Deadlock Detection Agents: A Distributed Deadlock Detection Scheme

*Natalija Krivokapić** *Alfons Kemper** *Ehud Gudes*[+]

*Universität Passau
Lehrstuhl für Informatik
94030 Passau, Germany
⟨lastname⟩@db.fmi.uni-passau.de

[+]Ben-Gurion University of the Negev
Department of Math. & Comp. Science
Beer-Sheva, 84105, Israel
email: ehud@indigo.bgu.ac.il

Abstract

We describe a new deadlock detection scheme that was devised for a distributed system of autonomously operating object managers. In this system two-phase locking-based synchronization of parallel transactions is done locally—e.g., by employing a semantic locking scheme based on the objects' interface operations. The deadlock detection is enabled by dynamically created *deadlock detection agents* (DDAs). Transactions start executing without any DDA. Only if a conflict with some other transaction occurs they are associated with a DDA: either a DDA one of the conflicting transactions is 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 one DDA. The DDA scheme is a "self-tuning" system: After an initial warm-up phase, dedicated DDAs will be formed for so-called centers of locality. A dynamic shift in locality of the distributed system will be responded to by automatically creating new DDAs while the obsolete ones terminate. A simulation study indicates the superiority of the DDA scheme over a so-called edge-chasing distributed deadlock detection approach.

1 Introduction

While commercial distributed database 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. Among others, Mariposa [SAL+96] and SHORE [CDF+94] are two recent prototype systems addressing this demand. 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.

In this paper we develop a new deadlock detection algorithm designed for distributed systems of autonomously operating objects (or object managers). 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 deadlock detection is carried out by dynamically created *deadlock detection agents* (DDAs). Each DDA maintains one part of the wait-for-graph (WFG) and searches for cycles that—in the resource model necessarily—constitute deadlocks. 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 transactions that are already associated with different DDAs encounter a conflict, these DDAs are merged into one DDA. This scheme guarantees that all—but only "real"—deadlocks will be detected.

In a large-scale distributed database system one can expect that so-called 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 so-called centers 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 automatically terminate.

We built a simulation system in order to validate and evaluate the performance of the DDA scheme. We compared the DDA approach against a representative of the so-called "edge-chasing" algorithms [RB88]. Our simulation results show that the DDA scheme induces much fewer messages than the edge-chasing algorithm—even though the contender that we chose is the one that generates the fewest messages of the edge-chasing algorithms. Due to the drastic difference in generated messages the DDA scheme is far superior when the system load (and, therefore, the number of transaction dependencies) is increased while it imposes about the same overhead under low system loads when relatively few dependencies and even fewer deadlocks are encountered.

The remainder of this paper is organized as follows. In Section 2 we outline the underlying model of computation. In Section 3 related distributed deadlock detection algorithms are surveyed. The deadlock detection scheme is described in Section 4. In Section 5 the correctness of the algorithm is discussed. The performance comparison of the DDA scheme with a representative of the so-called edge-chasing deadlock detection methods is reported in Section 6. Finally, Section 7 concludes this paper and gives an outline of ongoing work.

2 The Model of Computation

For this presentation we will abstract from a "full-fledged" distributed model of autonomously operating object managers [KLMW94] or site managers. We consider three kinds of autonomously operating entities in the distributed system: *Transaction managers* (TMs) that control the execution and commit processing of transactions (TAs), *object managers* (OMs) that receive requests for executing operations upon their objects from the transaction managers, and *deadlock detection agents* (DDAs) that detect deadlocks based on their parts of the wait-for-graph. In order to simplify the presentation we assume that each transaction manager (object manager) controls only a single transaction (object). Therefore, we will often use the terms transaction (object) synonymously for transaction manager (object manager). This simplification, however, does not constitute a loss of generality; the scheme can easily be adapted to a more realistic architecture.

Transactions, objects and DDAs are assigned system-wide unique identifiers which are generated in increasing order. The execution model is based on *asynchronous message passing*. Transaction managers send requests to object managers, which, in turn, send acknowledgments (and/or results) back to the TMs. If an OM determines a conflict such that a pending request cannot be granted, the dependency is sent to a deadlock detection agent. DDAs send messages to TMs reporting that the TA was associated with this particular DDA or request the abort of a TA which was chosen as a deadlock victim. Furthermore, a DDA may receive a request—e.g., from a transaction—in order to initiate a merge into another DDA. The corresponding DDA becomes passive and forwards all information (including messages that arrive after the merge) to the other DDA.

