ISSN 2229-5518

A Revised Algorithms for Deadlock Detection and Resolution in Mobile Agent Systems.

Rashmi Priya (TMU Research Scholar,India)

Abstract

A study on Deadlock detection is being done for many years. Not much work has been done on Deadlock resolution.
Wait –for model approach followed to avoid deadlocks offers incorrectness to many algorithms after deadlocks have been resolved.
In this paper, a theoretical framework for wait-for systems is provided, and general characteristics of a correct algorithm for deadlock detection and resolution are presented.
It is shown that the computational upper bounds(number of messages) for deadlock detection and resolution are both O(n3) in the worst case when n transactions are involved. This result is better than previous ones, which often are even exponential. In addition, two correct deadlock detection and resolution algorithms are described which both achieve these upper bounds.

Keywords : Mobile Agents, distributed computing, wait –for model, complexity.

1. Introduction

In recent years, a large number of algorithms for deadlock detection in distributed computing systems has been proposed. In the area of distributed databases, to which the attention is restricted in this paper, deadlock situations arise when locks on data objects are used in scheduling concurrent transactions. As a consequence, deadlocks need to be detected and then resolved . As has been pointed out by these and other authors , three problems arise in many of these proposals: First, some of them are incorrect in that they may either detect false deadlocks, or fail to detect real ones.
Second, deadlock resolution is sometimes neglected or not handled properly ; the latter is often the reason for errors occurring in the detection of (subsequent) deadlocks, since the wait-for relationship maintained by the algorithms is not updated correctly. Third, not enough attention is paid to the computational complexity of deadlock detection and resolution. For example, proposes an algorithm requiring the transmission of exponentially many messages for deadlock detection in the worst case, which is unacceptable in practical applications.

These three problems are addressed in this paper. We begin by providing a theoretical framework for wait-for systems that are employed to model a distributed environment, as it pertains to scheduling concurrent transactions. We then show that the upper bound for the number of messages to be transmitted during deadlock detection and resolution (and hence the overall time
complexity) is O (n3) when n transactions are involved. Next, we investigate how these results obtained for a static
situation carry over to the dynamic case as well. To this end, we give a characterization of when a deadlock detection and a resolution algorithm is "correct", respectively. Finally, two algorithms are presented, which differ in the use of priorities assigned to nodes of a wait-for graph, and it is shown that both are correct and achieve the upper bounds derived earlier. It should be pointed out that other recent papers also address the issues discussed here. In particular, [9] gives a thorough evaluation of various algorithms for deadlock detection (neglecting resolution), and discusses many issues related to them. Also, the work of Roesler is notable in this context for its detailed discussion of all three problems mentioned above, and for the provision of polynomial and provably correct solutions. The model used in these papers is an object-oriented one based on abstract data types, so that the approach is only partially comparable .

The organization of this paper is as follows: In Section 2, we introduce a graphical model for wait-for systems. This model will allow us to derive precise upper bounds for the complexity of deadlock detection and resolution algorithms; to this end, we will show (1) that the upper bound on time is O(n3) for both detection and resolution, and (2) that the upper bound on space is O(n3),w here n is the number of transactions/processes involved. In Section 3, we consider the dynamic case, in which the complete wait-for graph is changing over time. Criteria for correct deadlock detection in this case are provided. We present two new algorithms for deadlock detection and resolution in Section
4, which actually achieve the upper bounds. In Section 5 we survey previous work in some more detail and in particular exhibit a classification of relevant algorithms which captures their essential
features. Finally, we sketch several questions that deserve further study in Section 6.

2. Wait-For Systems: The Model and Its Static View

In this section, we present our model of a (distributed) system which, due to the use of locks in the scheduling of transactions, is subject to deadlocks and hence calls for their detection and resolution; we assume that deadlocks are indeed possible in the

system, and we are therefore not interested in algorithms for deadlock prevention.

Informally, in such a system there exist processes or transactions which operate on shared resources; any available resource can be used by several transactions at a time in shared mode, or by at most one at a time in exclusive mode. If a resource is in (exclusive) use, any other transaction trying to access it has to wait for the first to release the resource. A deadlock occurs when two or more transactions are waiting for each other in a cyclic manner.

In a distributed computing system, typically no single site has full knowledge of the entire system; thus, we assume, without loss of generality, that upon the occurrence of a resource conflict, each transaction or process must react accordingly (by sending messages) and independently (without reporting to a central site).

Formally, such a wait-for system can be modeled using a directed graph, the wait-for graph (WFG), defined next.

