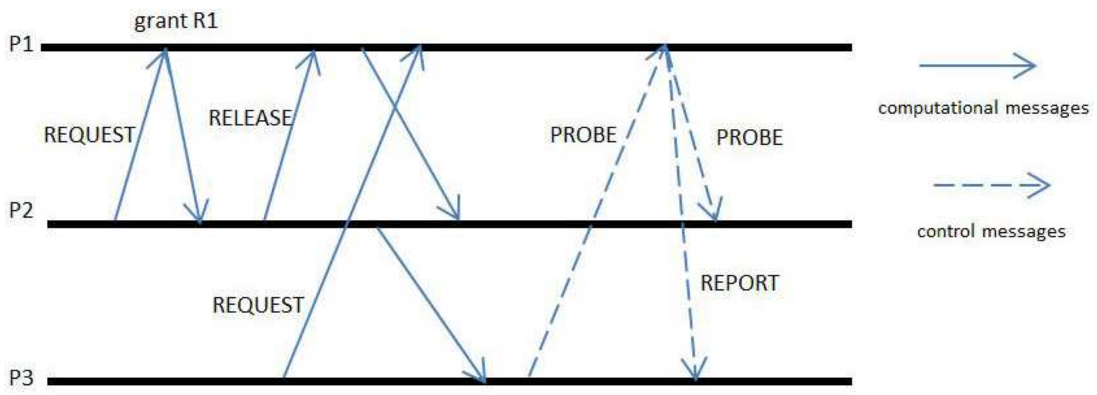

FIGURE 1. Event trace diagram of system model

PROBE and REPORT messages are control messages transmitted in deadlock detection. Mostly, our system model is the same as described in [7, 15]. Each process knows which processes it depends on. A process blocks when it sends a request, and it does not send any computational messages until it becomes unblocked. When the system is deadlocked, all the processes are directly deadlocked.

2.2. **Algorithm description for a single execution.** The proposed algorithm consists of probing phase and detection and resolution phase. In the system, each process maintains two sets, IN and OUT, and only knows its own request condition before the execution of the proposed algorithm. When the system is supposed to have a deadlock, one of these processes which is called *initiator* will start deadlock detection by propagating weighted *probes* to its successors. When a blocked process receives a probe for the first time, it joins in this instance of the algorithm and initiates a local counter of the probes which have been received, which is described in Algorithm 1. Then it transmits the probe to its successors with new weight. If the process has already joined the instance of the algorithm, it accumulates the weight in the probe instead of propagating it. When the counter is equivalent to the size of the set IN, the process sends a *report* directly to the initiator. The initiator accumulates the weight when it has received a message and adds the request condition to a set RC. As shown in Algorithm 3, the probing phase is finished when the accumulated weight is equivalent to one. To deal with phantom edges, it is necessary to check whether the sender of REPLY is in set IN. Every phantom edge will be sent to the initiator with the report message finally. And occurrence of phantom edges should be set to *true* in each request condition.

We use weight to determine when all the messages have been received and the probing phase is finished [7]. The initiator dispatches a weight of one to its successors evenly through probes. The other processes not only distribute the received weight to their own successors, but also leave a part of weight to themselves. That is the main difference between our algorithm and the others. The remaining weight is sent to the initiator with the request condition through a report. The initiator collects all the weights and starts deadlock detection when they sum up to one. Algorithm 2 describes the reduction of request conditions.

2.3. **Formal algorithm description for a single execution.** The following is a formal description of the proposed algorithm which is executed at a process i.

Data structures at a process i: (It is the initial value in the parentheses.)

- OUT_i: **set of integer**, the set of process identifiers which process i is waiting for;
- IN_i: **set of integer**, the set of process identifiers which are waiting for process i;
- P_i: **set of edge**$(\emptyset)$, the set of phantom edges;
- $weight_i$: **float**(0), the weight value which is accumulated locally before report is sent at a process i;
- rc_i: **string**, the request condition of process i;
- $first\_recv_i$: **boolean**$(true)$, the flag that determines whether process i has received a probe;
- cur_init_i: **integer**(-1), the identifier of the current initiator;
- $probe\_recved_i$: **integer**(0), the number of probes that process i has received.

Additional data structures at initiator: (It is the initial value in the parentheses.)

- $weight_{init}$: **float**(0), the weight value accumulated from messages, either probes or reports;
- RC_{init}: **set of request conditions**$(\emptyset)$, the request conditions collected from the reports.

