Synchronization in a Distributed Object System (Kurzbeitrag)

Natalija Krivokapić

Universität Passau
Fakultät für Mathematik und Informatik
94030 Passau, Germany
krivokapic@db.fmi.uni-passau.de

Abstract. We describe the integration of a synchronization protocol in a distributed object system, in particular focusing on autonomous objects. To ensure intra-object serializability a locking-based scheme exploiting the semantics of operations is used. In our system the decision if a lock can be granted or not can be made at the objects' level. The objects' autonomy then facilitates that the lock management can be done locally by the objects themselves, i.e., can be integrated into their behavior. The cooperation between a group of objects working on a common task is managed by control objects corresponding to transactions. These objects enforce a two-phase locking protocol, thereby achieving inter-object serializability. Objects execute operations on behalf of transactions and can autonomously react to the changes on its state these operations cause. The relationship between a transaction and the reactions it causes are incorporated into the system in a way similar to coupling modes as introduced in active database systems.

1 Introduction

The world-wide interconnection creates an ever growing need for large-scale distributed database systems. A recent prototype of a system designed for this purpose is Mariposa [18]. In such systems the coordination of objects working on a common task cannot be done in a centralized manner, but can be achieved only through cooperation between the objects themselves or sites these objects reside on. For these cooperation concepts a high degree of object or site autonomy are needed. Therefore autonomous objects seem to be a promising approach for the implementation of these systems since they provide the needed autonomy and can also in a natural way be extended by cooperation concepts such as [12].

In this paper we present a synchronization concept for a distributed object system that can in particular be used to control the cooperation of multiple autonomous objects involved in a common task. We do not develop a new synchronization scheme. Rather we try to extract those concepts, out of the overwhelming variety of existing ones, that are the most suitable for such a system, and incorporate them into our model.

In [9] an autonomous object model was introduced, based on asynchronous message passing between the objects. Objects can receive messages, interpret them and react to them. When an object has received more than one message it decides nondeterministically, considering their priorities, in which order they will be processed.

Here we will describe the first step towards our goal by incorporating concurrency control to enforce serializability. We will show that the work on synchronizing abstract data types [17,19,6,15] can be adopted for the autonomous object model. This way synchronization can be implemented locally within the autonomous objects, thereby reducing the need for global control. Additionally the semantics of object types' interface operations is taken into account, thus gaining parallelism.

This paper is organized as follows. In Section 2 the most relevant work on related topics is surveyed. The autonomous object model is briefly outlined in Section 3. Section 4 describes the incorporation of a synchronization protocol into the model while Section 5 concludes the paper.

2 Related Work

A lot of research has been carried out on all aspects of synchronization schemes, so only a small subset of it can be mentioned here. Most of the concepts used nowadays are described in great detail in [5].

The majority of existing database systems employ a locking-based concurrency control mechanism. Traditional *read/write* locking schemes are often too conservative in the sense that they lead to more conflicts than necessary. A protocol using operational locking modes was first introduced in [10]. Synchronization of abstract data types has been examined by Schwarz and Spector [17]. They presented a formalism for specifying concurrency properties of shared abstract data types, exploiting the semantics of the type's operations in order to increase concurrency. Our approach is mainly based on this formalism. In [7] and [19] a hybrid concurrency control protocol for abstract data types is developed, again taking the operation's specific semantics into account. Two optimistic concurrency control protocols for abstract data types considering type specific operations are introduced in [6].

Locking mechanisms can ensure serializability on the level of autonomous objects, but additional control, in form of transactions, is needed to enforce inter-object serializability. The behavior of autonomous objects has to be integrated into the transaction model. Active database systems also face this problem. In these systems the relationship between the transaction triggering a rule and the rule itself has to be specified. This is done through coupling modes, as introduced in [8]. To our knowledge the set of coupling modes supported in the HiPAC system [3] subsumes all coupling modes used in other existing close active database systems. The semantics of active database systems, in particular of transactions in these systems, is studied in [4]. The article also gives a good survey of the work done in the area of active databases.

In [1,2] a model for active objects and the integration of a transaction model into this system is described. Also the transaction model is presented in a formal manner, using the ACTA transaction metamodel.

3 Autonomous Object Model

While passive objects consist of data and a behavioral description, i.e., methods that can be invoked to manipulate the data, autonomous objects, also called active objects, are amended by an autonomous behavior. This means that they can autonomously react to events that have occurred.