Definition 1 (Wait-For Graph) A wait-for graph (WFG) G = (V,E) is a directed graph, where the set V of nodes represents processes or transactions, and the set E of edges represents resource dependencies between nodes s.t. an edge e from v1 to v2 indicates that v1 is waiting for v2 to release a resource. Nodes having out-degree zero are called active; all others are called blocked.

This model will serve as a standard metric to analyze the complexity of deadlock detection/resolution algorithms. We also assume that any WFG G under consideration has no self-loops on nodes. A sample WFG is shown in Figure 1. In order to describe a (possibly transitive) dependency between two distinct nodes, it is convenient to talk about one path through a WFG at a time:

Definition 2 (Wait-For String) Let G = (V,E)b e a WFG. A wait-for string (WFS) S is a path through G (without loops). Every node v E V appearing in S has at most one incoming and one outgoing edge, and there exist at most two nodes r, h E V s.t. r has no incoming edges, whereas h has no outgoing edges. r is called the root or tail of S, and h is called its head.

For example, in Figure 1 (v1,v3,v4) is a wait-for string. The next definition introduces useful shorthands for a further discussion of wait-for strings:

Definitions 3 (Down-Stream, Up-Stream) Let G = (V,E) be a WFG, and let v E V. The down-stream of v, denoted DS(v), consists of all nodes in V for which v is either directly or indirectly waiting, i.e., the nodes v' E V s.t. v is followed by v' in a WFS S. Similarly, the up-stream of v, denoted US(v), consists of all nodes in V which are waiting for v directly or indirectly, i.e., all nodes preceding v in S. The edges of v'S down-stream [up-stream] are called the down-stream [up-stream] edges of v,resp.

A down-stream DS(u) is called essential to U if no node U' of the upstream of U can reach a node in the down-stream without passing through U, i.e., U' cannot bypass U on another downstream.

An essential down-stream node & of node U is a downstream node of U s.t. for every path formed between a node d, d E US(u) U{v}, and k, this path must include U. The outgoing edges of k are called the essential down-stream edges of U. Note that an articulation point [I] is always an essential downstream node of its upstream nodes, but the reverse is not true in general.

In the example above, {v3,v2,v4} = DS(v1), {v1,v2,v3} = US (v4) , and {v2 ,v4 }=D S(v3) is essential to v3. We now consider a static situation in which some WFG G is given, and there exists an "oracle" that is able to tell us about cycles. Our first result, which is interesting in its own right, shows that in an arbitrary graph G = (V,E), there are exponentially many cycles in the worst case:

Lemma 1 Let G = (V,E)be a WFG, I V I= n. Then there are up to O(n!) cycles in G.

Proof. For n nodes, the possible cycles are formed by 2 nodes, 3 nodes, . . ., and n nodes, and for each cycle involving i nodes, there are (i - I)! possible permutations. Since each permutation yields a different cycle, the total number of possible cycles results from a summation of C(n,2), C(n,3), . . ., C(n,n), where C is the combinatorial selection function, i.e., the total number of cycles equals n! = z-E O(n!) our claim follows.

Since it is thus impossible in practice to keep track of all cycles in a WFG, we next explore a possibility to detect all cycles in a given WFG without explicitly tracking them: Given a WFG G, let cyc(G) denote the set of all cycles in G, each of which represents a deadlock situation. If G has nodes 1, . . . , n, we can distribute the members of cyc(G) into n groups (to which we refer as cycle-break pupa throughout the paper) GI,. . . ,Gn s.t. a cycle C E cyc(G) belongs to G; if i E C. Now we have: Lemma 2 If G;, , . . . , G;*-, is a selection of n - 1 distinct cycle break groups, then

n- 1 U Gij = cyc(G) j=1

Proof. Assume, on the contrary, that the union of all groups in the given selection does not equal cyc(G). Then there must be a cycle C E cyc(G) which is not in Gij for each j E (1,. . . , n - 1). Thus, cycle C does not go through nodes i1 , .. . ,i n-1 which implies that C goes through the one node in only, a contradiction to the exclusion of self-loops on nodes. We next consider the (time and space) complexity of cycle detection and resolution. Since we are considering a static case, let a WFG G be given, and let G contain cycles. (Notice that depth-fist search as described in [1] cannot be applied due the fact that we consider a distributed environment.) In order to detect the cycles, we color the edges of G, using the oracle, in such a way that a unique color corresponds to each individual cycle-break group. Thus, an edge will be assigned as many colors as there are cycles in which it appears. Then, in order to break a cycle, colors are removed

from edges as appropriate, again by referring to the oracle. We assume that each addition or removal of a color to or from a single edge can be performed in one unit of time, and that each color needs one unit of space. Now we have: Theorem 1 (Upper bound for detection) To detect all cycles in a given WFG with n nodes, the maximum number of operations needed is O(n3).

