ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Rainer Manthey and Viacheslav Wolfengagen (Eds)

Advances in Databases and Information Systems 1997

Proceedings of the First East-European Symposium on Advances in Databases and Information Systems, (ADBIS'97), St Petersburg, 2-5 September 1997

On Time Complexity of Distributed Algorithms for Generalized Deadlock Detection

Jerzy Brzezinski

Published in collaboration with the
British Computer Society

On Time Complexity of Distributed Algorithms for Generalized Deadlock Detection

Jerzy Brzeziński*

Institute of Computing Science, Poznań University of Technology
Poznań, Poland

Abstract

Deadlock detection in distributed asynchronous systems - such as distributed database systems, computer networks, massively parallel systems etc. - is peculiarly subtle and complex. This is because asynchronous systems are characterized by the lack of global memory and a common physical clock, as well as by the absence of known bounds on relative processor speeds and transmission delays. These difficulties imply also problems with performance analysis of distributed algorithms for deadlock detection.

This paper deals with worst-case one-time complexity analysis of two well known distributed algorithms for generalized deadlock detection. The time complexity is expressed as a function of the diameter d and the longest path l of the wait-for-graph (*WFG*) characterizing a state of distributed system. First, the algorithm proposed by Bracha and Toueg is considered. It is shown that its time complexity is of $2d + 2l$. Then, we prove that the time complexity of Kshemkalyani and Singhal algorithm is of $(d + 1) + l$.

1 Introduction

Deadlock handling is a very important problem in various applications, including information and database systems, computer networks, massively parallel systems etc. The problem is well defined and understood in centralized environments. However, in asynchronous distributed systems the problem remains peculiarly subtle and complex. This is because asynchronous system are characterized by the lack of global memory and a common physical clock, as well as by the absence of know bounds on relative processor speeds and message delays. As a result no node has accurate knowledge of the whole system state. These complexities brought about significant errors in a number of published distributed deadlock detection algorithms.

Informally, deadlock refers to situation in which some transactions (processes) request resources (services) from each other and then wait indefinitely for these requests to be satisfied, thus the progress of their execution is definitely halted. In this case, the execution can turn out to be completely useless unless proper and careful control is exercised.

In deadlock detection approach, messages are sent (i.e., resources are granted) without any constraints. However, the state of the system is checked periodically, or when a deadlock is suspected, to determine if a set of processes is deadlocked. This checking is performed by a deadlock detection algorithm. If a deadlock is discovered, recovery from it is done by aborting one or more deadlocked processes.

The suitability of a deadlock handling approach greatly depends on the environment characteristics and the assumed model of transaction (process) requests (see e.g., [16], [11]). The simplest possible request model is one in which a process can require at most one message (resource, lock) at a time. In the *AND* model (also known as the *resource model*), processes are permitted to request simultaneously a set of messages. A process cannot execute until it acquires *all* messages (resources) for which it is waiting ([5], [6]). This model represents, for instance, possible requests of transactions to lock several data items. Another model of requests is the *OR*

*This work was partially supported by the State Committee for Scientific Research (KBN), Poland, under grant for Research and Academic Network in Poland - DS/97.

model (also called the *communication model*), where a process requests messages from a set of processes and it can proceed only after it has received acknowledge from *any* one of the processes it is waiting for. Examples of this model include alternative control structures in programming languages like CSP and ADA, as well as replicated database systems, where a read request for a replicated data item can be satisfied by reading any copy of it ([6]).

In the *k out of n* model, processes are permitted to request any *k* messages from a set of *n* processes ([1]). This model is a generalization of *OR* and *AND* models since an *OR* request corresponds to *1 out of n*, and an *AND* request corresponds to *n out of n* one. A sample application of the *k out of n* model is replicated database systems, where a quorum-based replica control algorithm is used to preserve database consistency ([7]). In such an algorithm, a transaction that wants to read a replicated data item must read *r* copies out of *n*, and in order to write a replicated data, a transaction must write *w* copies out of *n*, where $r + w > n$ and $2w > n$. The most general model is *OR - AND* model (disjunctive *k out of n*). It allows a transaction to specify resource requests as a predicate expressed by any combination of logical “and” and “or” operators.

The above request models classify distributed deadlock detection algorithms according to the complexity of the resource requests they permit. Several algorithms have been proposed for *one resource* and *AND* models ([5], [6], [14]) as well as for *OR* model ([6],[15]). Distributed deadlock detection algorithms for the *k out of n* model have been developed by Bracha and Toueg ([1]) as well as by Kshemkalyani and Singhal ([13]). These algorithms can be extended to cope with generalized (*AND - OR*) deadlocks, and thus to be comparable with the solutions proposed in ([8], [2]).