A transaction consists of a sequence $O_1.op_{i_1}(\ldots);\ O_2.op_{i_2}(\ldots);\ O_3.op_{i_3}(\ldots);\ \ldots$ of operation invocations op_{i_j} 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.

The object managers receive the requests for operation invocations and schedule them in some order (e.g., FIFO). The object manager maintains the lock table and, in the case of semantic locking [SS84, Kor83], 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 requested operation can be executed. If not, the operation invocation is delayed and, thereby, the particular transaction's execution is blocked until the required lock(s) can be granted. If an operation is executed, the OM determines the lock mode and associates it with the corresponding TA. We assume a two-phase locking protocol: In the first phase TMs request operation executions on (possibly) different objects. During the first phase, 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.

If an operation invocation requested by transaction T_i has to be blocked, the OM determines the holders of conflicting locks. Note that there may be several transactions $T_{j_1}, \ldots, T_{j_n}$ that possess conflicting locks. The resulting dependencies (wait-for-relationships) $T_i \rightarrow T_{j_1}, \ldots, T_i \rightarrow T_{j_n}$ are sent to a particular deadlock detection agent–the details are explained in Section 4. In general, a transaction can wait for more than one transaction, and transactions waiting for locks on the same object can depend on different (sets of) transactions.[1]

[1] If we consider an object *bank* that consists of a set of *accounts*, transactions T_1, T_2 and T_3 could at

If some DDA detects a deadlock by finding a cycle in its part of the wait-for-graph, it chooses one of the TAs causing the deadlock as a victim. This victim TA is requested to abort. The corresponding TM will, after receiving the abort request from the DDA, initiate the undo action upon the already visited OMs. Thereafter, the resources (locks) held by the victim are released—thereby breaking up the deadlock cycle.

3 Related Work

Of course, quite a bit of work on distributed deadlock detection exists. An early survey of this work in the context of distributed database systems is given by Elmagarmid [Elm86]. However, many researchers feel that the performance and scalability requirements imposed by large-scale distributed systems require to revisit the implementation methods developed so far.

In a survey article Knapp [Kna87] classifies distributed deadlock detection algorithms in the following categories: (1) *centralized approach* [Gra78] by maintaining the global wait-for-graph, (2) *path-pushing algorithms* [Obe82] by sending parts of the WFG to neighboring sites, (3) *edge-chasing detection* by sending probes, (4) *diffusing computation* [DS80] by sending probes to all dependent processes (OMs) and waiting for the receipt of a response which, eventually, signals a deadlock at the root process (OM), and (5) *global state detection* where the relevant parts of the WFG graph are constructed in a globally consistent form without "freezing" the computation. The DDA scheme presented here can be classified under category (5).

It is interesting to observe that most path-pushing algorithms turned out to be incorrect in the sense that they detect so-called "phantom" deadlocks and may even fail to detect some genuine deadlocks. For example, Elmagarmid [Elm86] showed that the path-pushing approach of [Obe82] is defective.

Therefore, the more recent work on deadlock detection in distributed database systems concentrated on the edge-chasing paradigm. An early example of this approach is the resource deadlock algorithm of [CMH83]. This algorithm, however, generates an abundance of messages (probes); therefore, it was tried to be improved by a priority mechanism in [SN85]. Unfortunately, this improvement turned out to be incorrect; but it could be repaired by [CKST89]. [RB88]'s algorithm also used this edge-chasing idea as a basis for their algorithm. Their algorithm was developed in a context similar to ours and, furthermore, it appears to induce the fewest messages among the edge-chasing algorithms. Therefore, this algorithm was used by us as a "contender" to compare the performance of our DDA scheme. The results of a simulation study are reported in Section 6.

In the following we will briefly outline the working principles of this edge-chasing algorithm. The model of computation is similar to ours. For deadlock detection, the edge-chasing algorithm sends so-called probes between TMs and OMs. An example is illustrated in Figure 1. In this example, T_7 requested an operation on object O_1 which cannot be granted. Therefore, O_1 initiates a probe $\langle 7, 7\rangle$ on behalf of T_7 in order to start

---

one point of time hold the following locks $readBalance(acct_1)$, $readBalance(acct_2)$ and $readAllBalances()$, respectively. A transaction T_4 requesting a $debit(acct_1)$ would depend on T_1 and T_3, while a request $debit(acct_2)$ by T_5 would cause this transaction to wait for T_2 and T_3.

