Towards Modular Concurrency Control for Object-Oriented Distributed Systems*

Rachid Guerraoui

Département d'Informatique
Ecole polytechnique Fédérale de Lausanne
CH-1015, Lausanne, Suisse

Abstract

Object-based distributed systems are intended to merge the benefits of both object-based languages and distribution. To make such systems useful, it is however necessary to extend the basic object model so that it can permit the expression of distribution related aspects. In this paper, we discuss the issues raised by designing transaction concurrency control within an object-oriented distributed system. We propose an original approach based on conditions bound to each object operation. These conditions allow to express various concurrency control policies, e.g. pessimistic or optimistic, all in an orthogonal way to the serialisation protocol. By confining the concurrency control code within dedicated conditions, we enhance both encapsulation and inheritance.

Keywords: Atomic object, concurrency control, failure recovery, modularity, serialization protocol, specification, transaction.

1 Introduction

Merging object-oriented languages and distributed systems has given birth to an important investigation area which aims to build so-called object-oriented distributed systems. In such systems, objects may be shared by concurrent activities and have their coherency corrupted. To preserve coherency, it is necessary to provide concurrency control mechanisms that avoid unlegal interferences.

*Appears in Proc IEEE International Workshop on Future Trends in Distributed Computing Systems. 1993. Research supported by the Swiss FNS under contract number 5003-034344.

Many authors [8] [2] [10] have discussed the design of object concurrency controls that preserve internal state invariants, e.g. a bank account balance has always to be positive. Preserving this intra-object coherency goes through scheduling object invocations (scheduling in the small). Another form of coherency, which we call inter-object coherency, has also to be considered in distributed systems. It concerns inter-objects invariants, e.g. the sum of two bank account balances has always to be constant. This coherency may be preserved by scheduling atomic groups of consistent objects invocations, namely transactions (scheduling in the large). Transaction atomicity reduces the difficult problem of maintaining the coherency of a concurrent system subject to failures, into the simpler problem of maintaining the coherency of failure free sequential subsystems [12].

Applying early works on transactions to object-based distributed systems leads to consider uniform mechanisms which view objects as simple data items on which read and write operations are performed. To guarantee atomicity, transactions are not allowed to perform interfering updates on shared objects. This concurrency limit is not sensible for long transactions with localised concurrency bottlenecks (*hot-spots*).

This paper discusses the issues raised by designing transaction concurrency control within an object-oriented distributed system. It presents an original approach based on a generic protocol and a set of modular linguistic supports. First, each object owns its local control which can be tuned to the desired concurrency, with no repercussions on its clients. Second, object concurrency control is orthogonal to the protocol used to serialise transactions. The latter can thus be redefined in each application with no effects on objects. Finally, the concurrency control is separated from object operations and dispatched into several concurrency control conditions. This enhances the reuse of both the operations and the conditions into subclasses. Although many recent concurrency control proposals address some of these modularity features [15] [7] [14] [11] [17], we know of none which provides them all. Note that we do not propose a new concurrency control algorithm, we rather describe a modular approach that enables to merge various controls within a same application.

The paper is organised as follows. The next section proposes five design requirements that have to be fulfilled by a concurrency control in order to avoid interferences with object-orientation and needless concurrency restrictions. Section 3 presents our underlying framework composed of atomic objects, transaction managers and a serialisation protocol. Section 4 introduces concurrency control conditions and describes two examples of local concurrency controls: a pessimistic and an optimistic one. Section 5 constrasts related works with our approach, and concludes with some general remarks.

2 Five design requirements

This section presents five concurrency control requirements which we have identified as desirable in an object-oriented distributed system.

2.1 Encapsulation

To enforce encapsulation, each object must contain its own concurrency control. The rationale of this requirement is to permit modifications of internal implementations without effects on client implementations [13]. Consider for instance, a modification of an operation implementation which leads to a concurrency control modification, e.g. a *read-only* implementation which is changed for a *write* implementation affects the concurrency control. If the latter is implemented within the object itself, the modification would not have any repercussion on client implementations.

Therefore, objects have not only to protect their internal states against concurrent invocations, but they also have to guarantee atomicity of transactions using them by providing adequate concurrency control [15].

2.2 Inheritance

To enhance the reuse of operations into subclasses, it is necessary to separate the concurrency control code from operation implementations. Adding an operation to a subclass would lead to a concurrency control modification. Such modification would not have any effect on the operations if the concurrency control is designed in an independent manner [8] [10].

Moreover, to ease the reuse of the concurrency control itself, it is important to dispatch it into several parts. It is thus possible to refine parts of the concurrency control without modifying the whole.

2.3 Heterogeneity

Transaction concurrency control has been studied for over twenty years [12] [1]. On the one hand, pessimistic concurrency controls avoid conflicts by delaying or rejecting conflicting transactions. On the other hand, optimistic concurrency controls allow conflicts to occur and provide certification protocols at transaction commit times.

Pessimistic approaches are well suited for situations where many conflicts occur, while optimistic ones are better suited for situations with few conflicts. In a single application, some objects may cause more conflicts than others. Thus, the choice of a particular approach (i.e. pessimistic, optimistic) must be done at a per object basis, according to the object use. Hence, it is important to provide a way of defining heterogeneous concurrency controls within the same application.

2.4 Semantics

It is useful to enhance concurrency by taking into account object operation semantics. Consider for example two transactions incrementing a counter. An incrementation does not really read the value of the counter at a logical level, although at a lower level a read operation may be needed. Two incrementations are thus said to commute [16]. Identifying commutative operations allows to consider as atomic, executions which would not be considered as such in a more syntactic approach.