Message formats:

- $PROBE(i, w)$: This is a message that diffuses the weight to successors where i is the identifier of the initiator and w the weight;
- $REPORT(rc, P, w)$: It is used to send the weight value back to the initiator attached with the request condition. The rc is the request condition of the current process, the P is the set of phantom edges and the w is the accumulated weight.

Algorithm 1 Receiving a $PROBE(init, w)$ from process s at a non-initiator process i

```
if $first\_recv = true$ then
    for $j \in OUT_i$ do
        send $PROBE(init, w/|OUT_i + 1|)$ to $j$;
    end for
    $cur\_init_i \leftarrow init$;
    $weight_i \leftarrow 0$;
    $probe\_recved_i \leftarrow 0$;
end if
$weight_i \leftarrow weight_i w/|OUT_i + 1|$;
$probe\_recved_i \leftarrow probe\_recved_i + 1$;
if $s \notin IN_i$ then
    $P_i = P_i \bigcup \{s \rightarrow i\}$;
end if
if $probe\_recved_i = |IN_i|$ then
    send $REPORT(rc_i, P_i, weight_i)$ to $cur\_init_i$;
end if
```

Algorithm 2 Definition of $deadlock\_detection(RC, P)$

```
for $\{a \rightarrow b\} \in P$ do
    get $rc_a$ from $RC$ marked as $rc$;
    modify every occurrence of $b$ in $rc$ to $true$;
    replace $rc_a$ in $RC$ with $rc$;
end for
$tmpA \leftarrow \emptyset$;
repeat
    $A \leftarrow tmpA$;
    for $tmprc_i \in RC$ do
        if $eval(tmprc_i) = true$ then
            $tmpA \leftarrow tmpA \bigcup \{i\}$;
            $RC \leftarrow RC - \{tmprc_i\}$;
        end if
    end for
until $A = tmpA$
if $RC = \emptyset$ then
    declare no deadlock;
else
    declare deadlock;
    $resolution(A, RC)$;
end if
```

Algorithm 3 Receiving a $REPORT(rc, P, w)$ at the initiator process i

```
$weight_{init} \leftarrow weight_{init} + w$;
$RC_{init} \leftarrow RC_{init} \bigcup rc$
$P_i \leftarrow P_i \bigcup P$
if $weight_{init}$=1 then
    execute $deadlock\_detection(RC_{init}, P_i)$;
end if
```

2.4. **Algorithm description for concurrent executions.** Since multiple processes may get blocked at the same time, several instances of the deadlock detection algorithm may be invoked, that is, a process may join in two or more instances. Thus, many duplicated massages will be sent in probing phase and unnecessary victims may be selected in the resolution procedure. Many strategies have been proposed to deal with these issues. We choose the priority based method to solve this problem because it is easy to implement and has better performance. Each initiator has a priority which is diffused within PROBE messages. We follow the definition of priority in [9]. A priority is a tuple marked as (t, s). t is a time stamp when the blocked request is made and s is the sequence number of the initiation for that request. For $p_1 = (t_1, s_1)$ and $p_2 = (t_2, s_2)$, p_1 is higher than p_2 if $t_1 < t_2$ or $t_1 = t_2$ and $s_1 < s_2$. Each process sets its priority as soon as receiving its first message. Processes only transmit messages with higher priority than their own. When a process receives a message with higher priority, it resets its own priority and joins this execution of the algorithm.

3. **Performance Comparison.** In this section, we will compare the complexity of the proposed algorithm in three aspects: message number, message size and execution delay. We consider the worst case for a single execution as previous papers do. Also we do not consider some abnormal scenarios, such as phantom edges, in the analysis. Then, we will show the experimental results of our algorithm at the end of this section.

We follow the assumptions which are commonly used in the existing researches.

1) It takes one time unit for each message to transmit between any two processes.
2) When a process receives a message, it can complete local computation instantly.

The following is some notations used for analyzing the performance of the algorithm:

1) e: the number of edges in the WFG;
2) n: the number of processes in the WFG;
3) d: the diameter of the spanning tree built by the propagation of messages.

The message number is defined as the number of messages that are sent during a single execution of the algorithm. The message size is measured as the number of identifiers in a message. The delay is the time interval between the initiation and the end of the detection and resolution phase.