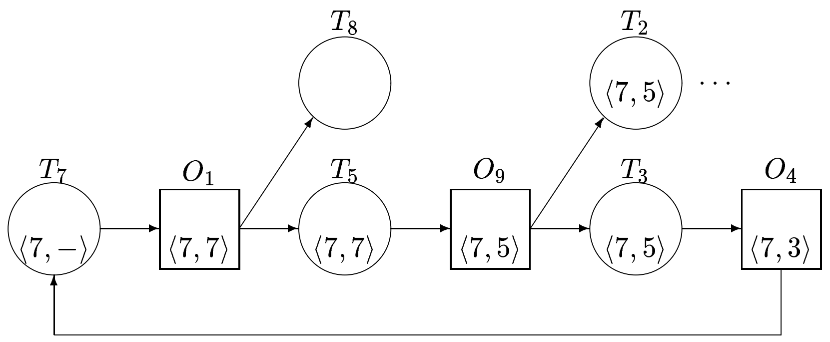

Figure 1: Illustration of the Edge-Chasing Algorithm

the deadlock detection. O_1 could forward this probe to all TAs, for which T_7 is waiting to release their locks on O_1. However, in order to reduce message traffic the OM only forwards probes to those TAs that have an identifier lower than the initiator (here T_7) of the probe. These probes that are being sent only to TAs with a lower identifier are called *antagonistic* probes. Therefore, T_8 is not sent the probe; only T_5 receives it. If T_5 waits for some OM to respond it forwards the probe to this particular OM—here the OM of object O_9. This OM, in turn, sends the probe to TAs T_2 and T_3. Let us follow the "journey" of the probe via T_3. Eventually the probe returns to the initiator T_7 which, upon receiving this probe, knows that the probe must have traversed a cycle which constitutes a deadlock.

To reduce the number of probe initiations the TAs as well as the OMs store the probes that they receive in order to forward them when a dependency occurs at some later time. However, the stored probes have to be removed upon detecting a cycle (deadlock). The initiator has to send a so-called *anti-probe* upon detecting the deadlock. Only after sending the anti-probe, the TA is aborted to break the deadlock cycle. The details of the algorithm and an analytical performance estimate can be found in [RB88].

There also exists some work on comparisons of deadlock handling schemes in databases: [Cho90] compares the algorithm of [CKST89] with a path-pushing algorithm; [Buk92] presents a comparison of two distributed deadlock detection algorithms. [ACM87] concentrated on a simulation study of various deadlock handling and deadlock avoidance algorithms in a central database system.

4 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 is a DDA responsible for detecting cycles in it. Each node (transaction) in the WFG belongs to only one connected component, so each transaction is associated with only 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. 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 emerging 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.

In a large-scale distributed database system it is likely that different types of transactions access different groups of objects, thereby creating so-called 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.

Our computational model assumes single-threaded transactions, hence each transaction can have only one outstanding request at a time. However, a transaction may still wait for a number of transactions to release their locks, i.e., a transaction can have several outgoing edges in the WFG.

For this algorithm the assumption is made that the transmission of messages is error-free, which means that all messages that are sent arrive in finite time and are correct.

4.1 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. 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 2.

T_2 and T_5 are holding non-conflicting locks on O_1, and none of the transactions is associated to 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 $T_1 \rightarrow T_2$ and $T_1 \rightarrow T_5$ arise. O_1 creates DDA D_5 and passes the dependencies to it. The object

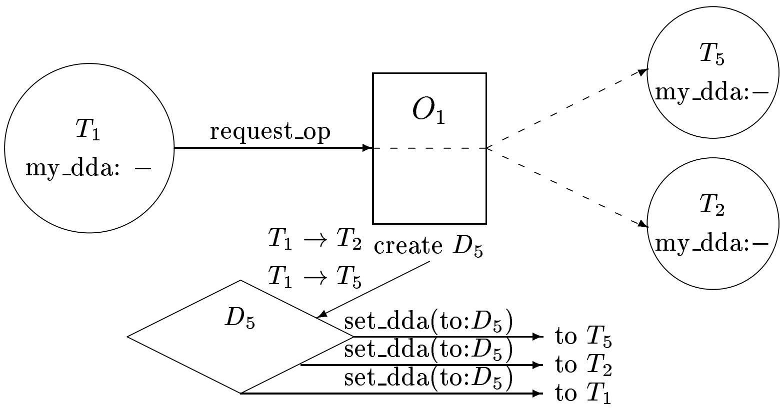

Figure 2: Creation of a DDA

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 every 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, no DDA has to be created. If there is only 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.