In our model, introduced in [9], the autonomous behavior of objects relies on asynchronous message passing. A message queue is associated to each autonomous object. An object can receive messages, interpret them and autonomously react to them. It also has the ability to send messages to other objects.

The behavior of objects is specified through a so-called *behavioral map* [9]. It consists of a set of guards that can be viewed as Event-Condition-Action (ECA) rules. A guard is described through the event that can trigger it, an optional priority, an optional precondition and an action. When a guard is triggered, the precondition is evaluated and, in case it is satisfied, the action part of the guard is executed. If a guard is triggered and its precondition evaluates to "true" the guard is called *passable*. If the precondition is omitted the guard is considered passable whenever it has been triggered.

The model supports three kinds of guards: **on**-guards, which handle requests for operation execution coming from the environment (i.e., message sent by other cooperating objects), **at**-guards, which handle time events, and **if**-guards, which handle changes of the object's state.

Consider an autonomous object type *Warehouse* which consists of an array of different items and provides four operations: *supply, consume, quantity* and *inventory*. The operation *supply* increases the available quantity of an item while *consume* decreases it. *Quantity* returns the amount of an item available in the *Warehouse* while *inventory* looks up the quantities of all items of the *Warehouse* object. A possible *behavioral map* for the type *Warehouse* is shown in Figure 1.

Now we will describe the execution model. As mentioned above, all incoming messages are buffered in a message queue until the object decides to process them. The object selects a guard nondeterministically, taking the guards' priorities into account. Afterwards, the message queue is searched for the first message matching the selected guard and evaluating the precondition to "true". If such a message is found, the action part of the guard is executed and the message is removed from the queue. Otherwise a new guard has to be selected.

Objects in our model are strictly sequential, i.e., each object has only a single thread of control. This implies that messages are processed one after another without interleaving.

persistent autonomous type Warehouse **is**
...
behavioral map
on consume(item_i, q) **if** $q \leq 20$ **and** $q \leq$ **self**.quantity(item_i) **priority** 9
do self.consume(item_i, q);
on consume(item_i, q) **if** $q \leq$ **self**.quantity(item_i) **priority** 1
do self.consume(item_i, q);
on supply(item_i, q) **priority** 90 **do self**.supply(item_i, q);
on quantity(item_i) **do return self**.quantity(item_i);
on inventory() **do self**.inventory();
if self.quantity(item_i) $\leq q$ **priority** 5 **do self**.order(item_i, 20);
implementation ...
end type Warehouse;

Fig. 1. Behavioral Map of the object type *Warehouse*

4 Serializing Autonomous Objects

Object autonomy demands that the synchronization control of cooperative tasks (e.g., ACID-transactions) is exerted locally within the autonomous objects—as opposed to a global scheduler. Objects can locally decide whether or not a lock can be granted and are also responsible for writing log records. This drastically reduces the communication overhead imposed by the synchronization protocol. Local synchronization at the level of individual autonomous objects also has the advantage, that the application-specific semantics can be exploited for concurrency control, and thus increases the degree of parallelism.

In this section we will show how semantic locking as proposed by [10,17] can be incorporated in our autonomous object model. Under this concurrency control scheme conflicts are detected at the granularity of high-level application-specific operations as opposed to the primitive *read*/*write*-operations. It is the type implementors responsibility to specify the potential conflicts. For this purpose all interface operations of an autonomous object have to be taken into account and for any pair of operation invocations it has to be established whether or not they commute.

To enforce inter-object serializability we introduce a control object (called transaction) for a group of cooperating objects. These control objects are first class objects in our model. They follow the two-phase locking and two-phase commit protocol to ensure serializability.

Deadlocks are inherent to locking-based synchronization protocols. In our model they are resolved by employing the DDA scheme, introduced in [11].

In addition to concurrency control we also have to incorporate recovery concepts at the level of autonomous objects. In particular for undo-recovery one can exploit the application-specific semantics of high-level operations. Due to space limitations this is not considered here.

4.1 Integrating Semantic Locking into the Object Model

A synchronization protocol based on semantic locking ([10], [17]) seems the most natural choice for an autonomous object model. Semantic locking exploits the semantics of operations to achieve more parallelism than the traditional *read/write* locking protocols, and is therefore more suitable for a system of autonomous objects that naturally yields multiple threads of control. Also, as in object-oriented systems in general, it is easy to decide on the needed lock modes since due to encapsulation, the operations that can change an object's data are limited to the operations provided by the object's type.