There are two types of messages, PROBE and REPORT, transmitted while the algorithm is executing. When the algorithm is finished, only the first probe is dispatched. Thus, the total number of PROBE messages is equal to e. From Algorithm 1, each process sends only one REPORT to the initiator. Thus, the total number of REPORT messages is equal to $n - 1$. Therefore, the message number of the proposed algorithm is $e + n - 1$.

As is shown in Algorithm 1, each process sends only one PROBE message to the initiator. That is, the request conditions are transmitted only one time. As a process may depend on the other $n - 1$ processes, a request condition has $n - 1$ process identifiers in the worst case. Therefore, the message size of the proposed algorithm is $O(n)$.

The delay of the algorithm depends on the time that the propagation of the PROBE messages costs. Let $d_{\max} \leqslant d$ be the maximum distance from the initiator in the induced spanning tree. Then, it takes $d_{\max} + 1$ time units for the PROBE to transit from the initiator to that furthermost process, marked as $p_{d\_\max}$. Since there is no path longer than that from the initiator to $p_{d\_\max}$, when $p_{d\_\max}$ receives this PROBE, it does not need to wait for other PROBE messages from its predecessors. That is it cannot send a REPORT immediately. Therefore, the execution of the proposed algorithm terminates in $d+2$ time units without interrupted.

Table 1 shows a summary of the performance of existing algorithms and our proposed algorithm. Our algorithm only needs $e+n$ messages to detect a deadlock in a single execution which is better than most of the others. In [5], their algorithm needs $2n$ messages but its delay is approximate to twice of ours. Message size means the number of process identifiers in a message. Although some algorithms only need constant message size, they need more delay and messages. In [8, 9], request conditions are sent along the path of propagating probes. The message carries $n-1$ request conditions in the worst case where each process has only one successor. However, in our proposed algorithm every message carries constant request condition.

Table 1. Single execution performance comparison for existing algorithms

Algorithms	Message number	Delay	Message size	Resolution	Type
Bracha [2]	$4e$	$4d$	$O(1)$	none	distributed
Wang [18]	$6e$	$3d+1$	$O(1)$	none	distributed
Kshemkalyani [7]	$4e-2n+2l$	$2d$	$O(1)$	e messages	distributed
Chen [5]	$2n$	$2d$	$O(n)$	$3n$ messages	centralized
Brzezinski [4]	$n^2/2$	$4n$	$O(n)$	none	distributed
Kshemkalyani [8]	$2e$	$2d+2$	$O(e)$	none	distributed
Lee [9]	$< 2e$	$d+2$	$O(d)$	1 message	centralized
Srinivasan [15]	$e+2n$	$d+2$	$O(n)$	1 message	centralized
Proposed	$e+n-1$	$d+2$	$O(n)$	1 message	centralized

e is the number of edges in the WFG, d is the diameter of the spanning tree, n is the number of processes, and l is the number of leaf nodes in the tree.

In this section, we analyze the performance of some algorithms in concurrent executions with experiment. We compare the performance of our proposed algorithm with Kshemkalyani's [7, 8], Lee's [9], and Srinivasan's [15] in terms of the number of messages sent in a single execution and the total message size. In this paper we only consider the size of request condition in each message, because it changes while detecting different deadlocks, and it can reflect the total message size more typically.

In the simulation, the programs are written in JAVA. As the request phase before a deadlock happens does not inflect the performance of the deadlock detection algorithm, we input a WFG to the programs at the beginning of the simulation instead of simulating the procedure of deadlock happening. This strategy makes the simulation more efficient. The WFG is auto generated randomly. And each algorithm detects deadlocks in the same WFG. We provide two types of WFGs with different number of request conditions in the simulation. Each process in WFG of type A has $n-1$ successors, which is the most complicated WFG. In type B, each process only depends on $n/2$ processes. While the program is running, we record the messages sent by the deadlock detection algorithm. After they are finished, we run another program to analyze the messages. All the instances of the algorithm are run for 100 times in deferent WFG. And we take the mean value as the final result.