Proof. By Lemma 2, if we can detect n - 1 of the n possible cycle-break groups, then we can detect all cycles in G. If each such group is identified by a different color, then each edge in a WFG will be colored at most n - 1 times. Since there are at most (n - 1)n edges in G, we only need (n - 1)(n - 1)n coloring operations to detect all cycles, which is O(n3).

Notice that in the above theorem, we do not distinguish between deadlock cycles which belong to the same cycle-break group.

Corollary 1 (Upper bound for resolution) To break all cycles, at most O(n3) operations are needed.

Proof. As in Theorem 1, each edge can get at most n - 1 colors; thus, to remove all colors, an edge has to be accessed up to n – 1 times. Since there are <= (n - 1)n edges, this requires <= (n -1) (n - 1)n

E O(n3) steps.

By similar arguments, we can easily obtain an upper bound of O(n3) for the space requirement of a given WFG with n nodes.

The next question that we consider is if and how these upper bounds can actually be achieved, since an "oracle" will not be available in reality. After this, we apply the techniques obtained to the (more realistic) dynamic case in the next section.

We exhibit a (static) method for keeping track of wait-for dependencies next. As a consequence of the above results, we adopt the cycle-break group idea as follows:

We assign each node a unique color. In order to detect cycles, we color the edges in the WFG as before by sending the color of a node along all its outgoing edges in a coloring probe; whenever a color is received by some node, it forwards it further. To break cycles, edge-colors are removed by sending around cleaning probes. Each addition or removal of a color to or from a single edge is counted as one step. Every coloring or cleaning probe needs one unit of space (for storing the representation of a color). When a coloring probe visits an edge, the edge stores the color carried by the probe, and the edge is said to be colored with that color. When a cleaning probe visits an edge, the color carried by the probe is removed if the edge was colored with it.

(We assume that both the edges and the nodes in the WFG can pile as many colors as required.)

A cycle with color c is detected if a probe tagged with c is returned to the node colored by c. Similarly, a cycle with color c is resolved, if c is erased from every edge of the cycle colored with c. A colored cycle C cannot simply be broken by erasing the color of C from all corresponding edges, since this color is not confined within that cycle only. Following the routes of a coloring process, the down-stream edges of a node U are colored with the same color as v, which is indicated by "v-c*-> w", where the asterisk indicates a (direct or indirect) wait-for relationship, and w E DS(v). (To simplify our discussion, we will omit the symbol c whenever confusion can be excluded.) The above indicates that we need to erase c from all edges corresponding to the downstream of v, if v is chosen as the victim. In order to erase the exact amount of the colors, we must proceed carefully. Consider the example in Figure 2, which is identical to that in Figure 1 except we associate colors with the wait-for edges. We also list some of the wait-for dependencies which are relevant to our discussion; note that not all nodes on a cycle are always capable of detecting a deadlock.

Here we assume that colors a, b,c, and d are assigned to nodes v1,v2,v3, and v4, respectively. If v3 is chosen as the victim, all colors should be erased from the edges (v3,v2) and (v2,v3). In addition, colors a, b and c need to be removed from edge (v3,v4). On the other hand, if v2 is chosen as the victim, b should be removed from edge (v3,v4), but not a and c. Thus, the relation v1 of v4 still exists, because the deletion of node v2 should not affect the wait-for condition between v1 and v4.It follows that before we can determine the cost of a deadlock detection and resolution algorithm, we need to know what makes a dynamic algorithm correct. The next section addresses this question.

3 The Dynamic Case and its Correctness Criteria

In a dynamic situation, a given WFG changes over time. In addition, the problem of message-propagation delay in the distributed system arises. As a consequence, the model presented above has to be slightly modified in order to reflect the dynamic situation; in particular, we make the following assumptions: (1) The underlying computer network is fault-free. (2) No messages are received in error. (3) All messages arrive at their respective destination in finite time. Properties 1 and 2 free us from the problems which are related to faults. After a node sends out a resource request, but before this request has been acknowledged, the node is in an idle state. During the idle period, the node will not process any new probes received. Because of Property 3, the period of the idle state is finite, and there is no possibility for a node to wait forever.