To use a semantic based concurrency control scheme for each pair of interface operations of an object type it has to be determined whether or not they commute. Depending on the commutativity of operations the needed lock modes are defined and the compatibility table is specified.

Consider again the object type *Warehouse* introduced in Section 3. For the specification of the commutativity of operations their semantics is of importance. Although both operations *supply* and *consume* read the quantity of an item and attempt to modify it afterwards, due to their different semantics they can be treated differently. Two *consume* operations do not commute while two *supply* operations do. In our example four lock modes are needed, reflecting the four kinds of operations the *Warehouse* object type provides: *ItemSupply, ItemConsume, ItemQuantity* and *Inventory.*

According to the commutativity of operations corresponding to the lock modes, the compatibility between them is determined. As proposed in [17] we include data in the lock instances, achieving even more parallelism. The resulting compatibility table for a *Warehouse* is as follows:

	Lock held by T_i			
Lock request by T_j	ItemSupply(It_1)	ItemConsume(It_1)	ItemQuantity(It_1)	Inventory
ItemSupply(It_1)	$\surd$	—	—	—
ItemConsume(It_1)	—	—	—	—
ItemQuantity(It_1)	—	—	$\surd$	$\surd$
Inventory	—	—	$\surd$	$\surd$

Note that locks obtained by one transaction are compatible with any further lock requests by the same transaction.

Semantic locking considers the commutativity of operations to increase the possible parallelism. Commuting operations have to be executed atomically. Let the operation *supply* be implemented as a sequence of a *read* and an *add* operation and $supply_1$ and $supply_2$ be two invocations of *supply* (of the same item). If the operations are interleaved, e.g., *read* in $supply_1$, *read* in $supply_2$, *add* in $supply_1$ and *add* in $supply_2$, the modification by $supply_1$ will be lost.

In an autonomous object system it can easily be guaranteed that operations are executed sequentially, but problems can arise when referentially shared subobjects are introduced, as described in [16]. Problems can also occur when complex objects are supported by the system and operations can directly be invoked on the complex object as well as on the implementing subobjects, see

[14]. In systems where operations executed on an object can directly invoke operations on their subobjects, locks for operations performed on these subobjects cannot be granted without considering locks granted at the " parent" object. In our model complex objects and referentially shared subobjects are not directly supported. This can be modeled in the system, but the subobjects are still autonomous objects, i.e., can still be accessed only through messages. Therefore the lock management can be done locally within the autonomous objects.

The previously outlined computational model of autonomous objects supports the local synchronization control. The message selection in the behavioral map can naturally be extended to verify that no conflicting (local) locks exist before a request is processed. Also the message queues associated with autonomous objects can be exploited to buffer requests that cannot be satisfied immediately because of conflicting locks.

4.2 Transaction Model

Semantic locking carried out by autonomous objects as explained in the previous section guarantees intra-object serializability, but cannot enforce inter-object serializability. This is achieved through transaction objects which coordinate the cooperation between a group of objects working on a common task. In our model transactions are implemented as "first class" objects.

To access an object a transaction sends a message to it that triggers the needed guard. When the object finishes processing the message, it sends an acknowledgment to the transaction which can then access the next object. At each point of time a transaction has only one outstanding request. While waiting, a transaction is blocked in the sense that it cannot access further objects, but of course it can process control messages, such as *abort*.

Autonomous objects should be modeled in a way that most of their operations can be executed locally. In rare cases an object may need to access other objects while processing a guard. Processing a transaction can therefore lead to a transaction (calling) tree in which the root is the transaction object itself and the other nodes are objects accessed by the transaction.

From an object's point of view an access to another object is atomic. The object sends a message and proceeds only after the "called" object has processed the guard triggered by this message. In the mean time the object is blocked, thereby ensuring the sequentiality of the object.

To ensure serializability we employ a two-phase locking protocol. During the first phase the transaction obtains locks on the accessed objects, in the second phase the transaction initiates the commit (abort) processing, during which the locks are released. The locks are granted and maintained by the accessed objects while the transactions keep track of the objects they have accessed so far. In our model the two-phase commit protocol is used. When a transaction decides to commit, it sends a *prepare-to-commit* message to each of these objects and then waits for their responses. Depending on the objects' votes the transaction sends *commit* or *abort* messages to the objects. In either case the locks held by the transaction will be released.