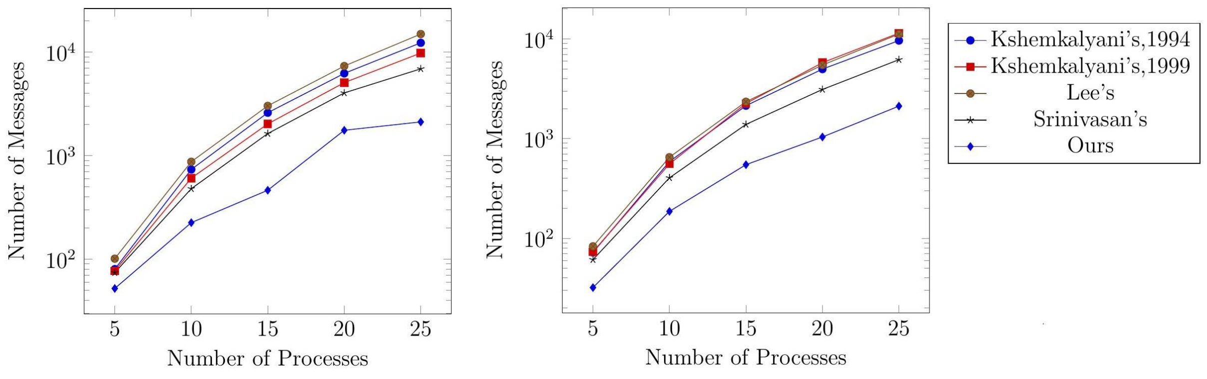

FIGURE 2. Number of messages for WFG type A (left) and type B (right)

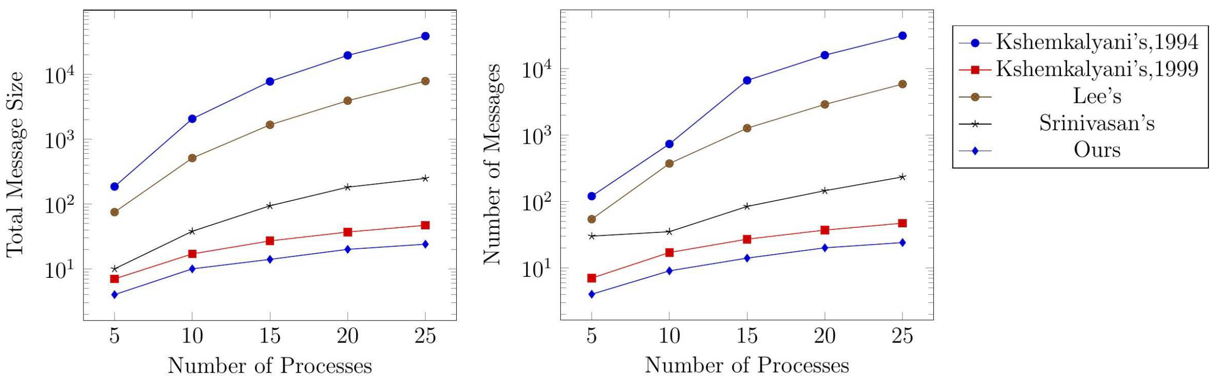

FIGURE 3. Total message size for WFG type A (left) and type B (right)

Figure 2 shows that our algorithm sends less messages than the other four algorithms which conforms with the theoretical result in Table 1. Although in Table 1 the message size in Kshemkalyani's algorithm [8] is larger than that of Srinivasan's algorithm [15], the experiment shows that Kshemkalyani's algorithm is better than Srinivasan's in terms of total message size in concurrent executions. Because in [15] request condition is sent as soon as the process receives a probe message, but in [8] it is sent when the process receives all the expected replies. Thus, the sending procedure in [8] can be put off by a higher instance while in [15] request condition is sent in different instances. And our proposed algorithm adopts similar mechanism to prevent the sending of request condition repeatedly. Figure 3 denotes that our algorithm sends smaller messages.

4. **Conclusions.** In this paper, we propose a new deadlock detection algorithm for generalized deadlocks. Our algorithm consists of two phases. In the first phase, each process diffuses probe messages to its successors recursively. Also they send request conditions to the initiator when they have received all the probe messages from their predecessors. The second phase begins when the weight accumulated at the initiator becomes one. In the second phase, the initiator performs the detection algorithm on the collected request conditions. After that, if a deadlock is found, the initiator executes the resolution algorithm to resolve the deadlock. We describe our proposed algorithm in Section 2 formally. Our algorithm has a time complexity of $d+2$ in the worst case where d is the diameter of the spanning tree. The message number is $e+n$ where e is the number of edges and n is the number of processes in the WFG. And the message size is n. The experiment shows that our algorithm has a better performance in terms of message number and total massage size in concurrent executions than the existing algorithms.