First we introduce a notion which captures the correct wait for relationship between victims and heir down-streams:

Deflnition 4 (Down-Stream Informality) Let A be an algorithm for cycle resolution. Suppose that A is applied to a cycle C, and that during the resolution of the deadlock represented by C node U is chosen as the victim to break C. If the set R of inward colors of U is removed from every edge in the essential down-stream of U, then we say that A has down-stream infomality.We turn to the issue of correctness next. Clearly, a deadlock detection algorithm is correct if it detects all real cycles and no false ones; a deadlock resolution algorithm is correct simply if it is able to resolve a deadlock. However, since distributed systems face the problem of propagation delay of messages, a cycle detection and resolution algorithm may detect false cycles due to this delay.

Clearly, this factor is difficult to eliminate and will therefore not be considered further here.

In [21] it is claimed that a DBMS employing 2PL will not introduce false deadlocks. However, this is not true in general, since after a deadlock has been detected and resolved, false deadlocks can still be detected under several circumstances; to see this, the algorithm described in [17] (which employs 2PL) can be taken as an example. [20] and [4] have shown that [17] either detects false deadlocks or ignores real deadlocks under several circumstances.

Thus, the use of 2PL is neither sufficient nor necessary for correct deadlock detection.

Another problem that should be taken into consideration can be seen from the following example. Consider Figure 2 once more: After u3 has been chosen as the victim and before the corresponding

message can be received by node u4, a new edge introduced between nodes u2 and u4 will let the algorithm conclude that a new cycle is formed among u2, u3 and u4. For this reason, we assume there is no time delay when considering whether an algorithm is correct. Now the following is obvious:

Lemma 3 A cycle detection algorithm is correct if it can calculate either all WFSs or all cycle-break groups.

Proof. The first condition is obvious. For the second, it follows from Lemma 2 that every cycle belongs to certain cycle-break groups. Thus, detecting all cycle-break groups actually detects all deadlocks.

Lemma 4 A cycle resolution algorithm is correct if and only if it has down-stream informality.

Proof. When a cycle is broken by deleting a victim node v and its adjacent edges from a WFG, the following issues have to be addressed:

The set R of inward colors of v is removed completely from the cycle-break group Gv , v is removed completely from the essential downstream edges of v. The former means that all wait-for dependencies passed through the victim node v must be removed completely. Because after v is removed, any node k E US(v) may not wait for nodes w E DS(v),and the wait-for information stored in any such w should be updated accordingly. The latter states how the wait-for information should be modified. The claim of the lemma can now be proven as follows: (only if) This direction is obvious, since a correct algorithm should erase the victim's color from every essential down-stream edge.

(if) Down-stream informality actually says that a color c E R will be removed from an edge e E DS(u) if and only if there are no paths which can be used to pass a c-colored probe to e other than through v. Informality, thus, guarantees to remove the dependency between U and its down-stream correctly.

By Lemmas 3 and 4, we have: Theorem 2 (Correct detection/resolution requirements)

A cycle detection and resolution algorithm is correct if it satisfies the following two conditions:

1. The algorithm calculates all WFSs or all cycle-break groups. 2. When a cycle is broken, it has down-stream informality. 4 Efficient Dynamic Algorithms When a WFG is dynamically built and modified according to the propagation of probes, it is impossible to know in advance which nodes will participate in a cycle and therefore become candidates for victims. On the other hand, no more than n victims can be chosen in a WFG that has n nodes. Even though there are no more than n - 1 cycle-break groups, in order to keep track of all possible cycles and non-cycles we will assign each node a unique color. We first introduce a correct cycle detection/resolution algorithm which stays within the upper bounds established earlier. The algorithm is then modified to reduce its average cost to one half the worst case cost by introducing priorities.

4.1 A Naive Algorithm

Every node is assumed to have a unique color. A node generates a coloring probe for each outgoing edge at the time the edge is fist generated. A coloring probe is received and forwarded unless the received color is identical to the color of this node, in which case a cycle is detected. Next, a cleaning probe is generated by the detector which itself is the victim. Now note that every edge in a WFG can be colored up to n times for a WFG with n nodes. Thus, in order to color all cycles in a dynamic setting, n2(n - 1) coloring probes are needed, which is still O(n3). Corollary 2 To dynamically color and detect all the cycles in a WFG G = (KE) , where IV(= n, at most O(nS) coloring probes are needed.

Corollary 3 To break all possible cycles in a dynamic environment, the number of cleaning probes is O(n3) in the worst case.