4.3 Integrating Guards into the Transaction Model

Transactions access objects by sending messages to them, requesting the execution of a certain operation. Messages received by an object trigger guards specified for the objects' types. Before an object can start processing a message it has to grant the locks needed for condition evaluation to the corresponding transaction. According to the two-phase locking protocol we employ here, the transaction holds the lock until it commits or aborts, no matter whether the triggered guard is passable, or its processing has to be delayed.

We use latches in order to avoid unnecessary holding of locks. They can be acquired by a transaction and in case the guard is passable they are converted into locks, otherwise they are freed.

On- and at-guards are triggered through messages containing the identifier of the transaction to which the needed locks should be granted. On the other hand if-guards are triggered through changes in the object's state. These changes are caused by processing the guards' actions. In general it is not known which transaction is responsible for the changes, so the locks needed for an if-guard cannot be associated with a certain transaction. Therefore if-guards would have to be executed in separate transactions. To avoid this we introduce "binding" of if-guards to guards that can trigger it.

"Binding" if-guards to other guards is similar to the concept of coupling modes as used in active database systems. Coupling modes as well as the "binding" of guards specify the relationship between a transaction and the processing of the triggered event.

4.3.1 Using Latches

The first step in processing a triggered guard is to try to grant the locks needed for condition evaluation to the corresponding transaction. Afterwards the condition is tested and depending on whether or not the guard is passable locks for the action part of the guard are requested or the guard's processing is postponed. In both cases the transaction keeps the already granted locks.

To avoid holding locks that are actually not needed we use latches. Then for condition evaluation only the latches are acquired by the transaction. If the condition evaluates to *true* the latches are converted into the corresponding locks. If not, the latches are freed and can be obtained by another transaction, thereby increasing concurrency.

A latch can be acquired only if:

- the latch is not already held by another transaction, and
- the corresponding lock does not conflict with any locks that have already been granted.

Using latches instead of locks while the evaluation of a condition does not lead to incorrectness since the objects are sequential and testing the condition does not involve any updates. The effect of using latches is the same as having a "smart" scheduler, foreseeing in which order guards could be processed. This "scheduler" does not prevent starvation. In the current system it is left to the programmer to implement object types in a way that every guard's condition will be satisfied at some point of time.

4.3.2 "Binding" If-guards

As described above, an autonomous object selects a guard nondeterministically and then checks whether the guard is triggered or not. On- and at-guards are triggered through incoming messages. They contain the identifier of the transaction accessing it, so that the object can grant the needed locks and write appropriate logs. The situation is different for if-guards since they are triggered by the execution of the action part of another guard.

It is possible that between two successive selections of a certain if-guard more than one guard that can trigger it has been processed. In this case the object cannot determine which guard has triggered the if-guard and can therefore not decide to which transaction the locks needed for it should be associated with. Therefore if-guard have to be executed in a separate transaction.

In most cases locks needed for an if-guard will not be compatible with locks granted for the on- or at-guard that has triggered it. Consider the *Warehouse* example again. A *consume* could trigger the if-guard, but the needed locks are not compatible. This would imply that if-guards could often be triggered only by already committed transactions.

To allow if-guards to be triggered within a transaction we introduce the possibility to force the selection of an if-guard after a guard has been executed. This is achieved by "binding" an if-guard to another guard. After a guard has been processed the conditions of all guards bound to it are tested and those which are passable are executed. One if-guard can be bound to more than one guard. For example, in the *Warehouse* specification the if-guard could be bound to both *consume* guards. Every time one of these guards is processed, it will be checked whether or not the if-guard is passable.

4.3.3 Relationship to Coupling Modes

The guards' execution in our model can be compared to rule processing in active database systems. The underlying semantics of these two models are different, but the execution models are similar. Receiving a message by an autonomous object corresponds to an event occurrence. Selecting a guard and finding a matching message is a counterpart to event detection, while condition evaluation and action execution are equivalent in both models.

In active database systems coupling modes are used to specify the relationship between the transaction triggering a rule, i.e., causing the triggering event, and the condition and action part of this rule [13,8]. To our knowledge all closed active database systems support some subset of the coupling modes used in HiPAC[8]. There four coupling modes are introduced: **immediate**, implying that the condition (action) is executed immediately after the event is detected (the condition is evaluated), **deferred**, imposing that the condition's (action's) execution is postponed until the commit of the triggering transaction, **detached**, leading to the creation of a separate transaction for the execution, and **detached but causally dependent**, also creating a separate transaction for the execution, but allowing this transaction to commit only if the triggering transaction commits.

