Deadlock detection in distributed database systems: A new algorithm and a comparative performance analysis

Natalija Krivokapić[1], **Alfons Kemper**[1], **Ehud Gudes**[2]

[1] Universität Passau, Lehrstuhl für Informatik, 94030 Passau, Germany; e-mail: <lastname>@db.fmi.uni-passau.de>
[2] Ben-Gurion University of the Negev, Department of Math. & Comp. Science, Beer-Sheva, 84105, Israel; e-mail: ehud@indigo.bgu.ac.il

Abstract. This paper attempts a comprehensive study of deadlock detection in distributed database systems. First the two predominant deadlock models in these systems, and the four different distributed deadlock detection approaches are discussed. Afterwards, a new deadlock detection algorithm is presented. The algorithm is based on dynamically creating *Deadlock Detection Agents* (DDAs) each being responsible for detecting deadlocks in one connected component of the global Wait-For-Graph (WFG). The DDA scheme is a "self-tuning" system: After an initial warm-up phase, dedicated DDAs will be formed for "centers of locality", i.e., parts of the system where many conflicts occur. A dynamic shift in locality of the distributed system will be responded to by automatically creating new DDAs while the obsolete ones terminate. In this paper we also compare the most competitive representative of each class of algorithms suitable for distributed database systems based on a simulation model, and point out their relative strengths and weaknesses. The extensive experiments we carried out indicate that our newly proposed deadlock detection algorithm outperforms the other algorithms in the vast majority of configurations and work loads and—in contrast to all other algorithms—is very robust with respect to differing load and access profiles.

1 Introduction

During the last decade computing systems have undergone a rapid development, which has a great impact on distributed database systems. While commercial systems are gradually maturing, new challenges are imposed by the world-wide interconnection of computer systems. This creates an ever growing need for large-scale enterprise-wide distributed solutions. Mariposa [SAL+96] is a recent prototype of a system addressing this demand. In future, distributed database systems will have to support hundreds or even thousands of sites and millions of clients and, therefore, will face tremendous scalability challenges with regard to performance, availability and administration.

Deadlocks can arise in each database system that permits concurrent execution of transactions using pessimistic synchronization schemes, i.e., locking protocols, which is the case in most of todays (distributed) database systems. In centralized database systems deadlock detection and resolution has been thoroughly investigated, e.g., in [ACM87]. Deadlocks have also been studied in other areas, such as operating systems.

Surveys of earlier work on distributed deadlock detection in distributed database systems are given in [Kna87, Elm86, Sin89]. The surveys described different algorithms, but no quantitative analysis, in terms of benchmarking, has been carried out.

First we describe the two predominant deadlock models underlying locking-based database transaction synchronization. Then, the four different distributed deadlock detection approaches are briefly surveyed: timeout, path-pushing, probing and global state detection schemes, and representatives of these classes are described.

To reflect the new developments, in this paper we present a new deadlock detection algorithm designed for distributed object systems. Also a comprehensive simulation study of different deadlock detection algorithms is given.

In our computational model transactions are carried out under the control of a transaction manager; synchronization is achieved by a two-phase locking scheme, which could, for

example, be based on semantic locking. The locking scheme is entirely under the control of the object managers. The execution and commit processing of transactions is controlled by the transaction managers.

The proposed algorithm detects deadlocks by dynamically creating *Deadlock Detection Agents* (DDAs). Each DDA maintains one part of the wait-for-graph (WFG) and searches for cycles. In the resource model—the underlying deadlock model of database transactions—a cycle always constitutes a deadlock. Transactions start executing without any DDA. Only if a conflict with some other transaction(s) occurs a transaction becomes associated with a DDA: either a DDA one of the conflicting transactions was already associated with or, if no such DDA exists, a newly created one. If two transactions that are already associated with different DDAs encounter a conflict, their two DDAs are merged into a single one. This scheme guarantees that all—but only "real"—deadlocks will be detected, which is one of the main problems distributed deadlock detection algorithms face.

In a large-scale distributed database system one can expect that centers of locality will be formed. These centers of locality comprise those transactions that access (compete for) the same objects. The DDA scheme automatically establishes different DDAs for different centers of locality—thereby decentralizing the deadlock detection.

The DDA scheme is a "self-tuning" system: After an initial warm-up phase, dedicated DDAs will become established for each center of locality. Moreover, if a shift in the system's load is encountered—e.g., because of a shift of activity between different time zones in the course of a day within an enterprise-wide distributed system—the DDA scheme adapts automatically. New DDAs will be created for the newly forming centers of locality and the obsolete DDAs will eventually terminate.

In order to evaluate the performance of different schemes we built a simulation system and chose to implement representatives of different classes of algorithms that seem to be the most efficient ones and are suitable for distributed database systems. We have implemented the following algorithms: 1) the DDA approach as the representative of global state detection algorithms, 2) the edge-chasing algorithm described in [RB89], that generates the fewest messages, 3) the often cited path-pushing algorithm of System R* [Obe82] and 4) different timeout approaches.

The rest of the paper is organized as follows. In Section 2 we outline the underlying model of computation. Section 3 gives an overview of the deadlock problem and the two deadlock models that mainly apply in distributed database systems. A classification of distributed algorithms for these models and representatives of the different classes are described in Section 4. In Section 5 the new deadlock detection scheme is presented. The performance comparison of the different algorithms is reported in Section 6. Finally, Section 7 concludes this paper.

2 The Model of Computation

The investigation of distributed deadlock detection algorithms presented here is based on a very general distributed database model, consisting of a collection of sites on which transactions and objects are residing. The execution and commit processing of transactions is controlled by *transaction managers* (TMs), while objects are controlled by *object managers* (OMs). An OM receives requests for operation executions upon the objects it controls, sent by TMs, and communicates the objects' answers back to the TMs. In order to simplify the presentation, but without loss of generality, we assume that each TM and OM controls only a single transaction and object, respectively. Therefore, we will often synonymously use the term transaction for TM and object for OM.

In our model each object and transaction has a unique identifier. Many deadlock detection algorithms require a total ordering on transactions for deadlock resolution and assume that the transactions' identifiers can be used for this purpose, e.g., to determine the youngest transaction. However, after a transaction is aborted it has to be restarted with a new identifier, otherwise information regarding the aborted and the restarted execution of the transaction could not be distinguished—possibly leading to inconsistencies. However, changing the identifier could alter the ordering of transactions, e.g., an old transaction might become the youngest. To avoid this, in our system a transaction is associated with a timestamp (additionally to the identifier), indicating the time it has entered the system, which is not modified after an abort and can therefore be used for transaction ordering. For simplicity reasons in the rest of the paper we use identifiers for the ordering of transactions and assume that they contain such a timestamp.

Transactions and objects communicate via asynchronous message passing. Transactions send, through their TMs, requests to objects, i.e., to their OMs, which, in turn, send acknowledgments (and/or results) back to the TMs.

We assume an error-free communication, meaning that each message sent arrives within finite time and is transmitted correctly. This assumption is needed by deadlock detection algorithms, since they gather information through messages. If messages would get lost, deadlocks might not be detected. In systems in which this cannot be guaranteed, e.g., when parts of the system are connected through an unreliable network, transactions have to set a timeout and abort after it expires. This is similar to what is done in other systems using network communication, e.g., the WWW, ftp, etc.. This timeout to handle communication errors should not be confused with the timeout approach to detect transaction deadlocks. The communication timeout is supposed to handle the rare cases in which a message cannot be delivered and, therefore, is set to a much larger value than the timeout handling deadlocks.

A two-phase locking protocol is assumed: During the first phase TMs request operation executions on (possibly) different objects; the acquired locks are maintained by the OMs on behalf of the holding transaction. In the second phase, the

TMs initiate the commit processing after which the locks are released.

A transaction T_i consists of a sequence $[O_1.op_{i_1}(\ldots); O_2.op_{i_2}(\ldots); \ \ldots]$ of operation invocations $op_{i_j}(\ldots)$ on one or several objects O_j—the "$(\ldots)$" denotes additional parameters of the invocation. The corresponding TM requests the execution of the operations in strictly sequential order; that is, it first sends a message to object O_1 requesting the invocation of operation $op_{i_1}(\ldots)$. The next request—i.e., the one to object O_2 requesting the execution of $op_{i_2}(\ldots)$—is not sent before the TM receives an acknowledgment from O_1 that the requested invocation $op_{i_1}(\ldots)$ has been successfully executed. Thus, transactions are single-threaded.

An object manager receives the requests for operation invocations on its object and schedules them in some order (e.g., FIFO). The object manager maintains the lock table and a compatibility matrix in which the commutativity of any two operation invocations is maintained. Based on this information the OM decides whether or not the locks needed for the requested operation can be granted. If not, the operation invocation is delayed and, thereby, the particular transaction's execution is blocked until the required locks are available. If the locks can be granted, the OM associates the lock mode with the corresponding transaction and invokes the requested operation.

We adopt a semantic lock model [SS84, Kor83], which is more flexible than an exclusive or read/write lock model. With the exclusive lock model, an object can be accessed by only one transaction at a time. The read/write lock model, in contrast, allows multiple readers to hold locks on an object at the same time while writers need exclusive locks.

The semantic lock model exploits the semantics of operations, i.e., their commutativity, to increase the possible concurrency, so even multiple transactions updating a data item can concurrently hold locks on it. Consider an object *bank* consisting of a set of *accounts*. With semantic locking, if transaction T_1 has made a deposit to $acct_1$, transaction T_2 could also deposit money to this account, before T_1 releases the corresponding lock. This is possible since two deposit operations commute, i.e., their order of execution is irrelevant.

3 The General Deadlock Problem

In the vast majority of modern database systems concurrency control is based on locking mechanisms. Most systems employ the *strict* 2PL protocol [GR93]. Locking protocols can lead to deadlocks. A *deadlock* is a permanent, circular wait condition. A set of transactions is *deadlocked iff* each of the transactions waits for locks held by other transactions from this set [BHG87]. All transactions from the set are in a waiting state, i.e., are blocked, and none of them will become unblocked without interference from outside.

Different locking models can be used by concurrency control algorithms. In the *exclusive* and the *read/write* lock models a transaction waiting for a lock to be granted has to wait for *all* transactions currently holding locks on that object. When semantic locking is employed, a transaction may have to wait for only a subset of the holders of the object. Also, different transactions being blocked on the same object may have to wait for different subsets of holders of the object.

Handling deadlocks involves two problems: *deadlock detection* and *deadlock resolution*. In a DBMS deadlock resolution means that one of the participating transactions, the *victim*, is chosen to be aborted, thereby resolving the deadlock.

A deadlock detection algorithm is correct if it satisfies two conditions: 1) every deadlock is eventually detected (*basic progress property*), and 2) every detected deadlock really exists, i.e., only genuine deadlocks are detected (*safety property*).

While the first condition is intuitive, the second one needs to be explained. Although a deadlock is a stable property, due to stale information it is possible that the same deadlock is detected and/or resolved twice. Detected deadlocks that do not really exist (anymore) are called *phantom deadlocks*, see [KS94b]. One could argue that an algorithm detecting phantom deadlocks could still be considered correct; but unnecessary transaction aborts are too expensive to be tolerable.

Each deadlock detection algorithm may detect phantom deadlocks if *spontaneous aborts* are permitted. If an algorithm decides to abort a transaction in order to resolve a deadlock, and at the same time some other transaction involved in the same deadlock aborts spontaneously—thus resolving the deadlock—the algorithm is breaking a phantom deadlock. Therefore, we will assume that no spontaneous aborts occur in the system.

3.1 Wait-For-Graph

Blocking conditions between transactions can be represented through a transaction wait-for-graph (WFG). A WFG is a directed graph in which nodes correspond to transactions and a directed edge from T_i to T_j expresses that T_i waits for a resource currently held by T_j. A deadlock can be detected by examining the structure of the WFG. Which graph structures indicate a deadlock depends on which deadlock model applies, as described in the next section.

3.2 Different Deadlock Models

Depending on the computational model, mainly on the types of requests made by transactions, different deadlock models apply. In distributed DBMS the *single resource* and the *AND* model are prevailing, so only those will be discussed here. Descriptions of the other models such as the OR model and the general model, which are much less common, are given in [BO81, MC82, Kna87, KS94a, BHRS95].

The simplest, and most widely used model in DBMSs is the *single resource* model. In this model a transaction has only one outstanding request at a time, i.e., it requests a lock on one object, waits until it is granted and only afterwards

requests a lock on the next object. Although a transaction has only one outgoing request at a time, it may wait for more than one transaction. [Kna87] erroneously concludes that in the single resource model a transaction can wait for only one other transaction, implying that each node in the WFG has only one outgoing edge. This is true only for the exclusive locking model.

A deadlock in the single resource model corresponds to a cycle in the WFG. Therefore, algorithms for this model declare a deadlock when a cycle of waiting transactions is determined. The cycle is resolved if one of the involved transactions is aborted, thereby releasing its locks.

Numerous algorithms have been proposed for this model [Obe82, RBC88, Bad86, SH89, MM79, KS91]. Some of these algorithms will be described in Section 4.

In a computational model in which a transaction can send more than one request at a time and has to wait until *all* of them are granted, deadlocks are described by the *AND* (or *resource*) model. This model applies, for instance, in systems supporting nested transactions, or when a transaction can request locks on several objects simultaneously. A deadlock in this model is again indicated through a cycle in the WFG.