Although our proposed algorithm needs less messages among the probe-based algorithms in a single execution, it sends repeating messages in concurrent executions. In the future, we can focus on reducing the number of messages in concurrent executions.

Acknowledgements. This work was supported in part by the National Natural Science Foundation of China (No. 61100143, No. 61272353, No. 61370128, No. 61428201, No. 61502028), Program for New Century Excellent Talents in University (NCET-13-0659), Beijing Higher Education Young Elite Teacher Project (YETP0583).

REFERENCES

[1] 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, 1998.

[2] G. Bracha and S. Toueg, A distributed algorithm for generalized deadlock detection, *Proc. of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, pp.285-301, 1984.

[3] G. Bracha and S. Toueg, Distributed deadlock detection, *Distributed Computing*, vol.2, no.3, pp.127-138, 1987.

[4] J. Brzezinski, J.-M. Helary, M. Raynal and M. Singhal, Deadlock models and a general algorithm for distributed deadlock detection, *Journal of Parallel and Distributed Computing*, vol.31, no.2, pp.112-125, 1995.

[5] S. Chen, Y. Deng, P. Attie and W. Sun, Optimal deadlock detection in distributed systems based on locally constructed wait-for graphs, *Proc. of the 16th International Conference on Distributed Computing Systems*, pp.613-619, 1996.

[6] Y. M. Kim, T. H. Lai and N. Soundarajan, Efficient distributed deadlock detection and resolution using probes, tokens, and barriers, *Proc. of International Conference on Parallel and Distributed Systems*, pp.584-591, 1997.

[7] A. D. Kshemkalyani and M. Singhal, Efficient detection and resolution of generalized distributed deadlocks, *IEEE Trans. Software Engineering*, vol.20, no.1, pp.43-54, 1994.

[8] A. D. Kshemkalyani and M. Singhal, A one-phase algorithm to detect distributed deadlocks in replicated databases, *IEEE Trans. Knowledge and Data Engineering*, vol.11, no.6, pp.880-895, 1999.

[9] S. Lee, Fast, centralized detection and resolution of distributed deadlocks in the generalized model, *IEEE Trans. Software Engineering*, vol.30, no.9, pp.561-573, 2004.

[10] W. Lu, Y. Yang, L. Wang, W. Xing and X. Che, A novel concurrent generalized deadlock detection algorithm in distributed systems, *Algorithms and Architectures for Parallel Processing*, pp.479-493, 2015.

[11] D. Menasce, R. R. Muntz et al., Locking and deadlock detection in distributed data bases, *IEEE Trans. Software Engineering*, no.3, pp.195-202, 1979.

[12] D. Mendívil, J. R. González, F. Fariña, J. R, Garitagotia, C. F. Alastruey and J. M. Bernabeu-Auban, A distributed deadlock resolution algorithm for the and model, *IEEE Trans. Parallel and Distributed Systems*, vol.10, no.5, pp.433-447, 1999.

[13] W. K. Ng and C. V. Ravishankar, On-line detection and resolution of communication deadlocks, *Proc. of the 27th Hawaii International Conference on System Sciences*, vol.2, pp.524-533, 1994.

[14] M. Raynal, Simple deadlock detection for the and-communication model, *The 8th International Conference on Complex, Intelligent and Software Intensive Systems*, pp.273-278, 2014.

[15] S. Srinivasan and R. Rajaram, An improved, centralised algorithm for detection and resolution of distributed deadlock in the generalised model, *International Journal of Parallel, Emergent and Distributed Systems*, vol.27, no.3, pp.205-224, 2012.

[16] Z. Tao, H. Li, B. Zhu and Y. Wang, A semi-centralized algorithm to detect and resolve distributed deadlocks in the generalized model, *IEEE the 17th International Conference on Computational Science and Engineering*, pp.735-740, 2014.

[17] J. Villadangos, F. Fariña, D. Mendívil, J. R. González, J. R. Garitagoitia and A. Córdoba, A safe algorithm for resolving or deadlocks, *IEEE Trans. Software Engineering*, vol.29, no.7, pp.608-622, 2003.

[18] J. Wang, S. Huang and N. Chen, *A Distributed Algorithm for Detecting Generalized Deadlocks*, Technical Report, Department of Computer Science, National Tsing-Hua University, 1990.