4.2 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,

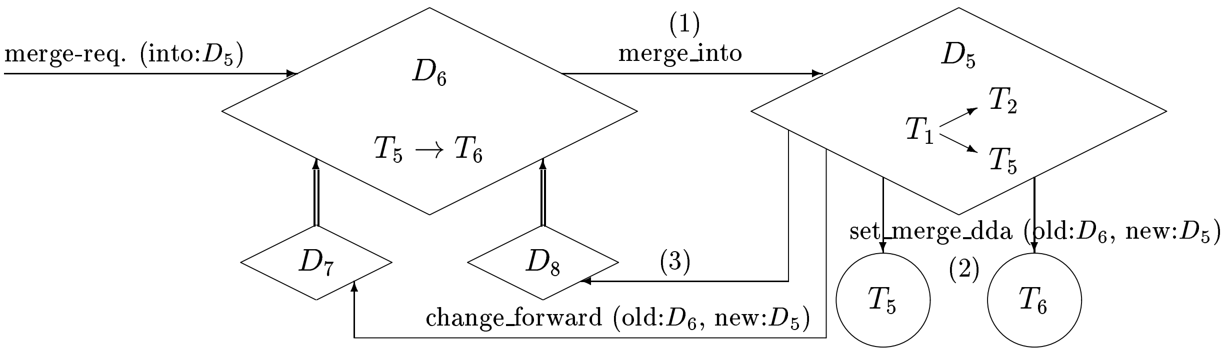

Figure 3: Merging of two DDAs

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 3.

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. 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, this is sufficient in order to avoid detection of phantom

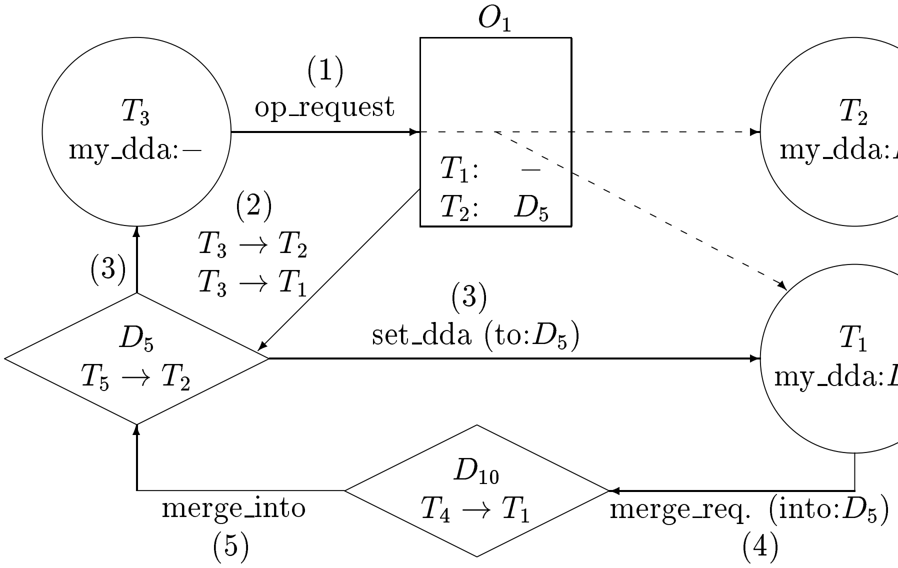

Figure 4: Transaction initiates merging

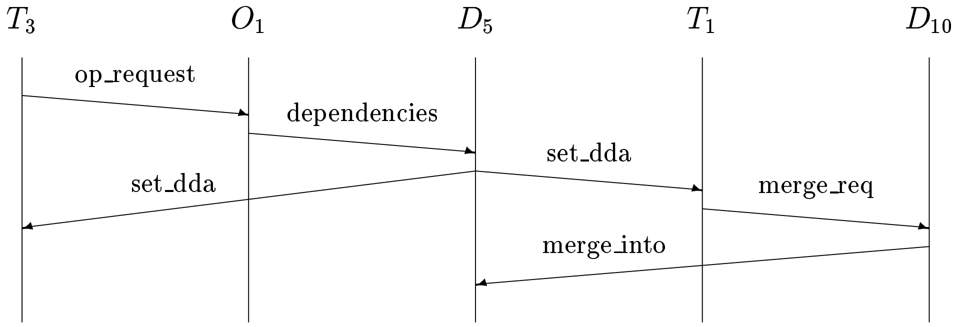

Figure 5: Messages sent on a merge initiation by a transaction

deadlocks.