As far as we can see, some of the algorithms designed for the single resource model could easily be extended for deadlock detection in a model in which a transaction can issue multiple requests at a time. Since the models are so similar sometimes the authors claim that their algorithms are for the AND model, although the computational model they describe is a single resource one, e.g. [RBC88], while others make restrictions to their model basically reducing it to a single resource model [Obe82]. An algorithm for deadlock detection in the AND model is given in [LK95], but it is not clear how deadlocks can be resolved with this algorithm. Algorithms especially designed for DBMSs supporting nested transactions are [Ruk91, RHGL97].

4 Distributed Deadlock Detection Algorithms

Numerous deadlock detection algorithms have been developed for distributed DBMSs, surveys can be found in [Kna87, Elm86, Sin89]. This paper focuses on distributed algorithms for the single resource and the AND model only. We classify the distributed algorithms based on the technique they use, similar to the classification proposed in [Kna87], explain the different techniques and describe one representative of each class in more detail. We chose those algorithms that appear to be the best in their class, i.e., induce the fewest number of messages and are correct or detect the fewest phantom deadlocks in their class. First, in Section 4.1 we will take a look at deadlock resolution strategies.

4.1 Deadlock Resolution

Deadlock resolution strategies determine which transaction(s) are to be aborted in order to resolve the deadlock(s). Many resolution strategies have been proposed for centralized systems. Some of them have been compared by [ACM87], in a simulation study of a centralized system. Based on the results of the simulation, two conditions were determined each resolution strategy should guarantee: 1) it is always guaranteed that at least one transaction in the system can finish (*guaranteed forward progress*), and 2) no transaction will be aborted (and restarted) an indefinite number of times (*no indefinite restarts*).

The first condition is related to the *basic progress property* stated in Section 3. This property guarantees that no deadlock will exist forever but it does not ensure that any transaction will ever finish. This is ensured by the first of the above conditions, by guaranteeing that the transaction processing will progress. A deadlock detection and resolution algorithm can achieve what is intuitively expected, i.e., that every transaction that enters the system will eventually finish, *only* if the deadlock detection is correct, i.e., fulfills the conditions given in Section 3, *and* the resolution strategy guarantees the above conditions. Obviously the given conditions also hold for distributed DBMSs.

4.2 The Timeout Approach

In this algorithm a transaction sets a timeout every time it makes an operation request. If it does not receive the acknowledgment that the operation has been executed successfully before the timeout expires, it assumes that it is involved in a deadlock and aborts.

The algorithm is simple and therefore easy to implement. Also, it does not cause any network traffic due to deadlock detection. The main disadvantage of the algorithm is that it aborts too many transactions. The algorithm aborts transactions that may not be deadlocked, thus causing unnecessary roll-backs and restarts of transactions.

Another disadvantage is that the timeout interval has to be tuned. If it is too short even more transactions are unnecessarily aborted, if it is too long deadlocks will persist in the system for a long time, thus delaying transactions in the deadlock and those waiting for locks held by them. Therefore, the timeout interval has to be chosen carefully, which is difficult when applications of widely differing profiles are running in the system. Usually it is set to be much longer than the average execution time of a transaction [BN97, Hof94].

In [BN97] it is said that it may sometimes be desirable to abort too many transactions, since if a transaction waits for a longer time than the timeout interval although it is not deadlocked, this indicates that the locking load is too high. This may be true but a deadlock detection algorithm should not be responsible for the scheduling.

Another drawback of this algorithm is its “resolution strategy”. The timeout scheme can neither guarantee that one transaction can finish, nor that a transaction will not be aborted an indefinite number of times. In particular, when the load is high, long transactions have no chance to get through.

However, the algorithm performs surprisingly well in distributed DBMSs[1] if it is carefully tuned, and therefore, many systems have implemented it because of its simplicity [BN97].

To alleviate the drawbacks of the timeout approach, some systems, e.g., Oracle [LMB97], introduce a deadlock detector at each site, responsible for detecting local deadlocks. Deadlocks involving not only local transactions are resolved through a timeout.

4.3 Classification of Distributed Algorithms

Many different distributed deadlock detection algorithms for distributed DBMSs have been published. They can be divided into the following categories (cf. [Kna87]): 1) path-pushing, 2) probe-based, and 3) global state detection algorithms.

Before we describe the different classes and algorithms, we will summarize the assumptions we make. They have been explained in previous sections, here we recall them. The first two assumptions regard the behavior of transactions: They follow the 2PL protocol, and do not spontaneously abort. Further an error-free communication is assumed, meaning that each message reaches its destination within finite time and that each received message is correct. Of course, algorithms also have to assume that there is a total ordering on transactions so they can choose a victim to resolve a deadlock.[2] Some algorithms make additional assumptions that will be pointed out in the algorithm's description.

4.4 Path-Pushing Algorithms

Path-pushing algorithms explicitly maintain the WFG. Each site periodically collects local wait dependencies, builds a local WFG, searches for cycles in it, and resolves the cycles it detects. Parts of the rest of the WFG are sent to some other (neighboring) sites. They incorporate the received wait dependencies into their local WFG and search for cycles in it. Afterwards the site again passes parts of the WFG on to other sites.

It is interesting to observe that many of the published path-pushing algorithms have turned out to be incorrect. Most of them detect phantom deadlocks while some algorithms even fail to detect real ones. E.g., counterexample to the algorithm presented in [MM79] is given in [GS80].

Obermarck's Algorithm

The algorithm by Obermarck [Obe82] was implemented in System R* [WDH$^+$81] and optimizes the path-pushing strategy by sending a part of a possible cycle, i.e., a "path", to another site only in case the first transaction in the path has a higher priority than the last one. This reduces the number of messages by one half.

Despite this optimization, the algorithm imposes significant overhead when it performs deadlock detection. This is done periodically, so the whole WFG has to be searched for cycles.[3] Additionally, the paths that have to be sent to other sites have to be identified. Also incorporating newly arrived paths from one site implies that the previous information received from this site has to be exchanged in the WFG for the new information.

In [Elm86] it is stated that the algorithm detects phantom deadlocks, since the parts of the WFG sent between the sites belong to asynchronously taken snapshots, i.e., might be inconsistent. Even if the snapshots were taken synchronously, the algorithm might detect false deadlocks, since inconsistencies can arise due to the breaking of cycles by the algorithm itself.

4.5 Probe-Based Algorithms

Probe-based algorithms do not explicitly maintain the WFG but send a special kind of messages, *probes*, along the edges of the WFG. There are two kinds of probe-based algorithms: edge-chasing and diffusing computation.

4.5.1 Edge-Chasing Algorithms

When a transaction T_i requests an operation execution on an object and becomes blocked because it has to wait for locks held by other transactions, a probe is sent to each of these transactions, i.e., T_i initiates a probe computation. A blocked transaction that receives a probe has to forward it to all transactions holding locks it waits for. If a probe initiated by T_i returns to T_i, this probe must have traversed a cycle which constitutes a deadlock.

A transaction that has requested an operation execution does not know whether it is blocked or the operation is currently being processed. Also, if it is blocked it does not know which transactions it waits for. Objects have this information, so in fact they send the initiation probe on behalf of the blocked transaction. For the same reasons a transaction forwards the probes to the object it waits for and the object then sends the probes to the appropriate transactions.

Numerous edge-chasing algorithms have been developed [CM82,CMH83,RBC88,SN85,SH89,KS91,CKST89,LK95], again, some of them turned out to be incorrect, see [RBC88, CKST89, KS91].

We will describe the algorithm by Roesler et al., [RBC88, RB88, RB89], which is an improvement of [CM82, SN85]. The algorithm seems to induce the least amount of messages and detects fewer phantom deadlocks than other algorithms in this class. Moreover, it can handle semantic locking while some algorithms, such as [SN85], can handle only exclusive locking.

[1] For centralized DBMSs it performs poorly, as shown in [ACM87].

[2] In our system the transaction's identifier can be used for this purpose.

[3] When deadlock detection is done continuously it has to be checked only whether the new edge(s) have created a cycle.

Algorithm by Roesler et al.

In order to reduce message traffic in this algorithm objects forward probes only to those transactions that have an identifier[4] lower than that of the initiator of the probe. Probes that are being sent to transactions with a lower identifier are called *antagonistic* probes.

The strategy of sending only antagonistic probes reduces the number of messages and additionally assures that each cycle will be detected only once. Only the probe initiated by the transaction with the highest identifier will be forwarded through the whole cycle.

To reduce the number of probe initiations transactions as well as objects store the probes that they receive in order to forward them when a dependency occurs at some later time. This is necessary in order to avoid periodical re-initiation of deadlock detection, which would drastically increase the number of messages.

However, the stored probes incur a performance penalty: They have to be removed when a deadlock is detected in order to bring the WFG back to a consistent state. Therefore, an object detecting that an antagonistic edge has ceased to exist has to send an (antagonistic) *antiprobe* for each probe that has arrived along the edge corresponding to the disappearing dependency. The antiprobe mimics the behavior of the probe, i.e., it follows the same paths and "cleans up" the WFG.

Sending antiprobes reduces, but does not eliminate, the likelihood of detecting phantom deadlocks [RB88]. A probe may pass along an edge that has ceased to exist before the antiprobe deletes it, thus detecting a phantom deadlock.

The algorithm is designed for the single resource model. It makes the additional assumption that messages arrive in the order in which they were sent.

4.5.2 Diffusing Computation

In algorithms in this class a transaction starts a diffusing computation [DS80,Cha82] when it has to wait for a lock. A deadlock is indicated, if the computation terminates.

A node of a directed graph starts a diffusing computation by sending messages to its successors. Upon receiving such a message a node can send messages to its successors, and so on. In order to determine when a diffusing computation terminates, nodes can also receive signals from their successors and send signals to their predecessor [DS80, Cha82]. Based on the signals it receives, the initiator can decide whether it is deadlocked or not.

Using diffusing computation is an "overkill" for the detection of single resource and AND-model deadlocks. In these models a deadlock is indicated by a probe that returns to its initiator, which induces much less overhead than diffusing computation. Therefore, algorithms based on diffusing computation [KS94a,KS97,MC82] are usually designed for more complex deadlock models for which edge-chasing algorithms cannot be used.

[4] Identifiers are used to denote the priority of a transaction.

4.6 Global State Detection

The main problem distributed deadlock detection algorithms encounter is that information may be stale and/or inconsistent, thus leading to the detection of phantom deadlocks. To avoid this, global state detection algorithms try to obtain a consistent snapshot of the WFG, and search for deadlocks in it.

In [CL85] an algorithm is given that on-line obtains a consistent snapshot of the system. An algorithm for detection of generalized deadlocks using these snapshots is presented in [BT87]. Another algorithm, also for the generalized model, is presented in [CDAS96]. However, this approach appears to be rather inefficient, since each blocked transaction initiates a deadlock detection process.

An algorithm for the AND model is presented in [ESL88]. In this algorithm one TM, e.g., TM_i, will have full control over all transactions that are in the same connected component of the WFG as T_i, and will additionally manage all objects any of these transactions have accessed. TM_i will also maintain the corresponding part of the WFG and search for cycles in it. If the component splits into two (or more) components, TM_i will pass over the control of the component(s) it is not part of to a TM in the new component. Note that a split of the WFG here means that the WFG has split and the sets of objects accessed by transactions from different components are disjunctive.

The algorithm assumes synchronous communication, i.e., when a transaction wants to make a request to an object it has to wait until the object is ready to receive it. Despite this unrealistic simplification, the algorithm incurs a severe overhead: All requests for operation executions to an object on which some transaction holds a lock will have to be forwarded, so sometimes a local request might go to a distant site. Also, a TM will take charge over another TM even when they do not conflict but only access the same object.

In Section 5 we present a new global state detection algorithm. The algorithm creates agents to which objects, i.e., their OMs, report wait dependencies between transactions, so the agents can perform deadlock detection. Different agents are responsible for different parts of the WFG, i.e., different connected components within the WFG and have a consistent snapshot of it. The algorithm is easy to integrate in a system, deals with asynchronous message passing and does not even depend on correct message ordering.

5 Deadlock Detection Agents (DDAs)

The main idea of the algorithm is to distribute the information about the global WFG between different agents, called deadlock detection agents (DDAs), in a way that for every cycle in the graph there is one DDA having complete information about it. Each cycle belongs to one connected component of the global WFG, so for each connected component there will eventually be one DDA responsible for detecting cycles in it.

This is achieved by allowing transactions to be associated with at most one DDA. Since only one DDA detects deadlocks one transaction is involved in, the same deadlock will not be detected twice, which is an inherent problem other algorithms for distributed deadlock detection encounter. By guaranteeing, that for each node in the WFG there is only one DDA having outgoing edges from this node in its part of the graph, the algorithm avoids detecting phantom deadlocks. This will be explained in more detail in Section 5.6. Moreover, a DDA has information about the whole connected component it is responsible for and can therefore even further reduce the number of transactions to be aborted, by deliberately choosing the victim.

The emergence of a new connected component leads to the creation of a new DDA. In case two connected components of the WFG join to one component, the two corresponding DDAs are also merged, and when the part of the global WFG one DDA is responsible for disappears, the DDA terminates.

The algorithm will be presented for the single resource model. We will also show how to extend it for the AND model. The DDA scheme does not require a particular lock model; all it assumes is that there is a component reporting every dependency to a DDA.

In a large-scale distributed database system it is likely that different types of transactions access different groups of objects, thereby creating centers of locality. Dependencies will arise between transactions belonging to the same center, building connected components of the WFG. In an initial warm-up phase dedicated DDAs for these centers of locality will be formed. Due to commits or aborts of transactions one connected component can split up into different connected components. In this case, the corresponding DDA is responsible for cycle detection in all of these components. If the connected components belong to the same center of locality, they will most likely soon merge into one connected component again. If, however, the connected components split because the access profile of the system shifted such that new centers of locality emerge, new DDA(s) will automatically emerge and the previous DDA(s) will eventually terminate.