Many of the above mentioned algorithms have been presented, carefully analyzed and compared in two surveys by Knapp ([11]), and Singhal ([16]). These surveys and recent results ([2], [3], [12], [13]) confirm that distributed deadlock detection problem remains difficult, and performance analysis of deadlock handling algorithms still offers opportunity for improvement. The latter arises as available performance analysis seems to be incomplete, ambiguous, and thus of a limited applicability. It is because performance evaluation of distributed deadlock detection is itself difficult, and in literature different assumptions are made concerning: distributed environment characteristics (e.g. FIFO or non FIFO channels), acceptable request models (e.g. *AND*, *OR*, *k out of n*, generalized *AND-OR*), formulation of deadlock detection problem (detection of a deadlock occurrence, detection of a deadlocked process, detection of a deadlocked set, and detection of the maximum deadlocked set - see e.g. [2],[3]), selected performance characteristics (e.g. time complexity, message complexity, memory complexity, failure resilience), and inputs of performance function (e.g. communication graph parameters, wait - for - graph parameters).

The rest of the paper is organized as follows. In Section 2 fundamental notions, models and problems concerning distributed deadlock detection have been introduced. In Section 3 we present Bracha and Toueg distributed algorithm for generalized deadlock detection, consider an example of its application, and then analyze performance of the algorithm in terms of worst-case one-time complexity. In similar way we discuss in Section 4 the algorithm proposed by Kshemkalyani and Singhal. Finally, Section 5 contains concluding remarks and indicates further lines of investigations.

2 Basic Definitions and Problem Formulation

First, we state the deadlock detection problem formally. To do this we denote by **P** the set of processes $\{P_1, P_2, \dots, P_N\}$ composing a distributed application. Processes communicate by asynchronous message passing through fully connected communication network. We assume that there is a one-to-one correspondence between nodes of the communication network and processes, and that the assignment is static.

Communication is assumed to be asynchronous. Thus, the process sending a message does not wait for the delivery of the message to be complete. Each message sent by a process running on a node is handed over to the underlying system. This system delivers the message to the destination node and puts it in a local buffer of that node (the message is then said to have *arrived*). This message can then be extracted from the buffer when the application process requires it (the message is then said to have been *consumed*).

At any time, a process is either *active* or *passive*. Only active processes can send and consume messages. An

active process can spontaneously become passive requiring some messages (resources).

Associated with a passive process P is its *dependent set* $\mathbf{DS}$, the set of processes from which P is expecting to receive messages. By definition, a process is not a member of its dependent set. For example, in the *OR-AND* model, the dependent set of a passive process is defined as $\mathbf{DS}_1 \cup \mathbf{DS}_2 \cup \dots \cup \mathbf{DS}_q$, where $\mathbf{DS}_i \subseteq \mathbf{P}$ for all i . The process becomes active only after a message from *every* process in $\mathbf{DS}_1$, *or* a message from *every* process in $\mathbf{DS}_2$, *or...*, *or* a message from *every* process in $\mathbf{DS}_q$ has arrived.

In order to abstract the activation condition of a passive process P with dependent set $\mathbf{DS}$, a predicate called *fulfilled*($\mathbf{A}$) is applied, where $\mathbf{A}$ is a subset of $\mathbf{P}$. Predicate *fulfilled*($\mathbf{A}$) is *true* if messages arrived from all processes belonging to set $\mathbf{A}$ are sufficient to activate process P (since P is passive, arrived messages are not consumed). Moreover, *fulfilled*($\emptyset$) is *false*, and for $\mathbf{DS} \neq \emptyset$, *fulfilled*($\mathbf{DS}$) is *true*. Of course, the following monotonicity property holds: if $\mathbf{X} \subseteq \mathbf{Y}$ and *fulfilled*($\mathbf{X}$) is *true*, then *fulfilled*($\mathbf{Y}$) is also *true*.

Definitions of *fulfilled* for other request models can be obtained as special cases of the above in the following manner:

- basic *k out of n*: $q=1$ and thus

$$\text{fulfilled}(\mathbf{A}) \equiv |\mathbf{DS} \cap \mathbf{A}| \geq k$$

- *OR-AND*: $(\forall j :: k_j = |\mathbf{DS}_j|)$ and thus

$$\text{fulfilled}(\mathbf{A}) \equiv (\exists j :: 1 \leq j \leq q :: \mathbf{DS}_j \subseteq \mathbf{A})$$

- *OR*: $(\forall j :: |\mathbf{DS}_j| = 1)$ and thus

$$\text{fulfilled}(\mathbf{A}) \equiv \mathbf{DS} \cap \mathbf{A} \neq \emptyset$$

where $\mathbf{DS} = \mathbf{DS}_1 \cup \mathbf{DS}_2 \cup \dots \cup \mathbf{DS}_q$.

- *AND*: $q = 1$ and $k = |\mathbf{DS}|$ and thus

$$\text{fulfilled}(\mathbf{A}) \equiv \mathbf{DS} \subseteq \mathbf{A}$$

Now we give precise definitions of a deadlock for the general request model. For sake of convenience, the quantification $(\forall P_i :: P_i \in \mathbf{B} :: \dots)$ means exactly $(\forall i :: (1 \leq i \leq n) \wedge (P_i \in \mathbf{B}) :: \dots)$. Thus, let *deadlock*($\mathbf{B}$) be a predicate signifying that the nonempty set $\mathbf{B}$ of processes is deadlocked. If *deadlock*($\mathbf{B}$) is *true* at some instant during the execution of an application program, it remains *true* as deadlock occurrence is persistent: *deadlock*($\mathbf{B}$) is a stable property. Generally, depending on the request model, the formal definition of this predicate can take several forms.