The merging of DDAs can also be initiated by a transaction. An example is shown in Figure 4. 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 $T_3 \to T_2$ and $T_3 \to T_1$ 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 5.

A transaction does not become associated to 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.

4.3 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 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 newly arrived transaction is aborted, and therefore all cycles 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.

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.

In order to avoid detection of phantom deadlocks, a DDA has to record the transactions it has aborted. This is necessary because due to message delays dependencies a transaction is involved in can reach the DDA after the transaction has already been aborted. They must not be added to the WFG.

4.4 DDA Termination

It would be beyond the scope of this paper to go into details of the termination of DDAs, so only a brief overview of two possible solutions is given.

A DDA can terminate 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 needed only by active DDAs, a passive DDA terminates on receiving a *forward_end* message from the DDA it forwards to. Another possibility is that a DDA terminates after it has not received any message for a long enough period of time. In the current system we have implemented the first method.

In the following sections we will omit the data structures and the pseudo code regarding termination.

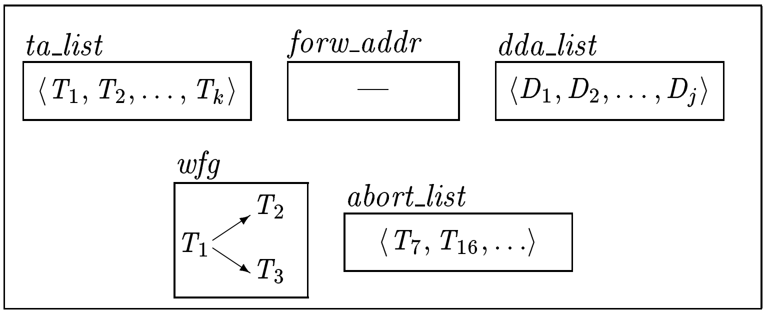

Figure 6: DDA Structure

4.5 Data Structures 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 mean time 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 6.

A DDA is an object having the following special attributes: *ta_list*, *wfg*, *forw_addr*, *dda_list* and *abort_list*. The *ta_list* is 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 reach after the DDA decided to abort it. Since adding this transaction again to the *wfg* could cause the detection of phantom deadlocks, the DDA keeps track of the transactions it has aborted in *abort_list*.

4.5.1 Code for Objects

- 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
    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. Is this not the case, the object chooses the DDA with the lowest identifier from the set of DDAs the transactions $T_1, \ldots, T_n$ are associated with. It is possible, that transactions $T_1, \ldots, T_n$, and T_j are associated to different DDAs. 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.

When determining a DDA, the object relies only on its own information about transactions being associated to DDAs. 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.

4.5.2 Code for Transactions

set_merge_dda and *set_dda* messages sent from different DDAs to a transaction, may not arrive in the order in which they were sent. The transaction deals with this by storing *set_merge_dda* messages that have arrived too early in *dda_merges*, and processing them after the appropriate information has arrived. In other words, it sorts the *set_merge_dda* messages it receives. This is possible since they contain the identifier of the DDA merged into the sender. This is not described here, since it makes the code more complicated and does not contribute to the understanding of the algorithm.

- 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
  case
    the operation request contained a DDA:
        return
    operation_ack does not contain a DDA:
        return
    default:
        proceed as on receive set_dda
        my_dda := DDA_new
  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.