In our model coupling modes are specified implicitly. The coupling between condition evaluation and action execution is always immediate. They are executed successively and within the scope of the same transaction. For on- and at-guards the coupling between event detection and condition evaluation is also immediate. Both are executed within the triggering transaction.

As described above, in general for if-guards it is not known by which transaction they are triggered so they have to be executed in a detached mode. Binding if-guards to other guards imposes the selection of a particular if-guard after another guard has been processed. In this case the if-guard is executed within the transaction of the guard it is bound to. This way immediate coupling between the triggering transaction and the if-guard is achieved.

5 Conclusion

In this paper we have described the integration of a synchronization protocol into a distributed object system, in particular a system of autonomously operating objects. We have shown how semantic locking can be incorporated into our autonomous object model and how the advantages of this locking mechanism can be exploited. The lock management is integrated into the behavior of objects thus supporting the objects' autonomy. We have also described how transactions, implemented as autonomous objects, can enforce inter-object serializability, employing a two-phase locking and two-phase commit protocol. The behavior of autonomous objects is also incorporated into the transaction model.

Acknowledgments

We wish to thank Professor Alfons Kemper for his valuable suggestions on this work. We also want to thank the anonymous referees for their useful comments.

References

1. A. Buchmann. Modeling heterogeneous systems as an active object space. In A. Dearle, G. M. Shaw, and S. B. Zdonik, editors, *Implementing Persistent Object Bases*, pages 279–290, San Mateo, CA, USA, 1991. Morgan-Kaufmann Publishers. (Proc of the Fourth Intl. Workshop on Persistent Object Systems, Martha's Vineyard, MA, USA, Sep 1990).
2. A. Buchmann, M. T. Özsu, M. Hornick, D. Georgakopoulos, and F. A. Manola. A transaction model for active distributed object systems. In A. K. Elmagarmid, editor, *Database Transaction Models For Advanced Applications*, The Morgan Kaufmann Series in Data Management Systems, pages 123–158. Morgan-Kaufmann Publishers, San Mateo, CA, USA, 1992.

3. U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauhari. The HiPAC project: Combining active databases and timing constraints. *ACM SIGMOD Record*, 12(1):51–70, 1988.
4. P. Fraternali and L. Tanca. A structured approach for the definition of the semantics of active databases. *ACM Trans. on Database Systems*, 20(4):414–471, December 1995.
5. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan-Kaufmann Publishers, San Mateo, CA, USA, 1993.
6. M. Herlihy. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Trans. on Database Systems*, 15(1):96–124, March 1990.
7. M. P. Herlihy and W. E. Weihl. Hybrid concurrency control for abstract data types. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 201–210, Austin, TX, March 1988.
8. M. Hsu, R. Ladin, and D. McCarthy. An execution model for active DB management systems. In *Proc. of the Third Intl. Conf. on Data and Knowledge Bases: Improving Usability and Responsiveness*, pages 171–179, Jerusalem, Israel, June 1988. Morgan-Kaufman.
9. 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.
10. H. F. Korth. Locking primitives in a database system. *Journal of the ACM*, 30(1):55–79, January 1983.
11. N. Krivokapić, A. Kemper, and E. Gudes. Deadlock Detection Agents: A distributed deadlock detection scheme. Technical Report MIP-9617, Universität Passau, 94030 Passau, Germany, November 1996.
12. L. Liu and R. Meersman. The building blocks for specifying communication behavior of complex objects: An activity-driven approach. *ACM Trans. on Database Systems*, 21(2):157–207, June 1996.
13. D. R. McCarthy and U. Dayal. The architecture of an active database management system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 215–224, Portland, OR, USA, May 1989.
14. P. Muth, T. C. Rakow, G. Weikum, P. Brössler, and C. Hasse. Semantic concurrency control in object-oriented database systems. In *Proc. IEEE Conf. on Data Engineering*, pages 233–242, Vienna, Austria, April 1993.
15. R. Rastogi, H. F. Korth, and A. Silberschatz. Strict histories in object-based database systems. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 288–299, Washington, DC, May 1993.
16. R. O. Resende, D. Agrawal, and A. E. Abbadi. Semantic locking in object-oriented database systems. In *Proc. of the ACM Conf. on Object-Oriented Programming Systems and Languages (OOPSLA)*, pages 388–403, 1994.
17. P. M. Schwarz and A. Z. Spector. Synchronizing shared abstract types. *ACM Trans. Computer Systems*, 2(3):223–250, 1984.
18. 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.
19. W. E. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Trans. Programming Languages and Systems*, 11(2):249–282, April 1989.