We formulate an abstract definition of deadlock, which holds irrespectively of the underlying request model. To do this, we introduce the following predicates [2]:

- *passive_i*: *true* iff P_i is passive.
- *arr_i(j)*: *true* iff a message from P_j has arrived and has not yet been consumed by P_i .
- *empty_i(j)*: *true* iff all messages sent by P_j to P_i have arrived.

Moreover, let $\mathbf{ARR}_i$ denote the set of all processes P_j such that *arr_i(j)* = *true*, and let $\mathbf{NE}_i$ denote the set of all processes P_j such that *empty_i(j)* = *false*. By *fulfilled_i* we denote the predicate *fulfilled* associated with P_i . An abstract definition for a set $\mathbf{B}$ of processes to be deadlocked is given below:

$$\begin{aligned} \text{deadlock}(\mathbf{B}) \equiv & \\ & (\mathbf{B} \subseteq \mathbf{P}) \wedge (\mathbf{B} \neq \emptyset) \wedge \\ & (\forall P_i :: P_i \in \mathbf{B} :: \\ & (\text{passive}_i \wedge \neg \text{fulfilled}_i(\mathbf{ARR}_i \cup \mathbf{NE}_i \cup (\mathbf{P} \setminus \mathbf{B})))) \end{aligned}$$

It means that any $P_i \in \mathbf{B}$ cannot be activated even after all messages from all processes in $\mathbf{ARR}_i \cup \mathbf{NE}_i$ and from all processes not deadlocked have arrived because these messages are not sufficient to satisfy the activation condition of any process in $\mathbf{B}$.

With the previous precise definition of generalized deadlocks, various formulations of the deadlock detection problem can be stated.

The problem of *detection of a deadlock occurrence* is to determine if there exist a set $\mathbf{B}$, such that $\text{deadlock}(\mathbf{B})$ is true. The result of a solution to this problem is a boolean dd , satisfying the following post-condition:

$$dd \equiv (\exists \mathbf{B} :: \text{deadlock}(\mathbf{B}))$$

The problem of *detection of a deadlock process* is to determine if a given process P is deadlocked, i.e., does there exist a set $\mathbf{B}$ such that $\text{deadlock}(\mathbf{B}) = \text{true}$ and $P \in \mathbf{B}$. The result of a solution to this problem is a boolean dd satisfying the following post-condition:

$$dd \equiv (\exists \mathbf{B} :: \text{deadlock}(\mathbf{B}) \wedge P \in \mathbf{B})$$

The problem of *detection of a deadlocked set* is to find a set of deadlocked processes. The result of a solution to this problem is a set of processes $\mathbf{PD}$ satisfying the following post-condition:

$$\text{deadlock}(\mathbf{PD}) \vee ((\mathbf{PD} = \emptyset) \wedge \neg(\exists \mathbf{B} :: \text{deadlock}(\mathbf{B})))$$

The problem of *detection of the maximum deadlocked set* is to find a deadlocked set which contains all deadlocked sets. This set is unique since the property of being deadlocked is closed under the set union operation. The result of a solution to this problem is a set of processes $\mathbf{PD}$ satisfying the following post-condition:

$$(\text{deadlock}(\mathbf{PD}) \vee (\mathbf{PD} = \emptyset)) \wedge \text{maxdead}(\mathbf{PD})$$

where

$$\text{maxdead}(\mathbf{PD}) \equiv (\forall \mathbf{B} :: \text{deadlock}(\mathbf{B}) \Rightarrow \mathbf{B} \subseteq \mathbf{PD})$$

Note that deadlock detection problems have been stated in order of increasing complexity. In addition, this order is consistent with an increasing amount of information which is available at the time the detection terminates. This additional information is important as it is useful for efficient recovery from the deadlock.

Solution of the deadlock detection problems is difficult in distributed systems because no process has accurate knowledge of the global system state, i.e., the global state is not visible to any process instantaneously. Thus, in practice, only states related to earlier observations can be obtained. However, during the collection of local states of processes, these states are changing. Therefore, any deadlock detection algorithm in a distributed system can only ensure that:

1. A deadlock which has occurred before the initiation of the algorithm will be detected.
2. The detected set of deadlocked processes was indeed deadlocked at the moment when the detection algorithm terminates.

We evaluate considered distributed deadlock detection algorithm according to their worst-case one-time complexity. The a worst-case one-time complexity (afterwards, time complexity for short) of distributed algorithm is the maximum time of a computation of the algorithm under the following assumptions:

- a process can execute any finite number of events in zero time,
- the time between the sending and receipt of a message is exactly one time unit.

Time complexity is expressed here as a function of *WFG* parameters, i.e. as a function of *WFG* diameter d and a length l of the longest path in *WFG*.

Distributed deadlock detection algorithm can be perceived as a set of control processes C_i , called *controllers*. Each controller C_i is associate with the application process P_i . The role of controller C_i is, on the one hand, to observe the behavior of P_i and, on the other hand, to cooperate with other controllers to consistently detect, if any, deadlock occurrence. In general, controllers need not be separated as special processes since their task can be incorporated into application processes using the superposition rules. Thus, the separation of controllers is merely a matter of interpretation and therefore controllers will not be explicitly specified further on.