- Transaction T_j receives a *set_dda*(to: DDA_new)

```
receive set_dda(to: 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 of DDA_new and next
        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.

4.5.3 Code for DDAs

For simplicity reasons we will omit some implementation details, such as measures to be taken when *change_forward* messages from different DDAs arrive in wrong order, since they do not contribute to understanding the algorithm. As mentioned earlier, we leave out the description regarding termination.

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 abort_list
  add dependencies to wfg
  update ta_list
  if message does not contain merge_request
  then send merge_request(into:D_j) to all DDAs in the received list,
    check for cycles in wfg
  else (message contains merge_request(into: DDA_merge))
    send merge_request into DDA_merge to all 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 the 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 is sent to DDAs already merged into D_j.

- DDA receives a *merge_request*(into: DDA_new)

```
receive merge_request(into: DDA_new)
  if own_id > DDA_new
  then send merge_into to 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 lower 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.

The *merge_into* message contains the *wfg, ta_list, dda_list* and *abort_list*, of the merging DDA.

- DDA receives a *merge_into* message

```
receive merge_into
  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 to own dda_list
  add received wfg to own wfg
  append received abort_list to own abort_list
  check for cycles in wfg
```

5 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. Detected deadlocks that do not really exist are called *phantom deadlocks*. Deadlocks in database systems are modeled by the resource deadlock model which implies that each deadlock in the system corresponds to a cycle in the WFG. 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.

5.1 All Deadlocks are Detected

Lemma 1 *All outgoing edges from one node (TA) are maintained in the* same *DDA.*

Proof The lemma 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 DDA D_y one transaction is associated with merges into another 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 forwards all dependencies it receives to D_o, so no information is lost. □

Lemma 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.*

Proof As shown in Lemma 1 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 in the global WFG $T_j \rightarrow T_i$ 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. □

Theorem 1 *All deadlocks in the system will be detected and resolved in finite time.*

Proof Let $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$ constitute a cycle of the global WFG. As shown in Lemma 2 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 (Lemma 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. □

5.2 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 7. 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.

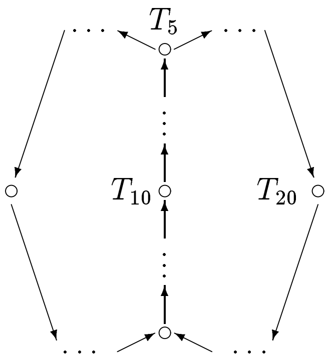

Figure 7: Cycle Overlaps

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 (Lemma 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. The DDA records that it has aborted this transaction, so if dependencies this transaction is involved in arrive later, they will not be added to the graph, and can therefore not lead to phantom deadlocks.

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. 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.

6 Performance Issues

In this section we present the results of a simulation study that compares the DDA scheme with the edge-chasing algorithm introduced in [RB88].

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 CPU server, so at each point of time only one autonomously operating entity (object, transaction, DDA) can be active at one site. A site is responsible for properly scheduling the entities located on it. We omitted modeling I/O servers, since we believe that this does not have an impact on the performance of deadlock detection algorithms and will equally influence the throughput and response time achieved using different algorithms. However, some "thinking time" is reflected in the average duration of operation execution.

Transactions can enter the system at a given point of time, or, as was done in the experiments described here, they can be started at the beginning of a simulation run and the system then preserves the same number of transactions (multiprogramming level, *mpl*) throughout the simulation.

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.

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, while for a committed transaction a new one is started, in order to maintain a constant number of transactions.

6.2 Performance Experiments

We obtained the following performance measures 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 gives the number of committed transactions per second, while the response time is the time a transaction stays in the system, i.e., the time from the moment it was started until its commit. Note that the number of messages and the restart ratio depend only on the deadlock detection and deadlock resolution strategies of the algorithms, not on the estimated durations of operations in the simulation system.

The simulation values used for measuring the number of messages and the restart ratio are given in Figure 8. They are based on values used in other simulation studies ([Buk92],

No. of sites	100
No. of objects	10000
mpl	100, 150, 200, 220, ..., 380, 400, 450, 500, 550 and 600
average tran_size	8
min tran_size	4
max tran_size	12