The DDA scheme combines the advantages of distributed deadlock detection algorithms (e.g., load distribution, localization of deadlock detection in the vicinity of the involved transactions) with the global view of (parts of) the WFG the centralized scheme has—thus getting the best of both worlds. It avoids detecting phantom deadlocks despite its distribution. Moreover, it can dynamically adjust to the system's load and, in particular, it automatically adjusts to shifts in the system's hot spots by forming new DDAs in that vicinity.

The algorithm does not make additional assumptions to those described in Section 4.3, i.e., 2PL, error-free message transmission, and no spontaneous aborts of transactions. For deadlock resolution a total ordering of transactions is needed.

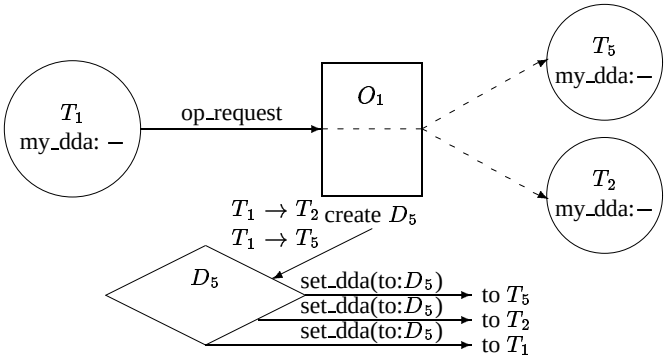

Fig. 1 Creation of a DDA

5.1 DDA Creation

The proposed algorithm dynamically creates DDAs when they are needed. A deadlock can occur only if there exists a conflicting lock on at least one object. As long as there are no conflicting lock requests, i.e., there are no dependencies between the accessing transactions, no DDA is needed. The lock management in the underlying model is done by the objects themselves, i.e., their object managers, hence, they are also responsible for the creation of DDAs.

When a transaction requests an operation on an object on which another transaction already holds a lock that conflicts with the one required for the requested operation, a DDA is needed. If the object has no knowledge of a DDA—even if one of the involved transactions already is associated with one—a new DDA has to be created.[5] The object creates it and passes the dependencies to it. All of the involved transactions become associated with the DDA. Becoming associated with a DDA means that the transaction is sent the identifier of the DDA and that all following operation requests by this transaction will contain the identifier of its DDA. The object that created the DDA records the association of a transaction with its DDA.

An example is shown in Figure 1. T_2 and T_5 are holding non-conflicting locks on O_1, and none of the transactions is associated with a DDA. T_1, also not yet associated with a DDA, requires a lock conflicting with locks held by T_2 and T_5—denoted through dashed arrows. So dependencies $T_1 \to T_2$ and $T_1 \to T_5$ arise. O_1 creates DDA D_5 and passes the dependencies to it. The object records the association of the transactions with D_5. After being created, D_5 sends a message to each of its new transactions, so they become associated with it. After receiving such a message T_1, T_2, and T_5 will send the identifier of D_5 with each of their future operation requests.

If a transaction's request conflicts with a set of locks already granted, and either the requesting transaction or one of the transactions holding the locks already has a DDA, the object knows about, no DDA has to be created. If there is only

[5] A DDA has to have a unique identifier and a total ordering on these identifiers is needed. The same kind of identifiers as it is used for transactions can be employed.

one such DDA, the dependencies are reported to it and all the other transactions will become associated with this DDA. In case the transactions have different DDAs, they have to be merged, as described in the following section.

5.2 DDA Merger

A conflicting lock request can create dependencies that lead to a join of two or more connected components of the WFG into one component. In this case the DDAs responsible for these components have to be merged. The initiation of a merge is explained below. Here the merging of two DDAs is described; merging more than two DDAs is done analogously.

When DDAs have to be merged, the DDA with the higher identifier, i.e., the younger one, which we will refer to as D_y, is merged into the older one, i.e., the one with the lower identifier. D_o will denote the older DDA. On receiving a request to merge into D_o, D_y sends all the information it has to D_o. This includes the dependencies D_y has received so far, the list of other DDAs that have previously been merged into it, and some other administrative information. D_y then records the identifier of D_o and becomes passive. This means that from now on D_y will forward all the information relevant for deadlock detection it receives to D_o. This way messages sent to D_y will still be processed. When D_o receives the merging message sent by D_y, it adds the received dependencies to its part of the WFG, thereby checking for cycles. It also adds the other information it got to the information it already has. Afterwards it sends a message to each of the transactions it became responsible for through the merger, so they become associated with it. Additionally a message is sent to each DDA that has previously been merged into D_y, informing them that from now on they should forward to D_o instead of to D_y. Note that this is done only for efficiency reasons, to prevent building forward chains between DDAs. It has no impact on the correctness of the algorithm, since D_y forwards the information it gets to D_o anyway.

An example is given in Figure 2. In this example D_5 is responsible for T_1, T_2, and T_5 while T_5 and T_6 are associated to D_6. It is obvious that D_5 and D_6 have to be merged, since the dependencies they know about belong to one connected component of the WFG. When D_6 receives a request to merge into D_5, in the first step it sends the information it has to D_5. In the second step D_5 informs T_5 and T_6 that their DDA has changed, and finally, in the third step, D_5 requests D_7 and D_8 to change their forwards.

Note that only transactions are informed about the merging of their DDAs. This implies that only they know their current DDA, or at least will get to know it within finite time, while objects may still refer to a DDA already merged into a new one. Due to the forwarding of messages this will not result in loss of information.

Initiation of a Merge

The merge of two or more DDAs can be initiated by an object or by a transaction. It is possible that different transactions involved in a conflict on one object have different DDAs. This means, that the conflict has caused the join of two or more connected components of the WFG into one, and therefore the corresponding DDAs have to be merged. The object chooses one DDA to report the dependencies to, and initiates the merging of the other DDAs involved, into the chosen one. The merge is initiated by sending a list of DDAs to be merged along with the dependencies. The chosen DDA sends a *merge_request* message to each of these DDAs.

The object chooses the DDA with the lowest identifier, i.e., the oldest one. The oldest one is chosen in order to avoid an infinite number of merges. In case the requesting transaction has a DDA different from the oldest one, the dependencies are sent indirectly, via the DDA of the requesting transaction. The message then also contains a *merge_request*, i.e., the information that the DDA has to merge into the older DDA. Since each transaction can wait for only one object, by sending the dependencies to the DDA of the requesting transaction, the algorithm guarantees, that at each point of time, only one DDA has outgoing edges from the node in the WFG corresponding to one transaction. As will be shown in Section 5.6, this is sufficient in order to avoid detection of phantom deadlocks.

The merging of DDAs can also be initiated by a transaction. An example is shown in Figure 3. If transaction T_1 is associated with D_{10}, but has accessed object O_1 before it became associated to this DDA, the object does not have this information. When a conflict on O_1 involving T_1 arises, the object can decide to send the dependencies to D_5, since it has the information that T_2 is associated to this DDA. Dependencies $T_3 \rightarrow T_2$ and $T_3 \rightarrow T_1$ will be sent to D_5. Upon receiving the dependencies, D_5 will send a message to T_1 and T_3, informing them that they are associated with it. When T_1 receives this message, it will notice that D_5 and D_{10} have dependencies it is involved in. T_1 will then initiate the merging of D_{10} into D_5. The message flow induced by this example is shown in Figure 4.

A transaction does not become associated with the new DDA immediately after it has initiated the merging of its previous DDA into the new one. This happens only after the new DDA confirms the merge, by sending a *set_merge_dda* message to the transaction. Until then the transaction stays associated with the old DDA. If the transaction immediately changed its DDA the following could happen: Its new DDA receives the information that the transaction waits for some other transactions, before the previous DDA has merged into it. This could result into two active DDAs having outgoing edges from one transaction (node) in their WFGs—which could possibly lead to the detection of phantom deadlocks.

5.3 Deadlock Resolution

A DDA starts a deadlock detection every time it receives new dependencies, i.e., it performs continuous detection. Dependencies an object sends to a DDA consist of one transaction waiting for a set of other transactions. Therefore the waiting

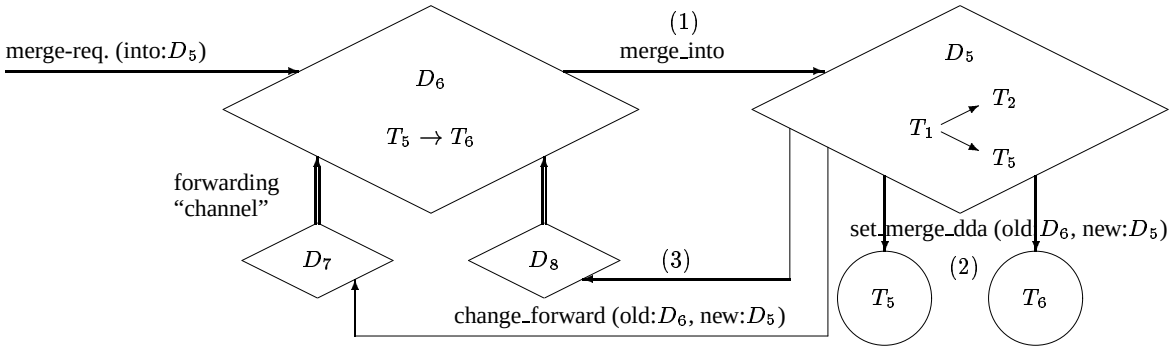

Fig. 2 Merging of two DDAs

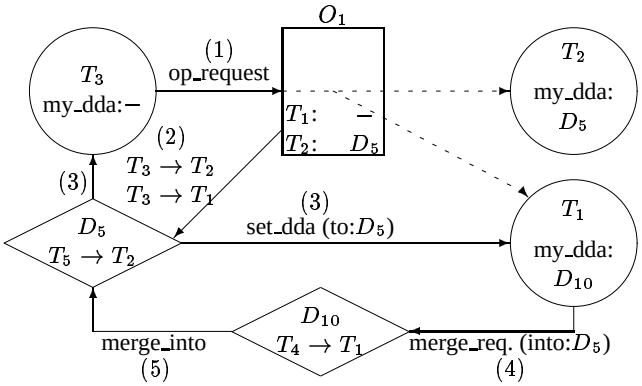

Fig. 3 Transaction initiates merging

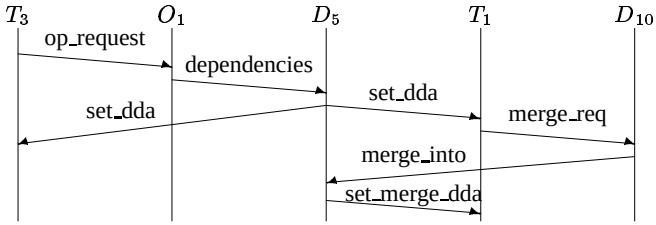

Fig. 4 Messages sent on a merge initiation by a transaction

transaction is a member of all cycles currently existing in the part of the WFG the DDA is responsible for. Hence the deadlock detection involves only a depth-first search starting from the waiting transaction. If a cycle is found, the DDA resolves it. Currently this is done by choosing the youngest transaction from the cycle to be aborted, if there is only one cycle. If more than one cycle have arisen through the new dependencies, the transaction whose wait dependencies newly arrived is aborted, and therefore all cycles get resolved. Since the DDA has complete information about the cycles in its part of the graph, any other deadlock resolution strategy could easily be integrated into the algorithm. Note that this is a clear advantage over other deadlock detection algorithms.

Besides from objects, a DDA can learn about new dependencies from another DDA merging into it. Currently the algorithm runs a depth-first search starting from each new node, i.e., treating each new node with its outgoing edges as new dependencies. This could also easily be replaced by a better strategy.

As will be explained in Section 5.5, for efficiency reasons, a DDA records the transactions it has aborted and those it knows have committed.

5.4 DDA Termination

The simplest possibility is that a DDA terminates after it has not received any message for a long enough period of time, i.e., a timeout. In the current system we have implemented this version, since DDAs that are not needed any more cause almost no overhead.

Another alternative is that a DDA terminates after all transactions in its part of the WFG have terminated. One possibility is that a DDA collects additional information about its part of the graph, i.e., about transactions which belong to this part of the graph but the DDA does not yet know of. The needed information can be provided by the transactions (as part of the message the transaction sends to its DDA upon its termination). This information is processed only by active DDAs, which then request obsolete DDAs to "dissolve".

5.5 Data Structures and Pseudo-Code for Deadlock Detection

For deadlock detection the object's structure has to be augmented by a list of transactions accessing it and the DDAs those transactions are associated with. Of course, each transaction needs to know its DDA (if it has one). As explained in the previous section, after a transaction initiates a merge of its current DDA into a new one, it waits for the confirmation of this merge before it becomes associated with the new DDA. In the meantime it stores it in *next*. Messages from DDAs to a transaction can arrive in a different order from the one in which they were sent. A transaction can receive a message that D_5 has merged into D_4 before it has learnt that D_5 is responsible for it. In order to avoid unnecessary forwards of messages, a transaction stores the *set_merge_dda* messages in *dda_merges*, until it can use the information they contain. In the following we will describe the structure of a DDA, shown in Figure 5.