3 On Time Complexity of BT Algorithm

3.1 Bracha and Toueg Algorithm ([1])

First we analyze Bracha and Toueg algorithm (BT algorithm) for a colorless *WFG* representing a static system (consistent snapshot) with instantaneous communication.

By $\mathbf{OUT}_v$ the dependent set of P_v is denoted. In other words, $\mathbf{OUT}_v$ is the set of nodes P_u from which P_v is expecting to receive messages. Let $\mathbf{IN}_v$ be the set of nodes that are waiting for message from P_v . It is assumed that the values of $\mathbf{IN}_v$, $\mathbf{OUT}_v$ are readily available at each node P_v .

A *WFG* $G=(V,E)$ models a static "snapshot" of the state of the system that contains no messages in the communication channels. Since message transmission is instantaneous, we have $\mathbf{OUT}_v = \{P_u \mid (P_v, P_u) \in E\}$ and $\mathbf{IN}_v = \{P_w \mid (P_w, P_v) \in E\}$. In this case, we have $P_v \in \mathbf{IN}_u$ if and only if $P_u \in \mathbf{OUT}_v$.

The algorithm BT consist of two phases: *notify* - in which processes are notified that a deadlock detection algorithm has started, and *grant* - in which active or potentially active processes simulate the message transmission. All the processes that are made "active" as result of this also simulate the message transmission. Deadlocked nodes are those never made "active" by *grant*. The *grant* phase is nested within the *notify* phase. This nesting ensures that the *notify* phase terminates only after the *grant* phase is over.

Formal description of Bracha and Toueg algorithm is presented in the following.

BT Algorithm

0. Initialization for every node P_v :
 - $\mathbf{OUT}_v := \{P_u \mid (P_v, P_u) \in E\};$
 - $\mathbf{IN}_v := \{P_u \mid (P_u, P_v) \in E\};$
 - notified, free*: = *false, false*;
 - $\mathbf{A} := \emptyset;$
1. **procedure** *Notify*(v);
 - notified*: = *true*;
 - for all** $P_w \in \mathbf{OUT}_v$ **send** (P_w , NOTIFY);
 - if** $\neg \text{passive}_v$ **then** *Grant*(v);
 - for all** $P_w \in \mathbf{OUT}_v$ **await** (P_w , DONE);
2. Upon receipt by P_v of NOTIFY from P_u :
 - if** $\neg \text{notified}$ **then** *Notify*(v);
 - send**(P_u , DONE);
3. **procedure** *Grant*(v);
 - free* := *true*;
 - for all** $P_w \in \mathbf{IN}_v$ **send**(P_w , GRANT);

for all $P_w \in \text{IN}_v$ **await**(P_w, ACK);

4. Upon receipt by P_v of *GRANT* from P_u :
 $\mathbf{A}_v := \mathbf{A}_v \cup \{P_u\}$;
if $\neg \text{free}$ **and** $\text{fulfilled}_v(\mathbf{A})$ **then** *Grant*(v);
send (P_u, ACK);

3.2 Examples of BT Algorithm Application

Let us analyze first an application of the BT algorithm for generalized deadlock detection in the system state represented by the *WFG* depicted in Fig.1a.

All processes P_a, P_b, P_c , and P_d composing the set $\mathbf{P}$ are passive and they are waiting for some messages from other processes of the set according to *WFG* edges. For a sake of presentation simplicity, we associate with each node (process) of *WFG* two vectors representing state of the algorithm with respect to a node. The first vector, denoted by $\mathbf{PS}$ (processing state) consists of two elements PS_1 and PS_2 . Thus, $\mathbf{PS} = [PS_1, PS_2]$. The number PS_1 , depends on a value of boolean variable *notified*, and on receipt of *DONE* message. Initially, PS_1 is equal to zero. After receipt of the first *NOTIFY* message by the analyzed detection process its PS_1 gets value one. Then, when the *DONE* message is received, PS_1 is changed to two. Similarly, PS_2 depends on a value of boolean variable *free*, and on receipt *ACK* message. Initially, PS_2 is equal to zero. After receipt of the first *GRANT* message by the analyzed detection process, PS_2 gets value one. Then, when the of *ACK* message is received, PS_1 is changed to two. The second vector associated with each node is denoted by $\mathbf{CS}$ (communication state). It consists of three elements *CS_GRANT*, *CS_ACK* and *CS_DONE* representing counters of *GRANT*, *ACK* and *DONE* messages received, respectively. Initially all these counters are equal to zero. They are accordingly incremented as a result of a receipt of *GRANT*, *ACK* or *DONE* message. In the Figure, the vector $\mathbf{PS}$ is placed inside a node, but the vector $\mathbf{CS}$ just by a node.