The following algorithm is divided into two parts, which are to be executed in an alternating fashion. Part 1 deals with detection, while Part 2 handles resolution. As in the static case, a node is considered to be active if it has no outgoing edges, and it is considered blocked otherwise. We use the following notation:

The set of inward colors of node a, i.e., the union of the colors of each incoming edge of i, is denoted e;, and the set of the colors in an edge (k, i) is denoted coZ(k, i). Also, let 0; be the union of the color of node i and its inward colors 0;. Each outgoing edge of node i has exactly the same color set as @i.

Algorithm I: Cycle Detection

Initialization Every node i is assigned a unique color r; Probe Initialization If a node i is blocked by another node j, an edge (i, j) is created, and for each color r, E Oi, node I sends a coloring probe to this (i, j) edge. at edge (k, i), the following steps are performed:

Probe Propagation When a probe r, is received by a node i

1. check for cycles. A cycle is found if a node receives its own color back. If a cycle is found, the probe propagation phase is terminated and the cycle resolution protocol is executed. Otherwise, continue with the next step.
2. r'' e {r,} - 0;
3. coZ(k,i) -+ rpUcoZ(k,i)
4. Forward the new probe r', to every outgoing edge if

{+,} # 0. Otherwise, or if there are no such edges, rlp is discarded.

Algorithm I: Cycle Resolution

Cleaning Probe Initialization The node i which detects the cycle removes itself from the cycle by switdring to the aborting phase: For every color k E @; a cleaning probe is created and sent to all its outgoing edges; then the aborting node i deletes all its adjacent edges and itself from the WFG.

Cleaning Probe Propagation When an node I receives a cleaning probe r, through the edge (IC, i), the following steps are executed:

1. coZ(k,i) -+ coZ(h,i) - r,
2. r', + {r,} - 0;
3. if there are no outgoing edges then r', is discarded, and the probe propagation is terminated. Otherwise, the next step is executed.
4. Forward the new probe r> to every outgoing edge if {ri} # 0; otherwise r> is discarded.
5. a new owner is elected, if available, from the waiting nodes, and a new cycle detection phase starts by entering the probe initialization phase for all waiting nodes.

It should be noted that when a node is in the phase of either committing or aborting, it will not forward any probes.

Theorem 3 Algorithm I is correct.

Proof. To show that Algorithm I is correct, we first show that it detects all cycle-break groups; then we show that it possesses down-stream informality. Since edges and probes can be created only due to resource waiting (probe initialization), a node receiving its own color indicates that there is a cyclic wait condition, and a cycle-break group is found. Step 3 of the cycle detection phase assures that the edges faithfully record all passing colors.

Step 3 combined with Step 2 optimizes the algorithm; Steps 2 and 4 guarantee that the color of every cycle-break group will be forwarded along the group's wait-for paths. Thus, all WFSs will be properly visited; hence, the algorithm detects all cycle-break groups.

To see that Algorithm I possesses down-stream informality, we notice that a victim passes every color which it has received (including its own) to its down-stream. Step 2 of the resolution phase forwards a cleaning color if and only if it is essential. Thus, Algorithm I has down-stream informality.

Theorem 4 Algorithm I achieves the upper bounds given previously.

Proof. During the detection phase, Step 2 allows only new colors to pass through each node. Therefore, each color probe can paint an edge only once. Hence, at most O(n3) probes are transmitted, assuming the wait-for system contains n nodes. In the resolution phase, Step 2 assures that a cleaning probe of a particular color can traverse any edge only once. Hence, the number of cleaning probes transmitted is also O(n3). Our claim follows.

Corollary 4 For an exclusive-lock only wait-for system, the minimum number of probes for the worst case is O(n2).

Proof. If we allow exclusive locks only in the wait-for system, there is at most one outgoing and one incoming edge for each node, which implies that there are O(n) edges in total. Thus, only O(nz) probes are needed to detect and resolve all possible cycles. 'This algorithm is similar to the one given in [13]; however, with the optimization in Step 4, we cut the number of the probes needed from exponentidy many to O(n'). 'The owner of a resource is a transaction not blocked at this resource.

4.2 A Priority Based Algorithm

We next describe how Algorithm I can be improved. To this end, we observe that if a cycle involves nodes kl, kz,. . . , k,, only one node needs to send a probe and detect the cycle. It remains to determine which one should be selected as the sender of this probe. If a total ordering3 is imposed on the colors of all nodes and a color Tk is forwarded by node i if and only if t k > r; holds, it can be shown that this prioritized protocol needs approximately one half of the amount of probes transmitted by Algorithm I. Note that similar approaches have been proposed in [2,14,17,20].