A DDA is an object having the following special attributes: *ta_list*, *wfg*, *forw_addr*, *dda_list* and *term_tas*. The *ta_list* is

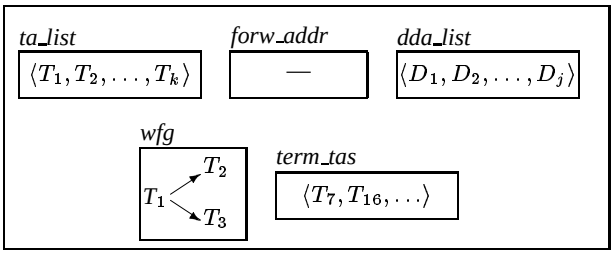

Fig. 5 DDA Structure

a list of all transactions the DDA is currently responsible for. The WFG information is maintained in the *wfg* structure. For a passive DDA, the *forw_addr* contains the address of the DDA the DDA has merged into. A DDA maintains the DDAs that have merged into it in *dda_list*. (These DDAs forward all their messages.) Dependencies involving a transaction may be received after it has terminated. To avoid adding this transaction again to the *wfg*, the DDA keeps track of the transactions that have terminated, i.e., it has aborted or it knows have committed, in *term_tas*. This information is needed only for efficiency reasons, i.e., to avoid unnecessary *set_dda* and *set_merge_dda* messages and to keep the WFGs smaller.[6] Note that this should also be done in other algorithms explicitly maintaining a WFG.

5.5.1 Code for Objects

The only change in the objects' code has to be made for the case a lock needed for an operation execution requested by a transaction cannot be granted. In this case the object has to determine a DDA to send the dependencies to.

– An operation request from transaction T_j is received

receive operation_request
  if lock can be granted
  then grant lock
  else (conflict with locks held by $T_1, \ldots, T_n$)
    determine DDA (create a new one if there
      is no DDA)
    send dependencies and DDAs to be merged
      into DDA (to determined DDA)
    update *TA/DDA Association List*
  fi

Determining a DDA is done as follows. If T_j is already associated with a DDA, this one is chosen. The object chooses the DDA with the lowest identifier from the set of DDAs the transactions $T_1, \ldots, T_n$ are associated with if T_j is not already associated with a DDA. It is possible, that transactions $T_1, \ldots, T_n$, and T_j are associated with different DDAs.[7] This causes a merge of these DDAs into the one with the lowest identifier. Note that the dependencies, along with the list of DDAs that have to be merged, are sent to the DDA of T_j (if it has one). If this DDA has to be merged into another DDA, the message also contains a merge request.

The object relies only on its own information about transactions being associated with DDAs when it determines a DDA. This information might be stale and lead to additional overhead (creating a DDA that is not needed), but it will not lead to incorrectness (loss of information) since the transactions will recognize the unnecessary creation and request the merging of the DDAs.

When the object has finished the execution of an operation it sends an acknowledgment to the requesting transaction. As described earlier this message also contains the DDA of the transaction, either the one the transaction has sent with its request or, if the transaction did not have a DDA but did cause dependencies, the one the object has sent these dependencies to. If the object has initiated a merge of DDAs, the transaction will learn about it from the DDA, not from the object.

5.5.2 Code for Transactions

Messages may not arrive in the order in which they were sent, so *set_merge_dda* and *set_dda* messages sent from different DDAs to a transaction may arrive in a "wrong" order. The transaction deals with this by storing information from received *set_merge_dda* messages in *dda_merges*.

– Transaction T_j requests an operation on object O_i

operation_request
  send operation request to O_i (with *my_dda*,
      if T_j has one)
  receive *operation_ack*([DDA_new])
  case
    the operation request contained a DDA:
      return
    operation_ack does not contain a DDA:
      return
    default:
      proceed as on **receive set_dda**
      update *my_dda*
  end

If the operation request contained a DDA the object sends the dependencies (if the transaction had to wait) to this DDA and returns its identifier to the transaction. This information is not new to the transaction, the transaction may even have the information that this DDA has already been merged into another DDA. Therefore it ignores it.

In case the transaction did not have a DDA when requesting the operation, it checks whether it has to initiate a merge (as in *receive set_dda*). This can be the case if it became associated with a different DDA while waiting for *operation_ack*. In order to guarantee, that only one DDA has outgoing edges from T_j in its part of the WFG, the transaction becomes associated with the DDA it got from the object, even if it has initiated the merging of this DDA. The transaction stays associated with this DDA until it receives a message that the merge has been accomplished. T_j may already have this information in *dda_merges* on receiving *operation_ack*.

[6] For the extension of the algorithm for the AND model at least aborted transactions have to be recorded in *term_tas* in order to prevent detection of phantom deadlocks.

[7] Transactions $T_1, \ldots, T_n$ may not conflict, e.g., if they all perform a read on the object, so they can have different DDAs.

– Transaction T_j receives a *set_dda*(DDA_new)

```
receive set_dda(DDA_new)
  case
    my_dda = – :       // not yet associated
                       // with any DDA
        my_dda := DDA_new
        next := DDA_new
    my_dda = DDA_new:  // I knew it already
        return
    next ≠ DDA_new:
        initiate merging, if needed
        update next
  end
```

The DDA (identifier) stored in *next* is the DDA the transaction is associated with or is to become associated with, but waits for the message that its previous DDA has merged into it. The transaction has to wait in order to guarantee that only one DDA has outgoing edges from the node T_j. To reduce unnecessary forwards, the transaction always uses *next*, not *my_dda*, for a merge.

In the last case transaction T_j checks (in *dda_merges*) whether DDA_new has merged into some other DDA (D_n). T_j then initiates merging of D_n (or DDA_new if D_n does not exist) and *next*. Of course, no merging is needed if D_n (or DDA_new in case there is no D_n) equals *next*.

– Transaction T_j receives a *set_merge_dda*(DDA_new, DDA_merged)

```
receive set_merge_dda(DDA_new, DDA_merged)
  insert received information in dda_merges
  if my_dda = – :      // not yet associated
                       // with any DDA
  then my_dda := DDA_new
    next := DDA_new
    return
  fi
  if next > DDA_new then next = DDA_new fi
  update my_dda considering dda_merges
```

DDA_merged is the DDA that has merged into DDA_new. *next* represents the oldest DDA T_j knows it will be associated with, so in case DDA_new is older than *next*, *next* is updated.

my_dda is updated to DDA_new or a DDA DDA_new has merged into.

5.5.3 Code for DDAs

When a DDA is merged into another DDA it becomes passive and forwards all messages it receives, except *forward_end* and *change_forward*. Here we consider only active DDAs.

– DDA D_j receives a message from an object containing a list of dependencies, a list of DDAs to be merged and (possibly) a *merge_request*

```
receive dependencies(dependencies, list of DDAs,
                     [merge_request])
  filter out dependencies involving transactions
        contained in term_tas
  add dependencies to wfg
  update ta_list
  send set_dda(self) to all new transactions
  if message does not contain merge_request
  then send merge_request(self) to all (new) DDAs
        in the received list,
    check for cycles in wfg
  else (message contains merge_request(DDA_merge))
    send merge_request(DDA_merge) to all
        (new) DDAs in the received list,
    merge into DDA_merge
  fi
```

The dependencies are always sent to the DDA of the blocked transaction. If the object knows about a DDA (with a lower identifier) of another transaction involved, it initiates a merge, by adding a *merge_request* to the message. The list of DDAs to be merged is also sent along with the dependencies. D_j requests the DDAs from the list to merge, either into itself or into DDA_merge. Of course, no message needs to be sent to DDAs already merged into D_j.

– DDA receives a *merge_request*(DDA_new)

```
receive merge_request(DDA_new)
  if own_id > DDA_new
  then merge_into DDA_new
  else if own_id < DDA_new
      then send merge_request to DDA_new
      fi
  fi
```

The case that the identifier of the DDA receiving the message is smaller than the identifier of DDA_new can arise due to the forwarding of messages. When this happens the receiving DDA sends a *merge_request* message to DDA_new.

– DDA receives a *merge_into* message

```
receive merge_into(wfg, ta_list, dda_list, term_tas)
  update ta_list and wfg according to received term_tas
  filter out received information involving transactions
        contained in term_tas
  send set_merge_dda to all transactions in
        the received ta_list
  update own ta_list
  send change_forward to all DDAs in
        the received dda_list
  append this list and the sender to own dda_list
  add received wfg to own wfg
  append received term_tas to own term_tas
  check for cycles in wfg
```

5.6 Correctness Issues

A deadlock detection algorithm is correct if all deadlocks that occur in the system are detected in finite time and all detected deadlocks really exist—at the time of their detection, see Section 3. In this section we will argue that the presented DDA algorithm is correct under the following assumptions.

We assume that message passing is error-free, i.e., that all messages that are sent arrive in finite time and are correct. Also the assumption is made that an existing dependency

$T_i \rightarrow T_j$ ceases to exist only if the transaction T_j commits or aborts. Under these assumptions the proof by Wuu and Bernstein [WB85] is applicable, saying that each cycle in the global WFG indicates a deadlock in the system. It also implies that each deadlock is permanent, i.e., a deadlock cannot be resolved without interference from outside. In case this is not true, every deadlock detection algorithm may detect phantom deadlocks, since it may detect a cycle that has already been spontaneously resolved. Of course, aborts on behalf of the deadlock detection algorithm are not considered as spontaneous aborts.

5.6.1 All Deadlocks are Detected

Observation 1 *All outgoing edges from one node are maintained in the* same *DDA.*

This Observation follows from the construction of the algorithm. Outgoing edges from one transaction arise *iff* a transaction waits for locks on one object. They are reported to a DDA and the waiting transaction learns about this DDA before it can make any further lock requests. This implies that after a transaction waits for the first time on some object, it knows which DDA is responsible for it and sends this DDA's identifier with every following lock request. Therefore, the objects on which the transaction waits at some later point of time will send the dependencies, that represent outgoing edges from the corresponding WFG node, to the same DDA.

If the (younger) DDA D_y one transaction is associated with merges into another (older) DDA, D_o, the transaction becomes associated with D_o, but only after its previous DDA D_y became inactive. So at each point of time only one of the DDAs contains outgoing edges from the corresponding node of the WFG.

After merging into D_o the previous DDA D_y forwards all dependencies it receives to D_o, so no information is lost.

Observation 2 *Let T_i be a node involved in a cycle*

$$T_1 \rightarrow \ldots \rightarrow T_j \rightarrow T_i \rightarrow \ldots \rightarrow T_1$$

of the global WFG. Then at some point of time the incoming edge to T_i involved in this cycle will be part of the WFG of the same DDA having the outgoing edges from T_i in its part of the graph.

From Observation 1 it follows that all outgoing edges from T_i are administrated by one DDA, say D_c. T_i has the information that it is associated with D_c, or a DDA merged into D_c. If T_i is involved in a cycle, then there is an edge $T_j \rightarrow T_i$ in the global WFG that is also part of this cycle. The dependency represented by this edge was recognized by an object and sent to a DDA. If it was sent to D_c nothing has to be proved. If it was sent to some other DDA, this DDA will inform T_i that it became its DDA. T_i will recognize that it has two DDAs and initiate their merging, so eventually a single DDA will have the edge $T_j \rightarrow T_i$ and all the outgoing edges from T_i in its part of the WFG.

Observation 3 *All deadlocks in the system will be detected and resolved in finite time.*

Let $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$ constitute a cycle of the global WFG. From Observation 2 follows that at some point of time there will be one DDA having all outgoing edges from T_1 and edge $T_n \rightarrow T_1$ in its part of the WFG. This is in particular true for edge $T_1 \rightarrow T_2$, so the DDA will also contain all outgoing edges from T_2 (Observation 2), in particular edge $T_2 \rightarrow T_3$. Continuing this will lead to the conclusion that at some point of time all edges involved in the cycle will be part of the WFG of one DDA. Note that DDAs also merge "in direction" of the DDA with the lower identifier—therefore, eventually this process must terminate.

Since a DDA starts a deadlock detection each time it receives new dependencies, the cycle will be detected and resolved when the edge completing the cycle reaches the DDA. As every deadlock is represented by a cycle in the graph, every deadlock will be detected and resolved.

5.6.2 No Phantom Deadlock Detection

Before showing that no phantom deadlocks are detected, we have to analyze what could lead to their detection. Under the assumptions we made, a deadlock cannot be resolved spontaneously and each cycle in the global WFG indicates a deadlock in the system. Therefore, the only possibilities that a phantom deadlock could be detected are that the same deadlock is detected by two different DDAs, or that by resolving one deadlock another deadlock also gets resolved, without the appropriate DDA noticing this.

One cycle can be detected at only one DDA since each dependency is reported to only one DDA and no other DDA has knowledge about it, as long as this DDA is active. If the DDA merges into another DDA it sends the dependencies to this DDA and again only one DDA has them.

For edge-chasing algorithms it is also true that each cycle is detected only once. A problem, however, arises when two cycles have at least one common node, as shown in Figure 6. If T_{20} (T_{10}) is the transaction with the largest identifier in the right-hand (left-hand) cycle, the following could happen. The probe sent on behalf of T_{20} passes T_{10} before T_{10} is chosen as a victim to resolve the left-hand cycle. Then, after resolving the left-hand cycle, T_{20} would be aborted to break the right-hand cycle which does not exist any more—due to aborting T_{10}. This inherent problem of edge-chasing algorithms was also reported in [RB88]. The above problem cannot happen in our DDA scheme.

When a DDA aborts a transaction to resolve a cycle it removes all edges the corresponding node is involved in. In particular it removes all outgoing edges from this node. Since only this DDA has outgoing edges from one node (Observation 1), if the transaction was involved in more than one cycle, all of these cycles will be removed from the graph, so no wrong deadlock detection is possible.