Let in our example P_a be the unique initiator of the deadlock detection. It calls *NOTIFY* procedure and thus, sequentially it changes $\mathbf{PS}$ to $[1,0]$, sends *NOTIFY* messages to all processes composing its dependent set OUT_a , and waits for *DONE* messages confirming all *NOTIFY* messages, as is depicted in Fig. 1b. Then concurrently, processes P_b, P_c and P_d receive *NOTIFY* messages and as a consequence they execute *NOTIFY* procedure changing appropriately PS_1 and relaying *NOTIFY* messages to the processes of their dependent sets (see Fig. 1c). In the next step, *NOTIFY* messages arrive to processes whose notified flags are *true*. Therefore, the processes simply send in response *DONE* message (Fig. 1d). As a result, processing state vectors of P_b, P_c and P_d are set to $[2,0]$. It enables then to send *DONE* messages to the initiator P_a . When the initiator receives all *DONE* messages, the algorithm terminates claiming in the considered case deadlock occurrence.

The second example illustrating behavior of BT algorithm is presented in Fig. 2. The *WFG* is very similar to the previous one but the process P_d has no outgoing edges. It means that P_d is active as opposed to all other processes. We assume here *AND* request model for all processes. Let again P_a be the initiator of deadlock detection. It initiates notify phase sending *NOTIFY* messages to all its descendants as shows Fig. 2b. Thus, P_a is *notifyinitiator* for all other processes. When *NOTIFY* message is arrived to active process P_d with empty dependent set, P_d calls *Notify* procedure. This procedure changes PS_1 of P_d to 2 (which means that P_d need not wait for any *DONE* messages) and calls *Grant* procedure. As a result, PS_2 of P_d is set to 1 and *GRANT* messages are sent to all processes composing IN_d set of P_d . In this step, processes P_b and P_c change their PS_1 to 1 and forward *NOTIFY* messages to all processes of their dependent sets (see Fig. 2c). Thus, P_d is *grant initiator* for P_a, P_b and P_c . It means also, that P_d will send *DONE* message to its notify initiator only after receiving *ACK* responses for its all *GRANT* messages. In the next step (Fig. 3d) P_c , among others, receives *GRANT* from P_d and forwards it to all its ancestors. It is important to note here, that P_b does not forward *GRANT* to P_a as it can be done in *AND* request model only when *GRANT* messages have been arrived from all descendants. This condition is fulfilled in the next step (Fig. 2e) when P_b receives *GRANT* message from P_c , as P_c is the *grant terminator* of P_b . In this case P_b sends *GRANT* to P_a (Fig. 2e). It implies that in the next steps *ACK* message goes subsequently to nodes P_b, P_c and P_d (see Fig. 2f, Fig. 2g and Fig. 2h, respectively).

Figure 1: Example of BT Algorithm Application for Deadlock Detection

Figure 2: Another Example of BT Algorithm Application for Deadlock Detection

When message ACK arrives at P_d , it terminates grant phase of P_d enabling him to send DONE message to its notify initiator.

3.3 Complexity Analysis of BT Algorithm

It is argued in [1], and then cited in literature (e.g. [13], [16]) that time complexity of the BT algorithm is of $4d$.

However, in the above analyzed example (Fig.2) the given *WFG* is characterized by $d = 1$ but the number of steps required to terminate the algorithm is 8. To find a cause of this ambiguity let us note that grant phase is propagated (GRANT message is sent to ancestors) by passive processes only when arrived finally the GRANT message setting predicate *fulfilled* to *true* for the first time. In general it requires that GRANT messages from all descendants have to arrive (e.g. for *AND* request model). But the time required to meet this point depends on the longest path from grant initiator to a given process. Indeed, let us consider a graph in which an initiator P_x of a grant phase is connected to its ancestor through two disjoint backward paths of length l_1 and l_2 , respectively, where $l_1 > l_2$. Thus, in the best case when arrived GRANT message is propagated at each node immediately, i.e. this message is enough for predicate *fulfilled* to be *true*, the first GRANT message arrives to P_y in l_2 time units. Now we assume, that process P_y needs two GRANT messages to evaluate predicate *fulfilled* to *true*. The second GRANT message requires l_1 time units and hence both messages are ready to be consumed in time equal to $\max(l_1, l_2)$. But in general, l_1 can be equal to l , where l is the longest path in the *WFG*. Thus, even if there is no delays in intermediate nodes, passing of GRANT message can take up to l time units. Then, ACK message is sent back to the grant initiator and its transmission takes also up to l time units.

On the other hand let us note that ACK message is sent immediately after receiving GRANT message when the node has been already granted or predicate *fulfilled* is not *true*. Hence, in general, time complexity of grant phase is of $2l$. However, as the grant phase is nested in notify phase, to find time complexity of the whole BT algorithm we have to add to grant phase complexity the time required to notify the grant initiator as well as the time required to send the DONE message from the grant initiator to the deadlock detection initiator. But NOTIFY messages are propagated immediately to all processes composing dependent set. So they need at most d hops (time units) to reach any node from a deadlock detection initiator. The same, one can conclude as about the time required to propagate DONE message when grant phase is terminated.

Thus, summing up the above time complexity analysis we can formulate the following theorem.

Theorem 1.

The time complexity of BT algorithm is of $2d + 2l$.