Lemma 5 For a totally ordered cycle, only the probe with the highest priority detects the cycle.

Proof. First note that for every cycle there is a node which has the highest priority, because the nodes are totally ordered. Next we assume each node assigns the probes initiated from it with its own priority. To detect a cycle, a probe must walk through every node of the cycle. If a probe a is passed through every node, then, according to the probe propagating rule, r; > t k , for every k in the cycle, and k = i. Thus, t; must have the highest priority.

Theorem 5 The average number of probes needed to detect deadlocks for a prioritized algorithm is one half of the amount compared to naive algorithm.

Proof. Without loss of generality, we assume that there are n nodes, and these are assigned the priorities { 1, . . ., n }, where 1 is the highest. The number of times each edge leaving a node of highest priority can be colored is one. Similarly, this number is two for the node with the second highest priority, and so on. Hence, the total number of probes is (1 +- 2 +- . . . +- n) .n - 1. The last n - 1 indicates that there are n - 1 possible outgoing edges per node.

We next modify Algorithm I by replacing the former Step 2 of the Probe Propagation Phase with a new one as follows:

Algorithm 11: Cycle Detection

Initialization (as in Algorithm I) Probe Initialization (as in Algorithm I)

Probe Propagation When a probe rp is received by a node a at edge (k, i), the following steps are performed: check for cycles. A cycle is found if a node receives its own color back. If no cycles are found then continue; otherwise, start the resolution protocol.

if rp < col(i) then discard rp; otherwise, fp e {tp} -oi.

coZ(k, i) -e r; U coZ(k,i)

if rp > r; then forward rp to every outgoing edge of node i. The cycle resolution protocol is the same as for Algorithm I.

Corollary 5 Algorithm 11 is correct.

Proof. The only difference between Algorithms I and 11 is that 'A total ordering can be created using Lamport's algorithm [lo]. the priorities are introduced in Step 2 of the detection phase.

Thus, by Theorem 2 and Lemma 5, Algorithm I1 is correct. Corollary 6 The priority based algorithm achieves the upper bounds.

Proof. The claim follows immediately from Theorem 5.

5 A Taxonomy of Deadlock-Handling Algorithms

In order to study the complexity of distributed deadlock detection and resolution algorithms proposed previously and to compare them to our approach, we divide them into two groups according to how the wait-for information is passed by the detection and resolution protocols. We further divide each group into two subgroups, and for each subgroup, we briefly analyze its computation and storage costs.

5.1 Structured Protocols

Protocols which pass probes that contain structural wait-for information are called strrcturedprotocols. By using the structural wait-for information collected from the probes, a partial wait-for graph can be constructed at each node, and these nodes can use their WFGs to determine if a deadlock cycle is formed and which parties are involved. Structured protocols can be further classifled according to the degree of structural information held by probes:

(1) Wait-For Strings (WFSs): Every node passes the probes which contain WFSs to its down-stream. The algorithms reported in [7,14,2,20]fa ll into this class. A deadlock is detected if a node h d s a cycle in a WFS that just arrived.

The advantage of using WFSs is that deadlock detection is easy, and that all parties involved are instantly known to the detector. An obvious disadvantage is that WFSs have variable lengths which makes probe transmission and storage more difficult.

Given a WFG G = (V, E) with n nodes, there may be O(n!) cycles by Lemma 1; hence, there exist O(n!) possible wait-for strings. According to the algorithm, each wait-for string is allowed to pass through a node only once. This means that each node can be traversed approximately O(n!) times, so that O(n!n) probes are needed to detect all possible deadlocks. With respect to space required, the size of a wait-for string is limited to be less than or equal to n, and there are only n! possibilities, because

the merge option always keeps the longest WFS with the same permutation. Therefore, O(n!n) space is needed. On the other hand, deadlock resolution is cheaper w.r.t. the number of cleaning probes needed: Since only the color of the victim is included in the probe, no matter how many wait-for strings are stored in an edge, the total number of the colors contained in an edge does not exceed n - 1. Now it is easily versed that the maximum number of probes needed to resolve all deadlocks is O(n3).

As an example, consider Figure 3: Node N, receives the three WFSs < Nl,N3 >, < N5,N3 >, and < N2,N3 >, and node Nz has WFSs < NI, N3 >,< N5, Ns >, and < N3 >.