In order to reduce the number of aborts due to deadlocks even further, the DDA scheme employs the following heuristics. If the transaction "closing" a cycle is involved in more than one deadlock, this transaction is chosen as the victim.

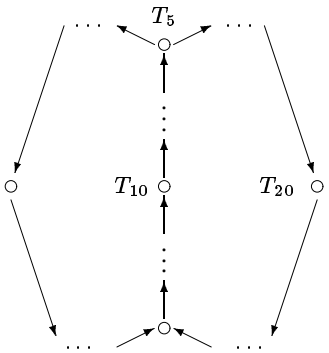

Fig. 6 Cycle Overlaps

Such a heuristics cannot be employed in an edge-chasing approach.

Thus, there are two reasons why the edge-chasing algorithms can be expected to induce more aborts. First, they inherently detect some phantom deadlocks. Secondly, they cannot deliberately choose transactions involved in several cycles. This observation is also backed by our simulation results in the next section.

5.7 Extension for the AND Model

The DDA approach can easily be extended to detect deadlocks in a model where one transaction can issue multiple requests in parallel.

The only reason why the presented algorithm is not sufficient for the AND model is that multiple requests of one transaction could lead to more than one DDA being responsible for it. Consider transaction T_1 having no DDA at the time it makes requests to O_1 and O_2. If at both objects dependencies arise, the objects might send them to different DDAs, or even create two DDAs for them. This could lead to detection of phantom deadlocks, since both DDAs would search for cycles T_1 is involved in. Note that this problem does not occur if T_1 already has a DDA when it makes the request, since the DDAs would properly merge before they start looking for cycles.

To avoid this problem, when used in the AND model the algorithm has to be extended as follows: Each transaction gets a *dummy* DDA assigned when it starts execution. Objects treat the dummy DDA in the same way as a "normal" one, the difference is, that a dummy DDA is instantiated only when it receives a message, i.e., a dependency involving the transaction arises. This way each transaction always has a DDA (and sends its identifier with every request it makes), so no phantom deadlocks are detected.

6 Comparative Performance Evaluation

Many distributed deadlock detection algorithms have been proposed, but only few quantitative comparisons of deadlock handling schemes in database systems exist. [ACM87] concentrated on a simulation study of various deadlock handling and deadlock avoidance algorithms in a centralized database system. [Cho90] compares the algorithm of [CKST89] with an algorithm that combines the path-pushing and the edge-chasing approach. [Buk92] presents a comparison of a centralized and a distributed approach. In [YHL94] two edge-chasing algorithms and the timeout approach were compared.

In this section we present the results of a much more comprehensive simulation study that compares distributed deadlock detection algorithms from different classes. The algorithms we have analyzed are the DDA scheme, as a representative of global state detection, the path-pushing algorithm by Obermarck [Obe82], the edge-chasing algorithm by Roesler et al. [RB89] and various timeout approaches. These algorithms cover all the different classes of algorithms, except the diffusing computation category. As already described in Section 4.5.2, the diffusing computation-based algorithms are not competitive for the restricted computational models underlying database transactions. Deadlocks occurring in the single resource and the AND model can be detected much more effectively with inherently less expensive algorithms, like the edge-chasing algorithms.

6.1 Simulation Model

In our simulation model a distributed database system consists of an arbitrary but, for one simulation run, fixed number of sites. Each site has one (time shared) CPU server, so at any point in time only one autonomously operating entity (e.g., object, transaction, deadlock detector) can be active at one site. A site is responsible for properly scheduling the entities located on it. They are scheduled on a first-come/first-serve basis. The system also models network latency: messages sent from one entity to another take different time, depending on the distance between their locations. Messages sent within a LAN take much less time than messages sent between distant sites. For simplicity, we omitted modeling I/O servers. However, we believe that this does not have an impact on the performance of deadlock detection algorithms because it will equally influence the throughput and response time achieved using different algorithms. To account for I/O delays, some "thinking time" is incorporated in the duration of operation execution.

Rather than having transactions enter the system at arbitrary points in time, we started a number of transactions at the beginning of a simulation run and the system then preserved the same number of active transactions (multiprogramming level, *mpl*) throughout the simulation. Instead of measuring a "cold start", we let the system run for a while and then, after the "warm-up phase", started to record the transaction processing. After completing the warm-up phase we recorded 10000 commits and analyzed the results.

The warm-up phase was intended to bring the system into a stable operational condition. It turned out that the different algorithms reach this stable condition after vastly different intervals. The real detection algorithms reach it rather soon, whereas the timeout approaches are very sensitive to

prolonged high-load runs. We report several experiments to demonstrate this effect.

When a transaction is initiated at a processor, it sends its first operation request to the object it wants to access. If the lock needed for this operation can be granted, the object acknowledges the operation by sending a message to the transaction. Upon receiving this message the transaction can send its next operation request. In case the lock cannot be granted, the object manager takes actions depending on the deadlock detection strategy used. If the edge-chasing algorithm is used, the object manager generates a probe; if the DDA approach is employed, dependencies are sent to a DDA, or a DDA is created. In case of the path-pushing algorithm and one of the timeout schemes the object manager sends the dependencies to the local deadlock detector.

Once a transaction has executed all of its operation requests, or was chosen as a victim of deadlock detection, it starts a commit or abort, by sending appropriate messages to the objects it has accessed. An aborted transaction has to be restarted after a restart delay. The delay varies for the different experiments, depending on the transaction mix, but is fix within one experiment. For a committed transaction a new one is started, at a random site, in order to maintain a constant multiprogramming level (*mpl*), i.e., number of active transactions. The transaction to be started is chosen according to the transaction mix specified for the experiment.

6.2 Implementation of the Algorithms

The implementations of the edge-chasing and the DDA approaches are straightforward, as described in previous sections, because no adaptation to the computational model is needed.

For the timeout approach transactions were changed in a way that a transaction sets a timeout each time it requests an operation execution on an object. If the operation execution is not acknowledged before the timeout expires the transaction aborts and is restarted after a restart delay. From now on we will refer to this algorithm as the *pure timeout* algorithm. The timeout approach augmented by a local deadlock detector at each site will be called the *timeout&detection* algorithm. As in the pure timeout approach, a transaction sets a timeout when it requests an operation execution and aborts if the acknowledgment does not arrive before the timeout expires. Additionally, an object sends dependencies to its local deadlock detector when dependencies occur. The detectors search for cycles in this local WFG and resolve deadlocks they detect, so the victim releases its locks faster.

The path-pushing algorithm had to be adapted to our computational model. In this algorithm each transaction starts execution on one site. If it wants to access data located at some other site, the transaction creates an *agent* at that site, if it does not already exist, which performs the request. Agents of one transaction are correlated through a unique transaction identifier. The algorithm assumes that when a transaction's agent is created at a site the local deadlock detector automatically knows about it. Also it assumes that the detector knows to which site an agent has sent a request, i.e., to which site it has moved its activity. In our implementation we did not really move the transaction but only informed the detectors about the transaction's current locus of control. That is, when a transaction moves its activity to another site it has to inform the remote deadlock detector about it, by sending a message to it. The transaction also has to send a message to its current local detector.

We investigated the cost of this adaptation by not "charging" the corresponding messages, i.e., by not counting them and letting them arrive immediately. This means that a request would cost as much as it costs in the other algorithms, which is not quite fair, since the algorithm needs additional information that cannot be obtained without any costs. However, except for the (modest) increase in the number of messages, this had only a minimal impact on the results.

This algorithm's overhead is dominated by its handling of the WFG—the costs for sending the messages informing deadlock detectors about the current locus of control turned out to be negligible. In a system in which some transactions access objects at more than one or two sites, the parts of the WFG that have to be sent grow tremendously; therefore, it takes a long time to construct the messages that have to be sent and to "unpack" them by the receiver. Also, since one edge might be sent several times as part of different paths, even with relatively small graphs the overhead becomes considerable. An important difference between this algorithm and the other ones explicitly maintaining the WFG is that the path-pushing scheme periodically exchanges parts of its WFG, while the other algorithms continuously update it.

6.3 Performance Experiments

We ran a number of experiments in order to study the performance of the algorithms under different loads. In the first scenario we generated a load of only fairly short transactions in a relatively homogeneous system representing one LAN. In this environment we analyzed the impact the timeout value has on the performance of the timeout algorithms. Also the behavior of different algorithms under the given load and the change in their behavior over time is studied. The performance of the algorithms was analyzed in two more scenarios. In the second scenario the same environment as in the first one was given, but a more typical load was introduced: a mix of short and a small number of very long transactions. The third scenario represents a WAN consisting of several LANs.

The following performance measures were obtained from the experiments: the number of messages, the restart ratio, the throughput and the response time. The restart ratio is defined as the number of aborts relative to the number of transactions executed. The throughput is measured in the number of committed transactions per millisecond, while the response time is the time transactions stay in the system, i.e., the time from the moment a transaction is started until its commit.

The simulation parameters used in the experiments are given in Figure 7. The number of objects used for the simulation runs is relatively small, but larger object bases lead to

No. of sites	100
No. of objects	10000
mpl	10,..,300 (400)
Operation execution	25 milliseconds
Undo of an operation	15 milliseconds
Commit (per operation)	3 milliseconds
Message delay local	3 milliseconds
Message delay within a LAN	10 milliseconds
Message delay remote, in WAN	200 milliseconds
Receiving (sending) a message	0.5 milliseconds
Cycle detection (DDA, timeout&detection)	1 milliseconds
Merging two DDAs	2 milliseconds

Fig. 7 Simulation Parameter Setting

	op_1	op_2	op_3	op_4
op_1	no	—	—	—
op_2	no	yes	—	—
op_3	no	no	yes	—
op_4	no	yes	yes	yes

Fig. 8 Compatibility Matrix

very few conflicts and even fewer deadlocks, so the behavior of the algorithms could not be investigated. Also, as pointed out in [ACM87], it is important to analyze the behavior of the algorithms under workloads with high conflict probabilities in order to evaluate their performance when "hot spots" exist in a database.

The synchronization was based on semantic locking. All objects were of the same type, having four operations; their compatibility matrix is shown in Figure 8.

The *mpl* used in the experiments varies depending on the given load. It ranges from 50 to 400 in the experiments where only short transactions were run; in other experiments, in which a small percentage of long transactions were added to the system, it varies between 10 and 300.

The transaction size, i.e., the number of accesses a transaction performs, is not given here, since we introduce different types of transactions to study realistic load profiles. The size ranges from the average size of 8 accesses, which is relatively short, to the size of 100, representing long transactions. Also the likelihood of an access being local varies for different transaction types. Therefore, the actual size of transactions and the probability of a request being local will be specified for each experiment separately.

The system-based parameters (message delays, cycle detection etc.) are based on measurements in our local area network. Other parameters are based on parameter settings of other simulation studies, [Buk92, Cho90], and our estimate of their average realistic values.

For the path-pushing algorithm the costs for handling the WFG was experimentally derived. Other schemes explicitly maintaining the WFG, i.e., the DDA and timeout&detection, continuously update the WFG and search for cycles in it. They insert new edges and only have to check whether these edges lead to a cycle, since the WFG was acyclic before the update. The path-pushing algorithm periodically updates the WFG and only periodically searches for cycles in it. Each time one site sends the paths of interest to another site, it has to search its whole WFG for potential cycles that have to be sent. The receiving site then has to exchange the old information for the newly arrived one. The time consumption for these operations strongly depends on the size of the graph, so we could not give an estimate for it, but had to calculate the actual costs anew each time. The costs for performing deadlock detection for this algorithm includes building the WFG, by incorporating local and received dependencies, and analyzing it. Analyzing the WFG means finding all cycles in it and identifying the paths that have to be sent to other deadlock detectors. In our simulation system these costs total in 1/8 millisecond per edge in the paths the detector has received. This cost measure was derived from performance experiments conducted on a Sun SPARCstation10/20, with a SuperSPARC processor.

The path-pushing algorithm is very sensitive towards the variation of the length of the interval after which it starts the periodic deadlock detection. We have tuned the length of this interval by running some simulations with different values for it. This was done for each of the different scenarios.

Generally, the results we report constitute averages of several simulation runs. The number of simulation runs per experiment depends on the variation of the results. The standard deviations of the recorded results are shown (as error-bars) in the graphs.

Average transaction size	8
Range of transaction size	4–12
Timeout values	1.5s, 3s, 5s, 10s
Interval for path-pushing	0.1s
Restart delay	1s
Transaction type 1	
Likelihood of a lock requested being local	100%
Share of type 1	50%
Transaction type 2	
Likelihood of a lock requested being local	60%
Share of type 2	50%

Fig. 9 Additional Simulation Parameters for Scenario 1

6.3.1 Scenario 1: Only Short Transactions in one LAN

In this scenario we compare the different algorithms when the system's load consists of short transactions only, as was done in other simulation studies [Cho90, Buk92, ACM87]. Though this is not a realistic scenario in a highly distributed system, since it represents only one part of the typical load, it helps understanding the algorithms.

Parameters used in this simulation are given in Figure 9. Most operation execution requests made by these transactions will be local: we introduced two types of transactions, one making only local requests, and the other making 60% local and 40% requests to randomly chosen objects in the system.