The above theorem is confirmed by the example presented in Fig. 2. Indeed, as the considered *WFG* is characterized by $d = 1$ and $l = 3$, the number of hops required to terminate algorithm should be 8 in accordance with the theorem. The same number of hops has been indeed enough in our example.

In the discussed till now BT algorithm it has been assumed that each node knows both dependent set and its *IN* set. The latter assumption seems be not very realistic in all possible cases, but the knowledge of *IN* set is necessary to propagate GRANT messages and to correctly acknowledge incoming GRANT message. For instance, in our example in Fig. 2 process P_a should not reply with ACK message until it is sure that its *IN* set is empty. It means that some extra processing is required to construct *WFG* and to define the *IN* sets of all nodes. It takes only d hops to notify all nodes and one extra hop to flush outgoing edges, but termination of this processing must be really perceived by the nodes before sending back some ACK. Despite this processing can be overlapped partially with deadlock detection process it may require extra time. For example in Fig. 2 processes really perceive incoming edges of *WFG* only after receiving NOTIFY messages. Thus, GRANT messages cannot be sent out until appropriate NOTIFY messages have been received. In the considered case it means that GRANT messages from P_d to P_b and P_c will be postponed one time unit.

4 On Time Complexity of KS Algorithm

4.1 Kshemkalyani and Singhal Algorithm ([12])

Let us analyze now a consistent snapshot without messages in transit (a static system with empty channels). The algorithm proposed by Kshemkalyani and Singhal, called here KS algorithm, consists of two concurrent sweeps of messages. In the *outward sweep*, the algorithm records a consistent snapshot of a distributed wait-for-graph. In the *inward sweep*, it performs reduction of the recorded distributed *WFG* to check for a deadlock. These two sweeps can overlap in time at a process.

Distributed *WFG* is recorded using FLOOD messages during the outward sweep and is examined for deadlocks using ECHO messages during in the inward sweep. When blocked, the initiator P_i records its local state, the time $t.block_i$ at which P_i has been last blocked, and sends FLOOD messages along its outgoing wait-for edges at the time it blocks.

At the time a node P_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 reachability set of the initiator participate in recording of the *WFG* in the outward sweep. 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.

ECHO messages perform reduction of the nodes and edges in the *WFG* by simulating the message transmission in the inward sweep. Assuming k out of n request model, a node gets reduced at the time it has received $k = p$ 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.

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. 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. Thus, the two activities of recording the *WFG* snapshot and reducing the nodes and edges in the *WFG* snapshot are done currently in a single phase, and no serialization is imposed between the two activities.

A termination detection technique based on weight [10, 17] detects the termination of the algorithm by using SHORT messages (in addition to FLOOD's and ECHO's). A weight of 1 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.

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 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. 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. When an ECHO arrives at a reduced node, its weight is sent directly to the initiator through a SHORT message. 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.

A node P_i stores the local snapshot to detect deadlocks in a data structure LS_i , which is an array of records. Record $LS_i[P_{init}]$ stores a snapshot at node P_i corresponding to deadlock detection initiation by the initiator node P_{init} .

In addition, the initiator has a variable wt_{init} in which it collects returned weights.

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

The formulation presented below is for k out of n request model. However, it can be easily transformed to general request model. To do this it is sufficient to evaluate predicate *fulfilled* when a node receives an ECHO message and reduce the node when accordingly to these messages *fulfilled* changes to *true*, instead of when $LS_i.p$ becomes 0.

KS Algorithm

1. Initialization for every node P_i :

```

 $LS_i$  : array [1..N] of record
  out: set of integer :=  $\emptyset$  ;
/*Set of nodes for which node  $P_i$  is waiting in the
snapshot*/
  in: set of integer :=  $\emptyset$ ;
/*Set of nodes waiting for node  $P_i$  in the snapshot*/
  t: integer := 0;
/*time when  $P_{init}$  initiated snapshot.*/
  p : integer := 0.
/*the number of messages required for unblocking.*/

```

2 Initiate a deadlock detection:

```

/*Executed by process  $P_i$  to detect whether it
is deadlocked.*/
 $init := i$ ;
 $wt_{init} := 0$ ; /*initial weight.*/
 $LS_i.t := t_i$ ;
 $LS_i.out := out_i$ ;
 $LS_i.in := \emptyset$ ;
 $LS_i.p := p_i$ ;
send FLOOD( $P_i, P_i, t_i, 1 / |out_i|$ ) to each  $P_j$  in  $out_i$ .

```

3. Upon receipt by P_i of FLOOD ($P_k, P_{init}, t_{init}, w$) :

```

/*Executed by process  $P_i$  on receiving a FLOOD
message from  $P_k$ .*/
[
/*Valid FLOOD for a new snapshot.*/
if  $LS_i.t < t_{init} \wedge P_k \in in_i$  := then
   $LS_i.out := out_i$ ;
   $LS_i.in := \{P_k\}$ ;
   $LS_i.t := t_{init}$ ;
   $LS_i.p := p_i$ ;
  if  $p_i > 0$  then
/* $P_i$  is blocked.*/
    send FLOOD( $P_i, P_{init}, t_{init}, w / |out_i|$ )
    to each  $P_j \in out_i$ ;
  if  $p_i = 0$  then
/* $P_i$  is unblocked.*/
    send ECHO ( $P_i, P_{init}, t_{init}, w$ ) to  $P_k$ .