Figure 3: A Wait-For System and its Dependencies.

(2) Wait-For Edges (WFEs): In this class, probes contain the wait-for edges of nodes only. A deadlock is detected if an incoming edge contains the color of the receiving node. [ll] uses this approach to build a partial global WFG at each site from which a transaction joins the system. In [3] this scheme is used to construct a WFG in the second part of the detection phase. This approach has the advantage that every probe has a k e d size. For instance, in Figure 3 node N2 receives (NI, N3),

(NS,N3), (N3rN2), and node N4 receives (Nl,N3), (NZ,N3), (N3, N4), and (Ns, N3). Whether the number of probes passed in this approach exceeds the WFS protocol described above actually

depends on the topology of the wait-for system. The complexity of this class for the worst case is better than that of the WFS class. To detect deadlocks, each edge lets no more than O(n2) probes pass. This is due to the fact that no edge is forwarded which has been encountered earlier, and there are at most O(n2) edges. Thus, the total number of probes needed to detect all deadlocks is O(n4).

To resolve deadlocks, the same complexity is obtained. However, this can easily be improved as follows: Since the wait-for strings can be reconstructed from the wait-for edges, only the victim's color needs to be sent. Thus, the cost of resolution is identical to that of the class WFS. Finally, the space requirement is O(n4), since each edge may store up to O(nz) edges.

Because edge-queues only store the colors of nodes, the space required by this class is O(n3). Also, each edge allows up to n different colors to pass through, so that the total number of probes needed cannot exceed O(n3) for either deadlock detection or resolution.

(2) Time-Stamped Wait-For Pools (TSWFPa): Each time a node initiates a probe, it attaches its color and a timestamp to it. Both timestamp and color are used to determine if a deadlock has occurred. [3] suggests such a scheme in the first part of the detection phase. One purpose of timestamps is to solve the problem which arises when the wait-for information is not persistent', another is to distinguish the wait-for information generated by different wait-for edges of the same node. Using timestamps, the original sender can detect if a received probe initiated by the node is out-of-date, and (hopefully) distinguish false deadlocks from real ones. However, as explained above, even for an

algorithm which employs two-phased locking, which guarantees the durability of the wait-for dependency, this is not sufficient to guarantee freedom from false deadlocks.

The only difference between this class and the class WFP is that in this class the probes are tagged with timestampes. For this reason, each edge will accept up to O(nz) probes, and the total number of probes needed -is O(n4) [12]. Since each queue can accumulate no more than O(n) different colors, the regenerating WFSs which puts their algorithm in the exponential class.

5.2 Non-Structured Protocols

If probes contain no structural wait-for information, the corresponding algorithm is called non-structured. In this case no

waitfor graphs can be constructed. However, this does not prevent
a deadlock/resolution algorithm to detect and resolve deadlocks.
Non-structured protocols can be classified as follows:
(1) Wait-For Pools (WFPs): Only the nodes' colors are put into the probes and propagated. Since a deadlock is detected by finding that a probe returns to its original sender, algorithms adopting this method only need to see if the color of a node is in the "pool" of received probes. This pool represents the wait for relationships space requirement is O(n3). Note that [3] resolves deadlocks by
between the receiving node and the nodes whose colors are in the pool, even though the detailed wait-for structure is not known. Our algorithms described in Section 4 fall into this class. The approach is not only conceptually simple, but also gives the best performance. The size of a probe can be either fixed or variable. This scheme is also partially implemented by the naTve protocol suggested in [13]. Because edge-queues only store the colors of nodes, the space required by this class is O(n3). Also, each edge allows up to n different colors to pass through, so that the total number of probes needed cannot exceed O(n3) for either deadlock detection or resolution.