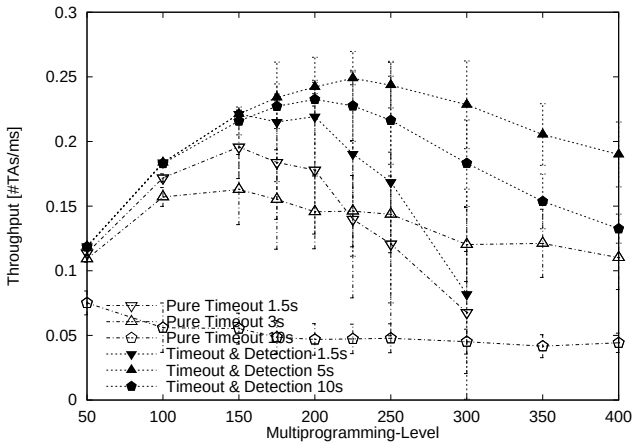

Fig. 10 Throughput for different timeout intervals

Adjusting the Timeout Interval

We have varied the values for the timeout interval in order to determine the optimal value for it. As described in [ACM87], the optimal timeout interval depends on the given workload and the *mpl*. Too short as well as too long timeout durations lead to performance degradation. A small value leads to too many restarts, a large timeout interval degrades the performance because deadlocks stay in the system for too long.

Figure 10 demonstrates the variations in the throughput induced by different intervals. The optimal value for the timeout&detection algorithm is about 5s. Decreasing the value to 1.5s significantly reduces the performance. For small *mpls*, when the system's load is low and deadlocks are rare, the algorithm performs well even for a short timeout interval, but with higher load it causes too many restarts. A timeout interval that is too long, e.g., 10s in this set-up, leads to a congestion of the system.

The pure timeout algorithm performs significantly worse than the timeout&detection algorithm—in this scenario—for all timeout intervals. For low *mpls* the optimal timeout interval is about 1.5s, but on high *mpls* the throughput drops below the one achieved with an interval of 3s. For *mpls* beyond 300 we could not obtain any results with the short interval. An interval length of 3s seems to be the best value overall, although for low *mpls* it is not optimal. A timeout interval of 10s performs poorly for any *mpl*, due to the congestion of the system it imposes.

The warm-up phase was 20000 commits; as the next experiment shows the timeout approaches achieve about their best throughput there.

Prolonged System Operation Under the Same (High) Load

This simulation experiment investigates the changes of the algorithms' performance over the time under a relatively high load. Figures 11 and 12 show the behavior of the algorithms over the time for *mpls* of 300 and 400, respectively—starting from a "cold" system. A point in the graph represents the throughput achieved during the last 10000 commits.

For both *mpls* the real detection algorithms become stable very fast[8], whereas the timeout approaches reach a stable state much later. The problem the pure timeout approach encounters, is that it does not have a resolution strategy but aborts all transactions "creating problems". The timeout&detection algorithm partly alleviates this problem by truly resolving at least local deadlocks. When a number of transactions try to execute operations that need conflicting locks, a good resolution strategy guarantees that at least one of them will finish, thus reducing the congestion. The timeout approaches accumulate these "problematic situations", so after some time almost no progress is possible. The pure timeout algorithm degrades very fast from the beginning. The timeout&detection algorithm maintains a surprisingly high throughput level for a long time, even at the *mpl* of 400, but then degenerates to a very low throughput. Note that the relatively high locality of the transactions is favorable for this algorithm.

The throughput of the real deadlock detection algorithms does not decrease over time, because they systematically resolve the occurring problems, not allowing them to stay in the system forever. The costs for this problem solving is the reason why at the beginning the timeout algorithms perform better; the price for their high throughput at the beginning is a very low throughput later on.

Variation of the Multiprogramming Level

The throughput results of the algorithms with a short warm-up phase (20000 Commits) and a long warm-up phase (70000 commits) are summarized in Figures 13 and 14, respectively. The measured results in Figure 13, for the *mpls* of 300 and 400, correspond to the left vertical lines in Figures 11 and 12, while the results shown in Figure 14 correspond to the right vertical lines. For the timeout approaches only the (previously measured) optimal timeout values are considered.

As could be expected from Figures 11 and 12 the real deadlock detection schemes achieve approximately the same throughput after the long as after the short warm-up phase. The timeout approaches have a much lower throughput after the longer warm-up phase for reasons explained above. The circled value in Figure 14 shows the result for the timeout&detection algorithm as far as we could obtain it. We had to stop some of the simulation runs since they ran for hours on our fastest machines (Sun's Ultras) without making any progress. For *mpls* beyond 300 we could not obtain any results. After a long warm-up phase the DDA performs much better than the other algorithms, as demonstrated in Figure 14.

The results the algorithms have achieved with a warm-up phase of 20000 will be discussed next, while additional results measured with the long warm-up phase will be omitted, since they would not convey any new information.

For low *mpls*, where conflicts are rare, the algorithms perform alike, except for the pure timeout approach which unnecessarily aborts too many transactions and thus performs

[8] The high standard deviation of the path-pushing algorithm will be explained later

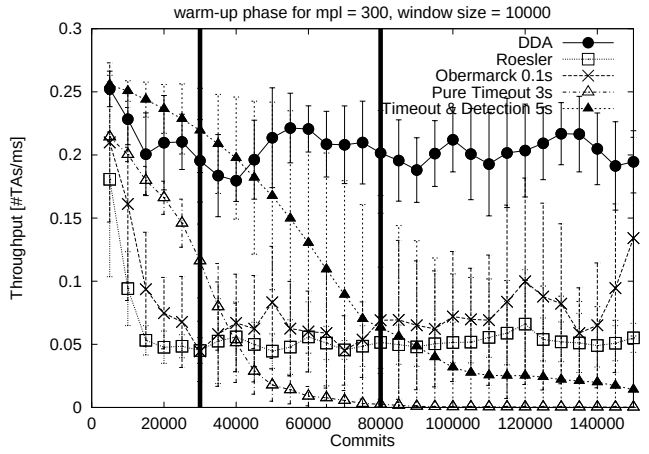

Fig. 11 Behavior over time, *mpl* 300

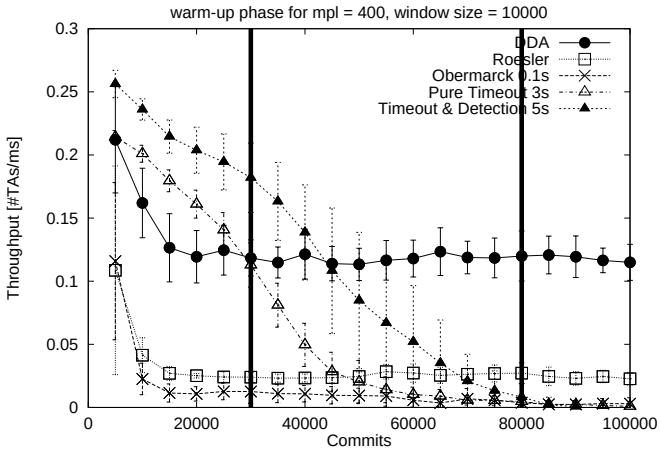

Fig. 12 Behavior over time, *mpl* 400

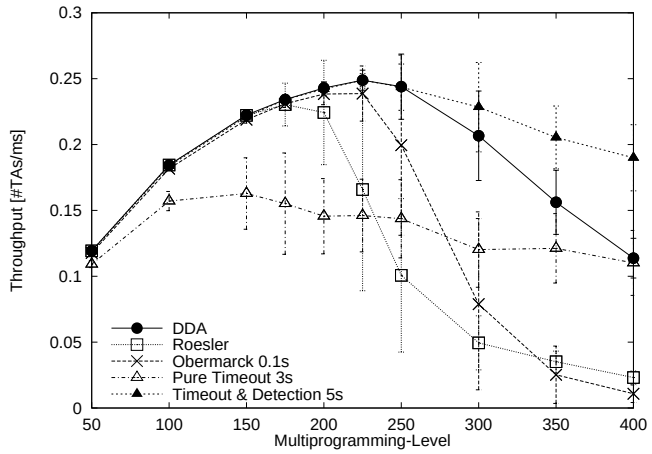

Fig. 13 Throughput, warm-up phase 20000

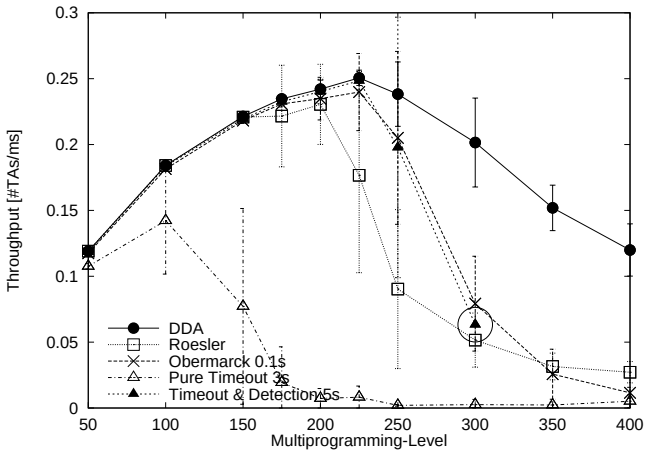

Fig. 14 Throughput, warm-up phase 70000

poorly. The given load is ideal for the timeout&detection algorithm. Most deadlocks here are local and can therefore be detected by the local deadlock detector, thus the algorithm resolves these deadlocks faster than other distributed algorithms. Therefore, for high *mpls*, a tuned timeout value and a short warm-up phase the timeout&detection algorithm performs better than other algorithms.

The DDA outperforms the other algorithms even after a short warm-up phase: the edge-chasing, the path-pushing and the pure timeout approach. The reason for the poor performance of the pure timeout approach was explained previously, the results achieved by the edge-chasing and the path-pushing algorithms are discussed below.

As pointed out in previous work [RB88], the performance of distributed deadlock detection algorithms strongly depends on the number of messages they induce. With the increase of the system's load the number of messages the edge-chasing algorithm generates grows drastically (see Figure 15) leading to a decrease in the throughput.

Obermarck's path-pushing algorithm appears to be more robust than the edge-chasing algorithm for lower *mpls*, but completely deteriorates for higher *mpls*. Some of the simulation runs beyond the *mpl* of 300 exceeded our computing facilities and some of them had no chance of ever getting done, i.e., to finish the required 10000 transactions, so we had to stop them. It turned out that the path-pushing algorithm is very sensitive to a "degeneration" of the WFG, meaning

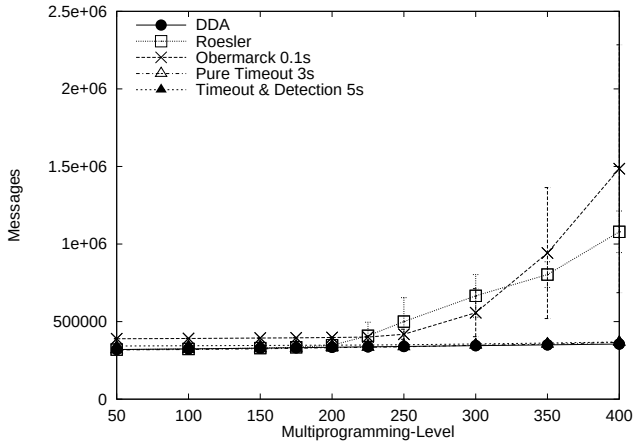

Fig. 15 Number of messages, warm-up phase 20000

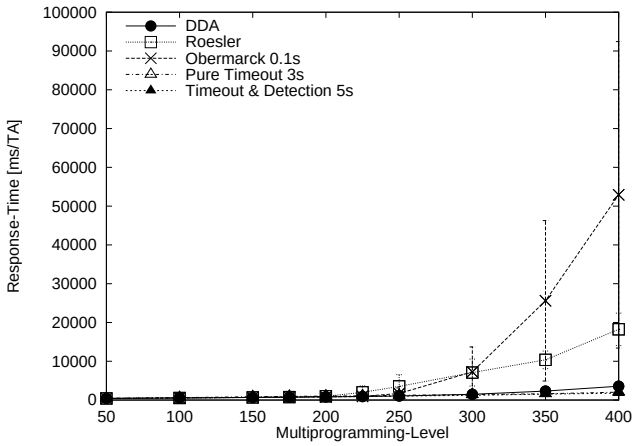

Fig. 16 Response time, warm-up phase 20000

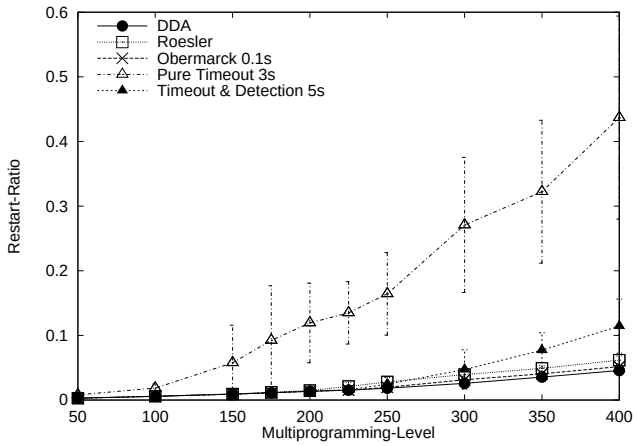

Fig. 17 Restart ratio, warm-up phase 20000

that the WFG becomes large with lots of cycles in it. Unfortunately, this situation occurred randomly in a number of simulation runs. This is also the reason for the high standard deviation of the simulation results of this algorithm.

The number of messages induced by the DDA approach does not significantly increase with a growing *mpl*. The reason is that even at high *mpls* each dependency is sent only once from an object to a DDA. Of course, there will be more merges when the *mpl* is higher, but only in the beginning before the centers of locality have built. Also, merging does not induce high message traffic.