Figure 8: Simulation Parameter Setting

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

Figure 9: Compatibility Matrix

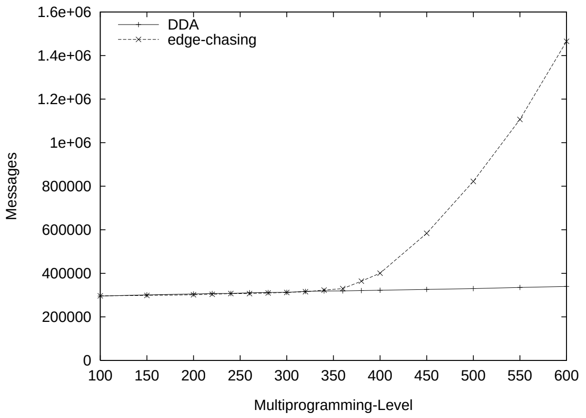

Figure 10: Number of Messages

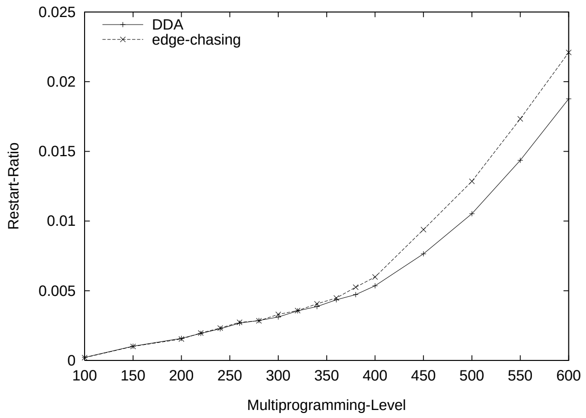

Figure 11: Restart Ratio

[ACM87]). The number of objects used for the simulation runs is relatively small, but larger objects bases lead to 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 9.

The time needed to transmit a message by far exceeds the time the computation needed for deadlock detection requires. Therefore the most important/sensitive parameter of the cost of deadlock detection appears to be the number of messages sent through the communication network. Figure 10 shows the number of messages the two algorithms have sent, for different multiprogramming levels (*mpl*). When the *mpl* is low, i.e., conflicting lock requests are rare, both algorithms send only few messages. The monitored messages are mostly sent on behalf of regular transaction processing and not for deadlock detection. With the *mpl* increasing, the edge-chasing algorithm sends far more messages than the DDA algorithm. The reason is, that in the DDA algorithm every dependency is sent only once from the object manager it occurred at, to a DDA, while the edge-chasing algorithm sends a probe for the same dependency every time the corresponding edge of the WFG could be involved in a deadlock. The DDA algorithm additionally sends messages between DDAs and transactions, and between DDAs when a merge is needed, but this does not occur too often.

Some of the messages sent by the DDA algorithm, such as *merge_into* messages, are

Likelihood of a lock requested being local	60%
Operation execution	25 milliseconds
Undo of an operation	15 milliseconds
Commit (in object)	3 milliseconds
Cycle detection	1 millisecond
DDA creation	1 millisecond
Message delay (local)	3 milliseconds
Message delay (remote)	10 milliseconds
Receiving a message	1 millisecond

Figure 12: Additional Simulation Parameters

longer than the probe messages, but the overhead caused by one long message is much less than the overhead induced by many short messages.

The restart ratio of the algorithms is shown in Figure 11. The restart ratio depends on the deadlock detection and resolution mechanism. Both algorithms in principle choose the youngest transaction to be aborted. The difference is, that the edge-chasing algorithm cannot avoid detecting phantom deadlocks whereas the DDA scheme detects only real deadlocks. When more than one cycle can be resolved by aborting the last transaction closing the cycle, the DDA scheme chooses this one, while the edge-chasing algorithm does not have the information that a transaction is involved in more than one cycle and therefore aborts one transaction for each of them (unless, by coincidence, the same victim is chosen for more than one cycle).

In order to measure the throughput and the response time, the duration of operations, the message delay, and therefore the percentage of local and remote operation requests, is of importance. The values used in our simulations are listed in Figure 12.

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 comparison reasons we also ran simulations with an *mpl* of 1. Since there is no parallelism, a low throughput, about three transactions per second, corresponding to a response time of about 0.33 seconds, is achieved.

The throughput, presented in Figure 13, appears to depend mainly on the number of messages sent by the algorithms. For low *mpls*, where conflicts are rare, the algorithms perform alike, but as the system's load increases the DDA scheme outperforms the edge-chasing algorithm. For an *mpl* of 380 the DDA scheme achieves a 21% higher throughput than the edge-chasing algorithm, and an *mpl* of 400 (450) leads even to a 37% (118%) increase of the throughput compared to the edge-chasing algorithm. The insensitiveness to high system loads is especially important in large-scale and highly distributed systems, since in these systems the load cannot be strictly controlled.