```

□

4. /* FLOOD for a new snapshot is received along an edge that has ceased to exist. In dynamic system a reply has already been sent back to the sender of the FLOOD. */
if $LS_i.t < t_{init} \wedge P_k \notin in_i$ **then**
 send ECHO($P_i, P_{init}, t_{init}, w$) to P_k .

□

5. /* FLOOD for a current snapshot is received along an edge that has ceased to exist. In dynamic system a reply has already been sent back to the sender of the FLOOD. */
if $LS_i.t = t_{init} \wedge P_k \notin in_i$ **then**
 send ECHO ($P_i, P_{init}, t_{init}, w$) to P_k .

□

6. /* Valid FLOOD for current snapshot. */
if $LS_i.t = t_{init} \wedge P_k \in in_i$ **then**
 $LS_i.in := LS_i.in \cup \{P_k\}$;
 if $LS_i.p = 0$ **then**
 /* P_i is unblocked in snapshot. */
 send ECHO ($P_i, P_{init}, t_{init}, w$) to P_k ;
 if $LS_i.p > 0$ **then**
 /* P_i is blocked in snapshot. */
 send SHORT(P_{init}, t_{init}, w) to P_{init} .

□

7. /* Outdated FLOOD. */
if $LS_i.t > t_{init}$ **then** discard the FLOOD message.

]

8. Upon receipt by P_i of ECHO ($P_j, P_{init}, t_{init}, w$):
/* Executed by process P_i on receiving an ECHO from P_j for a current snapshot for which $LS_i.t = t_{init}$. ECHO for an outdated snapshot ($LS_i.t > t_{init}$) is discarded. */
if $LS_i.t = t_{init}$ **then**
 $LS_i.out := LS_i.out - \{P_j\}$;
 if $LS_i.p = 0$ **then**
 /* P_i is unblocked in snapshot. */
 send SHORT (P_{init}, t_{init}, w) to P_{init} .
 if $LS_i.p > 0$ **then**
 /* P_i is blocked in snapshot. */
 $LS_i.p := LS_i.p - 1$;
 if $LS_i.p = 0$ **then**
 /* P_i becomes unblocked in snapshot. */
 if $init = i$ **then** declare not deadlocked; exit.
 send ECHO($P_i, P_{init}, t_{init}, w$ | $LS_i.in$ |)
 to all $P_k \in LS_i.in$;
 if $LS_i.p \neq 0$ **then**
 /* P_i remains blocked in snapshot. */
 send SHORT (P_{init}, t_{init}, w) to P_{init} .

9. Upon receipt by P_i of SHORT (P_{init}, t_{init}, w) :
- /* Executed by process P_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.*/
 - if** $t_{init} = t_{block_i} \wedge LS_i.p = 0$ **then**
 discard the message.
 - if** $t_{init} = t_{block_i} \wedge LS_i.p > 0$ **then**
 $wt_{init} := wt_{init} + w$;
 if $wt_{init} = 1$ **then** declare deadlock and abort.

4.2 Example of KS Algorithm Application

Let us analyze an application of KS algorithm for generalized deadlock detection. The assumed *WFG* is the same as in the previous example presented in Fig.2. and we assume again *AND* request model for all processes. The considered now example is depicted in Fig.3.

Let again P_a be an initiator of deadlock detection, and P_d the only active process. First, the initiator P_a sends out to processes composing its dependent set FLOOD messages with weight 1/3 each (see Fig. 3.a). As P_d is active it can send ECHO message back immediately with weight equal to the weight received in the FLOOD (Fig. 3b). Process P_c simply forwards FLOOD message with weight 1/3 to its descendent, but P_b forwards FLOOD messages to P_c and P_d sharing incoming weight by two, i.e., associating with each message weight 1/6. In the next step (Fig. 3c), P_a as active simply responses sending ECHO messages to P_c and P_b with weights 1/3 and 1/6, respectively. P_c sends SHORT message to the initiator in response to the subsequent FLOOD message. Then, after receiving ECHO with weight 1/3, P_c forwards this message to its ancestors P_a and P_b sharing equally the received weight. P_b , however, in response to ECHO which is not enough for its activation, sends back to P_a SHORT message with weight 1/6. In the last step, P_b forwards the received ECHO message to P_a . The algorithm terminates when the deadlock detection initiator P_a receives back messages with total weight equal to 1.

4.3 Complexity Analysis of KS Algorithm

When comparing BT algorithm with KS one, it is easy to note some similarities. FLOOD message corresponds in some sense to NOTIFY, ECHO corresponds to GRANT and DONE, but SHORT substitutes partially both DONE and ACK messages. The key difference result from application of weight based termination algorithm. As a consequence, KS algorithm is more time efficient and does not assume the *a priori* knowledge of *IN* sets.