Apart from the throughput and the number of messages sent during the simulation, we have also measured the response time of transactions. The huge number of messages the edge-chasing algorithm induces, and the time the path-pushing algorithm spends constructing the WFG and searching for cycles in it, increases the response time of transactions, see Figure 16. The response time achieved by the DDA algorithm is much shorter than for the path-pushing and the edge-chasing algorithms, but longer than for the timeout approaches. Note that only the response times of committed transactions are considered. This favors the timeout schemes, since transactions staying in the system for a long time that have not yet finished, i.e., some of the problematic transactions which are aborted again and again, are not included in these measurements.

The restart ratio of the different algorithms is as one would expect, see Figure 17. The DDA does not detect phantom deadlocks and therefore aborts less transactions than the other algorithms, although the differences in this experiment are not too significant.

6.3.2 Scenario 2: Mix of Transactions in one LAN

In this scenario we examine the behavior of algorithms in the same environment as in the first one, but with a more realistic transaction mix. Additional simulation parameters are given in Figure 18.

The load in the following experiment consists of three types of transactions: relatively short transactions accessing only local objects, transactions twice that length accessing local objects with a probability of 60% and remote ones with 40%, and a few long transactions accessing randomly chosen objects.

Timeout value	5s
Interval for path-pushing	0.1s
Restart delay	5s
Transaction type 1	
Average transaction size	8
Range of transaction size	4–12
Likelihood of a lock requested being local	100%
Share of type 1	30%
Transaction type 2	
Average transaction size	16
Range of transaction size	12–20
Likelihood of a lock requested being local	60%
Share of type 2	68%
Transaction type 3	
Transaction size	100
Access pattern	random
Share of type 3	2%

Fig. 18 Additional Simulation Parameters for Scenario 2

Throughput

This mix of transactions induces a heavier load on the system than it was the case in the first scenario, so the maximum throughput is lower and is obtained at a lower *mpl*. The reason is that longer transactions hold locks for a long period of time, and therefore all transactions waiting for these locks also stay longer in the system—even if they are short and only conflicts, but no deadlocks, occur.

Due to the higher load the system stabilizes much faster. Figure 19 demonstrates this for the *mpl* of 150. This leads to a small standard deviation in the measurements which is in most cases not even visible in the graphs. For Obermarck's path-pushing algorithm we could not obtain results for the

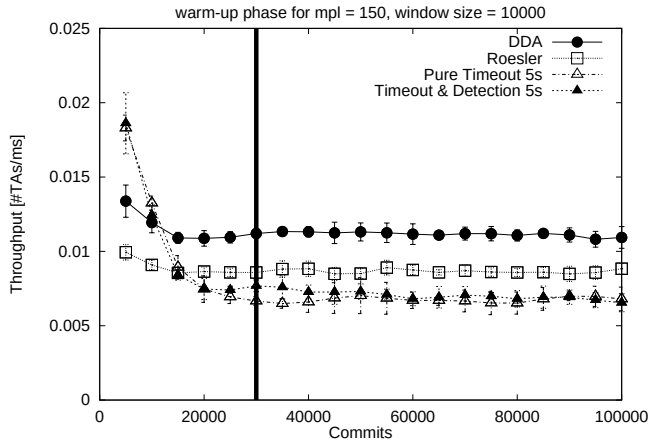

Fig. 19 Behavior over time, mpl 150

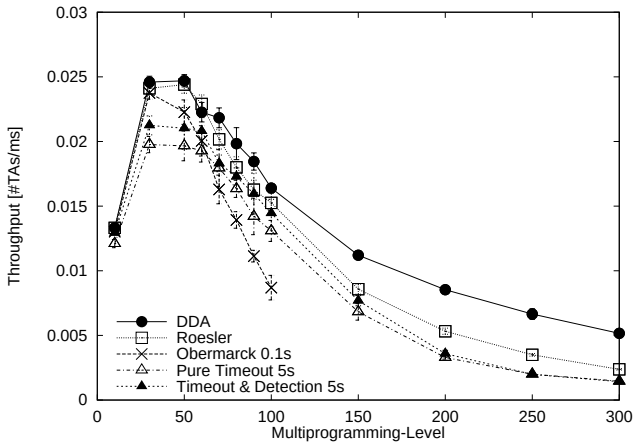

Fig. 20 Throughput, warm-up phase 20000

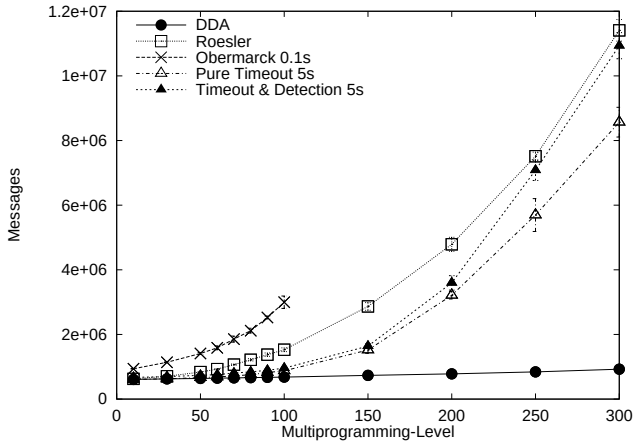

Fig. 21 Number of messages

mpl of 150, since the algorithm induced an enormous overhead, exceeding our computing facilities.

In this scenario the DDA scheme outperforms the other algorithms, see Figure 20. For lower *mpls* the edge-chasing algorithm performs almost as good as the DDA scheme, since the number of probes sent is not too large. With an increasing *mpl* the number of messages sent by the edge-chasing algorithm drastically grows—compared to the DDA scheme—and the throughput degrades. For the *mpl* of 150 the DDA achieves a throughput between 24% and 35% higher than the edge-chasing algorithm, cf. Figures 19 and 20. For the *mpl* of 250 the DDA achieves a 90%, and for the *mpl* of 300 even a 117%, higher throughput than Roesler's edge-chasing algorithm. The timeout approaches also perform poorly since they again accumulate "problems"—in this scenario even faster than in the first one. For the *mpl* of 150 the DDA performs between 46% and 67% better than the timeout&detection algorithm[9], cf. Figure 19. An *mpl* of 300 leads even to a 263% higher throughput by the DDA compared to the timeout&detection algorithm, cf. Figure 20. In this scenario the locality of transactions is not as high as in the previous one, so many deadlocks involve objects from two (sometimes even more) sites, leading to almost equal performance of the pure timeout and timeout&detection algorithms. The path-pushing algorithm by Obermarck achieves a low throughput even for low *mpls*. As explained above, the algorithm is sensitive to large WFGs so for *mpls* beyond 100 we could not obtain any results.

The reason for the poor performance of the edge-chasing algorithm is that it induces a high overhead by sending a large number of probes, see Figure 21. The timeout approaches also send many messages due to the enormous number of aborted transactions, as will be explained later. The timeout&detection algorithm sends even more messages than the pure timeout algorithm since additional deadlock detection messages are sent.

[9] Considering only results after the warm-up phase, i.e., 20000 commits for this experiment.

Both timeout approaches favor short transactions above the long ones, as can be seen in Figures 22–24. Figure 22 shows the average response time per operation over all transaction types. Here the response time is given in milliseconds per operation because of the wide range of the transactions' lengths. The DDA achieves the best response time. The edge-chasing algorithm has a poor response time, again because of the large number of probes it induces. The response times of the timeout approaches are long because they disadvantage long transactions. This becomes evident when the response times for transactions of different types are examined separately. For transactions of type 1 and 2, i.e., relatively short transactions, the timeout algorithms achieve much better response time than the edge-chasing algorithm and even than the DDA scheme. Figure 23 shows the response time of transactions of type 1 and 2. However, for type 3, i.e., long transactions, the timeout algorithms achieve a poor response time, see Figure 24, since these transactions are often aborted. This is not true for the DDA and the edge-chasing algorithm, since they favor old, not short, transactions in order to guarantee forward progress of all transaction types. Note that here only committed transactions are considered, so transactions that have been in the system for a long time, but have still not

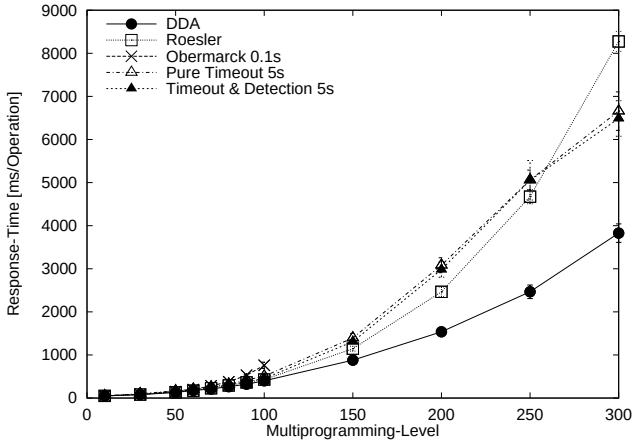

Fig. 22 Response time, all transactions

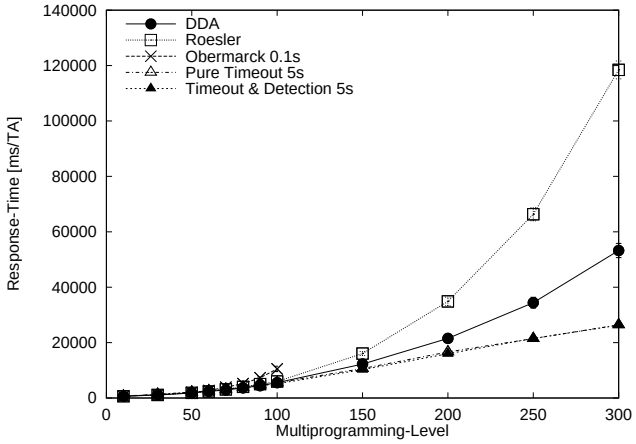

Fig. 23 Response time, short transactions

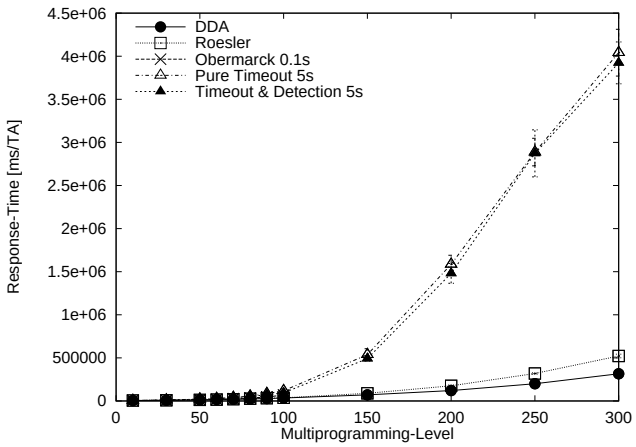

Fig. 24 Response time, long transactions

committed are not included in the results, which favors the timeout approaches.

The restart ratio of the algorithms is shown in Figure 25. The timeout approaches abort a transaction on the average fifteen times, on the *mpl* of 300. Aborting a transaction so many times implies that much more messages have to be sent, explaining Figure 21.

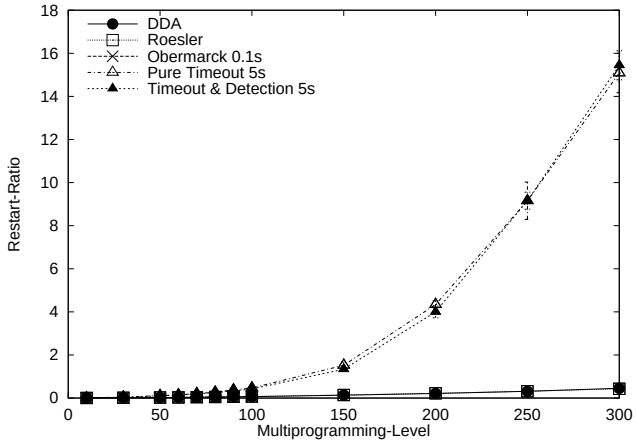

Fig. 25 Restart ratio

Number of LANs	5
Number of sites per LAN	20
Timeout value	5s, 7s
Restart delay	5s
Network disturbances	Every 10s for 1s–5s between two random LANs
Transaction type 1, as in load 2	
Share of type 1	35%
Transaction type 2, as in load 2	
Share of type 2	13%
Transaction type 3, as in load 2	
Share of type 3	2%
Transaction type 4	
Average transaction size	8
Range of transaction size	4–12
Likelihood of a lock requested being local	60%
Likelihood of a lock requested being in same LAN	40%
Share of type 4	50%

Fig. 26 Additional Simulation Parameters for Scenario 3

6.3.3 Scenario 3: Mix of Transactions in a WAN

In the last experiment we analyze the algorithms in a WAN environment, consisting of several LANs. We have utilized the transaction types[10] from scenario 2 and added one new type, which accesses only local objects and objects within the LAN it is started in. The additional simulation parameters are given in Figure 26. In order to simulate the irregularities of message delivery in a WAN, we introduced network disturbances. Every 10 seconds the connection between two randomly chosen LANs was disturbed for a random time duration between 1 second and 5 seconds (in one direction only). This means that sending of messages along this connection

[10] However, non-local accesses of these transactions constitute WAN-wide accesses.