(2) Time-Stamped Wait-For Pools (TSWFPa): Each time a node initiates a probe, it attaches its color and a timestamp to it. Both timestamp and color are used to determine if a deadlock has occurred. [3] suggests such a scheme in the first part of the detection phase. One purpose of timestamps is to solve the problem which arises when the wait-for information is not persistent', another is to distinguish the wait-for information generated by different wait-for edges of the same node. Using timestamps, the original sender can detect if a received probe initiated by the node is out-of-date, and (hopefully) distinguish false deadlocks from real ones. However, as explained above, even for an
algorithm which employs two-phased locking, which guarantees the durability of the wait-for dependency, this is not sufficient to guarantee freedom from false deadlocks.
The only difference between this class and the class WFP is that in this class the probes are tagged with timestampes. For this reason, each edge will accept up to O(nz) probes, and the total number of probes needed -is O(n4) [12]. Since each queue can accumulate no more than O(n) different colors, the space requirement is O(n3). Note that [3] resolves deadlocks by regenerating WFSs which puts their algorithm in the exponential class. 6 Conclusions In this paper, deadlock detection and resolution in distributed systems has been considered from the point of view of correctness
and of efficiency. To this end, upper bounds for the complexity of corresponding algorithms have been derived, correctness criteria have been established, and appropriate algorithms have been
presented. As we have pointed out earlier, most incorrect algorithms work correctly until a deadlock is detected and resolved. As we have seen, this problem stems from the lack of down-stream informality. In order to detect new deadlocks correctly after old ones have been resolved, the resolution protocol must possess down-stream informality which eliminates nothing but the essential down-stream dependencies. Thus, we can easily determine whether a deadlock detection/resolution algorithm is incorrect by first checking whether or not it has down-stream informality.
In some papers it is stated that deadlock cycles are short in general [6], so that all algorithms perform equally well in such a situation. However, it remains unclear whether the average
length of WFSs which are not deadlocked is short, or whether the average number of the wait-for edges is small in a large data base with thousands of transactions [8].
We have shown that if the goal is to detect and distinguish every deadlock cycle, O(n!) probes are needed for n node^.^ If we do not wish to identify each individual deadlock cycle, we only need O(n3) probes to detect deadlocks. We have exhibited 'for example, if the system allows a transaction to release a lock while the transaction is blocked.
An O(R') deadlock detection algorithm which can identify individual deadlocks, if a node is willing to reconstruct a partial WFG internally. However the, computation to reconstruct the partial WFG is not included in the complexity of this algorithm.

Two algorithms which achieve these bounds by using colors to propagate probes, and by carefully maintaining the probes circulating between nodes. The advantages of our algorithms over previously proposed ones include that resolution probes leave the maximal amount of correct information behind, thereby saving work in subsequent detections, and that equivalent messages can be identified so that no forwarding of duplicates is done.
On the other hand, it is not always satisfying to let the detector of a deadlock become the victim for the required resolution; another problem is that several nodes may detect the same deadlock
Simultaneously (in the absence of priorities) and, as a consequence, several transactions are aborted unnecessarily. In order to eliminate these drawbacks, two solutions seem feasible: First, global information could be maintained, which requires a central monitor, but this can easily result in new problems with respect to the availability of the monitor, for example. Second, semantic information on the individual transactions could be employed if available, thereby rendering it possible to choose a victim more carefully, and to avoid the multiple detection of a deadlock. To this end, works could be relevant, where semantic information on the underlying data model, the transactions itself, or on the types of locks was shown to improve concurrency control. These questions deserve further study.

1. **References**
2. Aho, A.V., Hopcroft, J., Ullman, J.D.: The Design and
3. Analysis of Computer Algorithms; Addison-Wesley Publ.
4. Co. 1974.

5. Badal, D.Z.: "The Distributed Deadlock Detection Algorithm";
6. ACM Transactions on Computer Systems 4,1986,
7. 320-337.
8. Chandy, K.M., Misra, J.: "A Distributed Algorithm for
9. Detecting Resource Deadlocks in Distributed Systems";
10. PTOC. 1st ACM Symposium on Principles of Distributed
11. Computing 1982, 157-164.
12. Choudhary, A.N., Kohler, W.H., Stankovic J.A., Towsley,
13. D.: "A Priority Based Probe Algorithm for Distributed
14. Deadlock Detection and Resolution"; Proc. 7th ZEEE Int.
15. Conference on Distributed Computing Systems, Berlin,
16. West Germany, 1987,162-168.
17. Elmagamid, A.K.: "A Survey of Distributed Deadlock
18. Detection Algorithms"; ACM SIGMOD Record 15 (3)
19. 1986), 37-45.
20. Gray, J., Homan, P., Obermarck, R., Korth, H.: "A Straw
21. Man Analysis of Probability of Waiting and Deadlock";
22. IEM Research Report RJ 3066, San Jose 1981.
23. Haas, L.M., Mohan, C.: "A Distributed Deadlock Detection
24. Algorithm for a Resource-Based System"; IBM Research
25. Report RJ 3765, San Jose 1983.
26. Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan,
27. M., Sidebotham, R., West, M.: "Scale and
28. Performance in a Distributed File System (extended abstract)";
29. Proc. 11th ACM Symposium on Operating Systems
30. Principles, Austin,