It has been argued that the KS algorithm has the time complexity of $2d$. However the above analyzed example shows that we need 5 steps to terminate the algorithm instead of $2 * 1 = 2$. Again to find a cause of the ambiguity let us note that to send ECHO messages through all incoming edges FLOOD messages must first flush all edges. It needs, as is easy to verify, $d + 1$ steps. Then, following the argumentation similar to that for GRANT messages in BT algorithm, let us note that to set predicate *fulfilled* to *true* at P_b , both ECHO messages from P_c and P_d have to be received. In general the time required to reach this state depends on the longer path linking active process P_d with P_b . Thus, the longest path in *WFG* determines the time required to propagate ECHO messages from an active process to the deadlock detection initiator. The above observation leads directly to the following theorem. **Theorem 2**

The time complexity of KS algorithm is of $(d + 1) + l$.

This result is consistent with the example presented in Fig. 3.

Figure 3: Example of KS Algorithm Application for Distributed Deadlock Detection

5 Conclusions

In this paper the problem of worst-case one-time complexity analysis of two well known distributed algorithms for generalized deadlock detection has been addressed. The time complexity has been expressed as a function of the diameter d and the longest path l of the wait-for-graph (*WFG*) characterizing a state of static distributed system with instantaneous communication. First, the algorithm proposed by Bracha and Toueg has been considered. It has been shown that its time complexity is of $2d + 2l$. Then, we have proved that the time complexity of Kshemkalyani and Singhal algorithm is of $(d + 1) + l$. These results improve time complexity estimates proposed till now.

In the above context it seems to be natural to state the problem of finding lower-bound for the worst-case one-time complexity of generalized deadlock detection in static as well as dynamic systems. From the discussion presented in this paper we can only conclude that the lower-bound of the time complexity for distributed reduction of a given *WFG* is $O(l)$. Thus further investigations in this line are required.

Moreover, additional analysis is necessary to precisely estimate how far a phase of consistent *WFG* construction can be overlapped with *WFG* reduction, especially in dynamic systems.

Another noteworthy point is the difference between the system model considered in this paper and the one comprising REQUEST, REPLY and CANCEL messages for resource allocation. The latter model seems to be less general but it gives extra opportunity to exploit these application messages in some preliminary phases of deadlock detection as it has been done in [1] and [12].

References

- [1] Bracha G, Toueg S. Distributed deadlock detection. Distributed Computing 1987; vol. 2, 3:127-138.
- [2] Brzezinski J, Helary JM, Raynal M, Singhal M. Deadlock models and a general algorithm for distributed deadlock detection. Journal of Parallel and Distributed Computing 1995; vol. 31, 2:112-125.
- [3] Brzezinski J, Helary JM, Raynal M. Deadlocks in distributed systems - request models and definitions. Proc. of the Fifth IEEE CS Workshop on Future Trends of Distributed Computing Systems, Cheju, Korea, August 1995, pp 186-193.
- [4] Chandy KM, Lamport L. Distributed snapshots: determining global state of distributed systems. ACM Trans. on Comp. Systems 1985; vol. 3, 1:63-75.
- [5] Chandy KM, Misra J. A distributed algorithm for detecting resource deadlock in distributed systems. Proc. of ACM Symposium on Principles of Distributed Computing, ACM, New York, 1982, pp 157-164.
- [6] Chandy KM, Misra J, Haas LM. Distributed deadlock detection. ACM Trans. on Comp. Systems 1983; vol. 1, 2:144-156.
- [7] Gifford DG. Weighted voting for replicated data. Proc. of the 7th ACM symposium on Operating Systems Principles. ACM, New York, 1979, pp 150-163.
- [8] Hermann T, Chandy KM. A distributed procedure to detect AND/OR deadlocks. Tech. Report LCS-8301, Dept. of Computer Science, University of Texas at Austin, 1983.
- [9] Holt RC. Some deadlock properties of computer systems. ACM Computing Surveys 1972; vol. 4, 3:179-196.
- [10] Huang S. Detecting termination of distributed computations by external agents. Proc. 9th Int. Conf. Distributed Comput. Syst., 1989, pp 79-84.
- [11] Knapp E. Deadlock detection in distributed databases. ACM Computing Surveys 1987; vol. 19, 4:303-328.

- [12] Kshemkalyani AD, Singhal M. Efficient detection and resolution of generalized distributed deadlocks. IEEE Trans. on Soft. Eng. 1994; vol. 20, 1:43-53.
- [13] Kshemkalyani AD, Singhal M. On the characterization and correctness of distributed deadlock detection. Journal of Parallel and Distributed Computing 1994; vol. 22:44-59.
- [14] Mitchell D, Merritt MJ. A distributed algorithm for deadlock detection and resolution. Proc. of the 3rd ACM Symposium on Principles of Distributed Computing, ACM, New York, 1984, pp 282-284.
- [15] Natarajan N. A distributed scheme for detecting communication deadlocks. IEEE Trans. Soft. Eng. 1986; vol. SE-12, 4:531-537.
- [16] Singhal M. Deadlock detection in distributed systems. IEEE Computer 1989, vol. 22, 11:37-48.
- [17] Wang J, Huang S, Chen N. A distributed algorithm for detecting generalized deadlocks. Tech. Rep. Dept. of Comput. Sci. National Tsing-Hua University, 1990.