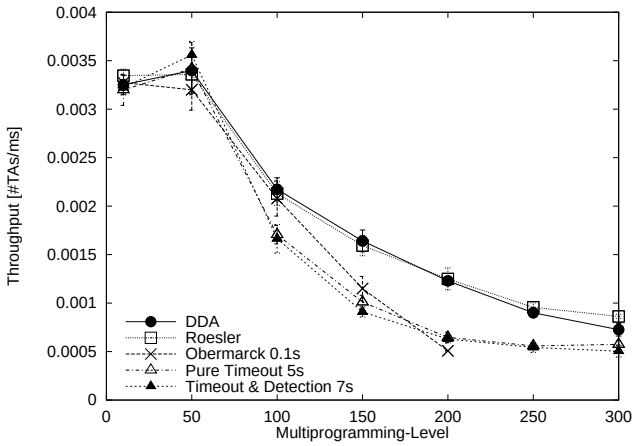

Fig. 27 Throughput

would be delayed, meaning that the arrival of messages would be postponed up to the length of the disturbance.

Throughput

The throughput of the algorithms measured in this experiment is presented in Figure 27. It shows the superiority of the real deadlock detection algorithms over the timeout approaches, although they have been tuned for this experiment. Again, on low *mpls* the algorithms perform alike, but as the *mpl* increases the throughput of the timeout approaches significantly drops. For the *mpl* of 200 the DDA and the edge-chasing algorithm achieve a throughput about 95% higher than the timeout approaches. The path-pushing algorithm by Obermarck again performs poorly on higher *mpls* because of the large WFGs it constructs, so we could not obtain any values for *mpls* beyond 200.

In this experiment the edge-chasing algorithm and the DDA perform alike. This is surprising since the DDA scheme outperformed it in the previous experiments. The reason is that the long message delays will, in some rare cases, postpone the detection of deadlocks by a DDA. Also, in the given environment, the DDA may sometimes take longer to detect local deadlocks—in case two DDAs mainly responsible for deadlock detection in different centers of locality merge. When the reason for the merge disappears, i.e., the centers of locality split up again, it takes some time before new dedicated DDAs form again.

Also the number of messages, the restart ratio and the response time have been measured, but these graphs did not reveal any new information, so we will not present them here.

6.4 Summary of the Experiments

In this section we reported on a thorough simulation study of the behavior of deadlock detection algorithms for distributed DBMSs in different environments. We ran quite a number of simulation experiments for many different database configurations—only a fraction of which could be presented here. However, they all showed the same relative trends that were observed on those experiments we discussed here.

Our study showed the importance and the difficulty of determining a good timeout interval for the timeout approaches, since it varies depending on the given load and the *mpl*.

The behavior of the algorithms over the time was analyzed, showing the stability of the DDA and the edge-chasing scheme, and the lacking robustness of the timeout approaches over prolonged heavy system loads, even when only relatively short transactions are involved.

On loads including mainly short but also a few long transactions, as introduced with the second scenario, all algorithms stabilized very fast, on different throughput levels. The DDA outperformed the other algorithms in this scenario by achieving a significantly higher throughput. The timeout algorithms performed much worse from the very beginning, while the edge-chasing algorithm performed well on low *mpls*, but deteriorated on higher *mpls*. The path-pushing algorithm performed better than the edge-chasing algorithm on lower *mpls* in the first scenario, but showed very poor performance in the other scenarios. In some of them no simulation results could be obtained for this algorithm.

In the last experiment, simulating a WAN environment, the timeout approaches performed poorly on higher *mpls*. The DDA and the edge-chasing algorithm achieved a much higher throughput.

The simulations show that overall the DDA outperforms the other true deadlock detection algorithms. It performs better or as good as other algorithms in all the different scenarios. Additionally, the DDA is the only algorithm of the analyzed ones that showed robustness towards different loads. For each of the other algorithms there was at least one experiment where we could either not obtain results for all *mpls*, or the throughput degraded almost to zero. The robustness of an algorithm is of great importance, especially in a distributed system where no global load control is feasible.

The reason the DDA outperforms the other algorithms is because it is robust against very high loads, i.e., it does not “explode” when the load increases. It establishes DDAs for the centers of locality which resolve deadlocks in these centers. Each dependency is sent only once from an object to a DDA, so the number of messages does not increase significantly and deadlocks can be detected fast since the information has short “travel time”.

7 Conclusion

In this paper deadlocks in distributed DBMSs have been analyzed. We identified the deadlock models that are represented in DBMSs and gave a detailed survey of existing algorithms for these models.

We have devised a new distributed deadlock detection algorithm, based on dynamically created deadlock detection agents (DDAs). A DDA is responsible for one connected component of the WFG. When a new component emerges a new DDA is created. In case two previously unconnected components interconnect the corresponding DDAs will merge.

When a component dissolves the DDA terminates after some time. This way the DDA scheme adapts very well to shifting hot spots and varying loads.

The paper also reports on a very thorough simulation study of deadlock detection algorithms which shows that the DDA scheme outperforms the other algorithms and is robust towards all the different loads. Therefore, we have implemented DDAs in our persistent, distributed system of autonomously operating objects, called AutO [KGI+97].

The computational model of AutO basically corresponds to the one described in Section 2. Objects are extended by an autonomous behavior, which is modeled through asynchronous message passing. Object managers are integrated into the objects, i.e., objects autonomously manage their resources, e.g., locks. Transactions, i.e., transaction managers, are also realized as autonomous objects.

AutO operates on many different sites connected through the Internet. It is fully implemented in Java, so it can run on a wide variety of platforms. Such a system has to be robust and reliable with respect to different workloads and access profiles. Based on the results of our simulation study, we chose the DDA as the best deadlock detection algorithm for our purpose. Porting the DDA-code from the simulation system into Java we were able to very quickly implement the deadlock detection method—which has proven to be very robust in a "real" system, just as the simulation results have indicated.

8 Acknowledgments

We would like to thank Stefan Pröls for his great help on the implementation of the algorithms. We also thank the anonymous referees for their helpful suggestions.

References

[ACM87] R. Agrawal, M. J. Carey, and L. W. McVoy. The performance of alternative strategies for dealing with deadlocks in database management systems. *IEEE Trans. Software Eng.*, 13(12):1348–1363, December 1987.

[Bad86] D. Z. Badal. The distributed deadlock detection algorithm. *ACM Trans. Comp. Syst.*, 4(4):320–337, November 1986.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, USA, 1987.

[BHRS95] 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*, 31(2):112–125, December 1995.

[BN97] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann Publishers, San Mateo, CA, USA, 1997.

[BO81] C. Beeri and R. Obermarck. A resource class independent deadlock detection algorithm. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 166–178, Cannes, France, 1981.

[BT87] G. Bracha and S. Toueg. Distributed deadlock detection. *Distributed Computing*, 2:127–138, 1987.

[Buk92] O. Bukhres. Performance comparison of distributed deadlock detection algorithms. In *Proc. IEEE Conf. on Data Engineering*, pages 210–217, Tempe, AR, February 1992.

[CDAS96] S. Chen, Y. Deng, P. Attie, and W. Sun. Optimal deadlock detection in distributed systems based on locally constructed wait-for graphs. In *Proc. of the 16th Intl. Conf. on Distributed Computing System*, pages 613–619, 1996.

[Cha82] E. J. H. Chang. Echo algorithms: Depth parallel operations on general graphs. *IEEE Trans. Software Eng.*, 8(4):391–401, 82.

[Cho90] A. N. Choudhary. Cost of distributed deadlock detection: A performance study. In *Proc. IEEE Conf. on Data Engineering*, pages 174–181, L.A., CA, February 1990.

[CKST89] A. N. Choudhary, W. H. Kohler, J. A. Stankovic, and D. Towsley. A modified priority based probe algorithm for distributed deadlock detection and resolution. *IEEE Trans. Software Eng.*, 15(1):10–17, January 1989.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comp. Syst.*, 3(1):63–75, February 1985.

[CM82] K. M. Chandy and J. Misra. A distributed algorithm for detecting resource deadlocks in distributed systems. In *Proc. of the ACM Symposium on Principles of Distributed Computing*, pages 157–164, 1982.

[CMH83] K. M. Chandy, J. Misra, and L. M. Haas. Distributed deadlock detection. *ACM Trans. Comp. Syst.*, 1(2):141–156, May 1983.

[DS80] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1), August 1980.

[Elm86] A. K. Elmagarmid. A survey of distributed deadlock detection algorithms. *ACM SIGMOD Record*, 15(3):37–45, 1986.

[ESL88] A. K. Elmagarmid, N. Soundararajan, and M. T. Liu. A distributed deadlock detection and resolution algorithm and its correctness proof. *IEEE Trans. Software Eng.*, 14(10):1443–1452, October 1988.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, CA, USA, 1993.

[GS80] V. D. Gligor and S. H. Shattuck. On deadlock detection in distributed systems. *IEEE Trans. Software Eng.*, 6(5):435–440, September 1980.

[Hof94] M. Hofri. On timeout for global deadlock detection in decentralized database systems. *PROCESS-LETTERS*, 51(6):295–302, September 1994.

[KGI+97] N. Krivokapić, S. Grießer, M. Islinger, M. Keidl, S. Pröls, S. Seltzsam, and A. Kemper. AutO—A distributed system of autonomous objects. `http://www.db.fmi.uni-passau.de/projects/auto/auto.html`, 1997.

[Kna87] E. Knapp. Deadlock detection in distributed databases. *ACM Computing Surveys*, 19(4):303–328, December 1987.

[Kor83] H. F. Korth. Locking primitives in a database system. *Journal of the ACM*, 30(1):55–79, January 1983.

[KS91] A. D. Kshemkalyani and M. Singhal. Invariant-based verification of a distributed deadlock detection algorithm. *IEEE Trans. Software Eng.*, 17(8):789–799, August 1991.

[KS94a] A. D. Kshemkalyani and M. Singhal. Efficient detection and resolution of generalized distributed deadlocks. *IEEE Trans. Software Eng.*, 20(1):43–54, January 1994.

[KS94b] A. D. Kshemkalyani and M. Singhal. On characterization and correctness of distributed deadlock detection. *Journal of Parallel and Distributed Computing*, 22:44–59, July 1994.

[KS97] A. D. Kshemkalyani and M. Singhal. Distributed detection of generalized deadlocks. In *Proc. of the 17th Intl. Conf. on Distributed Computing System*, pages 553–560, 1997.

[LK95] S. Lee and J. L. Kim. An efficient distributed deadlock detection algorithm. In *Proc. of the 15th Intl. Conf. on Distributed Computing System*, pages 169–178, 1995.

[LMB97] L. Leverenz, R. Mateosian, and S. Bobrowski. *Oracle8 Server – Concepts Manual*. Oracle Corporation, Redwood Shores, CA, USA, 1997.

[MC82] J. Misra and K. M. Chandy. Termination detection of diffusing computations in communicating sequential processes. *ACM Trans. Programming Languages and Systems*, 4(1):37–43, January 1982.

[MM79] D. A. Menasce and R. R. Muntz. Locking and deadlock detection in distributed data bases. *IEEE Trans. Software Eng.*, 5(3):195–202, May 1979.

[Obe82] R. Obermarck. Distributed deadlock detection algorithm. *ACM Trans. on Database Systems*, 7(2):187–208, June 1982.

[RB88] M. Roesler and W. A. Burkhard. Deadlock resolution and semantic lock models in object-oriented distributed systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 361–370, Chicago, IL, USA, May 1988.

[RB89] M. Roesler and W. A. Burkhard. Resolution of deadlocks in object-oriented distributed systems. *IEEE Trans. on Comp.*, 38(8):1212–1224, August 1989.

[RBC88] M. Roesler, W. A. Burkhard, and K. B. Cooper. Efficient deadlock resolution for lock-based concurrency control schemes. In *Proc. 9th Int. Conf. on Distributed Computing Systems*, pages 224–233, 1988.

[RHGL97] F. F. Rezende, T. Härder, A. Gloeckner, and J. Lutze. Detection arcs for deadlock management in nested transactoins and their performance. In *Proc. of the 15th British Nat. Conf. on Databases*, London, UK, July 1997.

[Ruk91] M. Rukoz. Hierarchical deadlock detection for nested transactions. *Distributed Computing*, 4:123–129, 1991.

[SAL+96] M. Stonebraker, P. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *The VLDB Journal*, 5(1):48–63, January 1996.

[SH89] B. A. Sanders and P. A. Heuberger. Distributed deadlock detection and resolution with probes. In *Proc. of the Third Intl. Workshop on Distributed Algorithms*, volume 392 of *Lecture Notes in Computer Science (LNCS)*, pages 208–218, New York, Berlin, etc., 1989. Springer-Verlag.

[Sin89] M. Singhal. Deadlock detection in distributed systems. *IEEE Computer*, 22(11):37–48, November 1989.

[SN85] M. K. Sinha and N. Natarajan. A priority based distributed deadlock detection algorithm. *IEEE Trans. Software Eng.*, 11(1):67–80, January 1985.

[SS84] P. M. Schwarz and A. Z. Spector. Synchronizing shared abstract types. *ACM Trans. Computer Systems*, 2(3):223–250, 1984.

[WB85] G. T. Wuu and A. J. Bernstein. False deadlock detection in distributed systems. *IEEE Trans. Software Eng.*, 11(8):820–821, 1985.

[WDH+81] R. Williams, D. Daniels, L. Haas, G. Lapis, B. Lindsay, P. Ng, R. Obermarck, P. Selinger, A. Walker, P. Wilms, and R. Yost. R*: An overview of the architecture. IBM Research, San Jose, CA, RJ3325, December 1981. Reprinted in: M. Stonebraker (ed.), Readings in Database Systems, Morgan Kaufmann Publishers, 1994, pp. 515–536.

[YHL94] C. Yeung, S. Hung, and K. Lam. Performance evaluation of a new distributed deadlock detection algorithm. *ACM SIGMOD Record*, 23(3):21–26, 1994.