The response time shown in Figure 14 also mainly depends on the number of messages, but also reflects the restart ratio an algorithm induces. The DDA scheme achieves a decent response time even on high system loads, while the edge-chasing algorithm appears to be

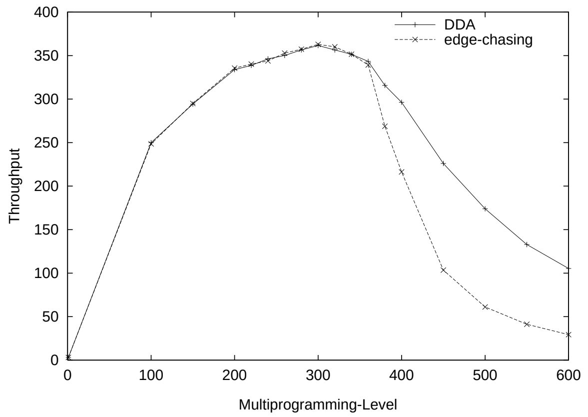

Figure 13: Throughput

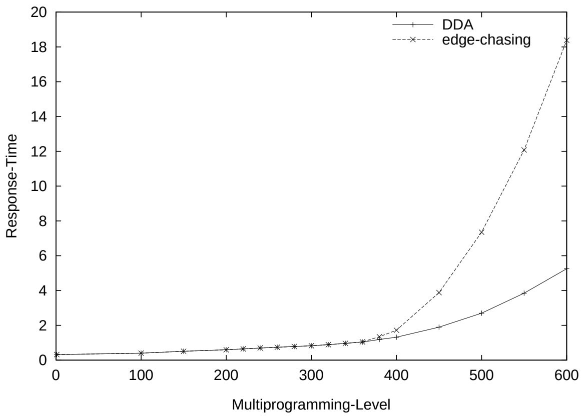

Figure 14: Response Time

highly sensitive to even small changes in the systems load, beyond a certain *mpl* threshold (here 400).

We ran quite a number of simulation experiments, varying different parameters (number of sites, the *mpl*, the percentage of local object accesses, etc.)—they all show the same relative trends. The performance of the algorithms when only few conflicts arise is alike. But as soon as the number of conflicts increases the DDA scheme performs much better, sending substantially fewer messages and aborting much fewer transactions, therefore achieving a higher throughput and a lower response time.

7 Conclusion

In this paper we have devised a new distributed deadlock detection algorithm. The algorithm is based on dynamically created deadlock detection agents (DDAs). A DDA "takes care" of deadlocks within one so-called center of locality comprising transactions accessing the same sets of objects. The DDA scheme is self-tuning in the sense that it adjusts automatically if the system load shifts dynamically.

The simulation study indicates that the fix-costs (i.e., the cost of deadlock detection when very few deadlocks are present) of the DDA scheme is comparable to the edge-chasing algorithms. However, when the system's load increases and more deadlocks are encountered, the DDA scheme generates much fewer messages, aborts fewer transactions and, consequently, has a higher throughput and lower response time. Especially in large-scale distributed systems it will become increasingly difficult to control the system's load and, therefore, one can assume that (some parts of) the system will periodically encounter very high loads. In this case the edge-chasing algorithms appear to induce a prohibitively high deadlock detection overhead while the DDA scheme manages to maintain a decent performance.

To further show the practicability of the DDA approach we are currently implementing other deadlock detection algorithms. As the representative of the path-pushing algorithms we chose the one presented by Obermarck [Obe82], since it is the best known algorithm of this class. We will also compare the performance of the DDA and the timeout approach. As shown in [ACM87], in a centralized system the timeout approach was outperformed by all the deadlock detection and deadlock prevention algorithms, but due to its low communication costs it may be more suitable for decentralized systems.

8 Acknowledgments

We would like to thank Stefan Pröls for his great help on the implementation of the algorithms.

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.

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

[CDF+94] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 383–394, Minneapolis, MI, USA, May 1994.

[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.

[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.

[Gra78] J. Gray. *Notes on Database Operating Systems*, volume 60 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, New York, Berlin, etc., 1978.

[KLMW94] A. Kemper, P. C. Lockemann, G. Moerkotte, and H. D. Walter. Autonomous objects: A natural model for complex applications. *Journal of Intelligent Information Systems (JIIS)*, 3(2):133–150, 1994.

[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.

[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.

[SAL+96] M. Stonebraker, P. M. 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, Jan 1996.

[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.