Therefore, programmers must be provided with a set of linguistic supports that permit the use of object operation semantics within concurrency control.

2.5 Orthogonality

Atomicity infers that each execution appears as if transactions were executing in a sequential order, called the serialisation order. Since this order must be the same for all objects, a mechanism called serialisation protocol defines this order and ensures the agreement among objects. The serialisation protocol is crucial as it determines many characteristics of the concurrency control algorithm, such as deadlock occurrence and application performances [11].

Since no serialisation protocol is better than others [11] [17], it is desirable for this protocol to be hidden from each object concurrency control. This permits to switch from a protocol to another without modifying concurrency controls.

3 Computational model

To achieve our five requirements, we have designed a concurrency control mechanism and a set of associated linguistic supports.

We view transactions as execution units and objects as data containers accessible through their operations. When executing, a transaction invokes operations on objects through requests, receives the corresponding replies, and terminates either by aborting or committing. Transactions may be nested: beside objects invocations, each transaction may encompass subtransactions, which may contain requests as well as sub(subtransactions), and so on [9]. Since subtransactions may act concurrently and abort independently, nesting transactions permits to introduce concurrency within transactions and to confine failures within subtransactions. We have shown in [4], how nested transactions can be efficiently used in an object-based distributed system.

The basic framework supporting our concurrency control mechanism is compound of atomic objects, transaction managers, and a serialisation protocol (Figure 1).

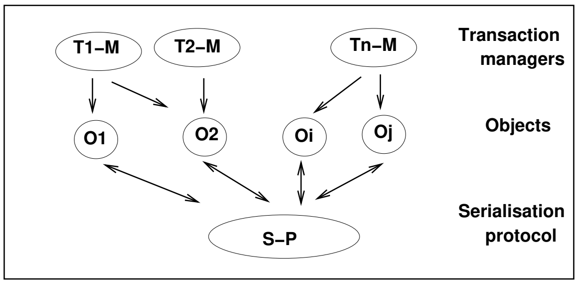

Figure 1: The framework

3.1 Objects

We focus in this paper on particular objects, called atomic objects [17], for which transaction concurrency control is necessary. For example, an object local to a transaction does not require any concurrency control. In our framework, classes of which instances are atomic objects are called atomic classes. They are derived through inheritance from a basic class, noted `AtomicObject`.

Each atomic object supports the operations `askcommit()`, `commit()` and `abort()`, so that it can be informed about the outcome of each transaction (terminated, committed or aborted). Furthermore, each atomic object maintains a log of previously executed requests which associated transactions have not committed yet. When an atomic object receives an `askcommit()`, it checks whether the transaction is locally allowed to commit or not (next section), and returns either a positive or a negative reply. When the object receives a `commit()`, it deletes the corresponding requests from its log. If the object receives an `abort()`, it discards the transaction modifications by performing appropriate *reverse* requests. This supposes that there exists a reverse request corresponding to each possible request. For example, the reverse request of a $deposit(5)$ would be a $withdrawal(5)$.

3.2 Transaction managers

We associate a manager to each transaction. A transaction manager records the identifiers of all the accessed atomic objects and informs them about the outcome of the transaction. A transaction is said to be active when it is performing some operation, and pending when it is blocked on some condition. When the transaction terminates successfully, its manager starts a two phase certification protocol:

1. The transaction manager sends `askcommit()` to all accessed atomic objects and waits for their replies.

2. If the replies are all positive, the transaction manager sends out `commit()` to all accessed atomic objects. Otherwise, the transaction is aborted and its manager sends out `abort()` notifications.

The certification protocol may be combined with a two phase commit protocol in order to ensure commit failure-atomicity [3].

3.3 Serialisation protocol

For each execution, a serialisation order is determined through a particular mechanism common to all atomic objects. This mechanism is hidden from concurrency control implementations, through a specific operation, noted `order()`, and defined as follows:

- `order(Ti,Tj) = true;` means that Ti is serialised before Tj.

- `order(Ti,Tj) = false;` means that Tj is serialised before Ti
- `order(Ti,Tj) = unknown;` means that the serialisation order between Ti and Tj is yet unknown.

An incoming transaction cannot be serialised before a transaction which has accessed the object, and has already committed. This is guaranteed by aborting each incoming transaction which violates this property.

The operation `order()` is defined within the basic class `AtomicObject`. By default, this operation implements a static serialisation protocol based on transactions' timestamps. Each transaction is bound to a timestamp at creation time. The order of timestamps determines the serialisation order of transactions. An atomic object may then know the serialisation order between two transactions at the time of their accesses.

The serialisation protocol may be changed by modifying the implementation of the operation `order()`, within the class `AtomicObject`. This is an important feature of a transaction-based distributed system since there is no best serialisation protocol for all applications [11][17]. We may for example implement a serialisation protocol based on commit times of transactions [11]. An atomic object will know the serialisation order between two transactions $T1$ and $T2$, if either they are both committed, or $T1$ invokes an operation later.

```
class AtomicClassExemple   : public AtomicObject   {
   public:
      Operation(bar)   { ... }
        Pre(bar)    { ... /*  pre−condition attached to operation bar()    */ }
        Wait(bar)   { ... /*  wait−condition attached to operation bar()   */ }
        Post(bar)   { ... /* post−condition attached to operation bar()    */ }

      Operation(foo)   { ... }
        Pre(foo)    { ... /* pre−condition attached to operation foo()   */ }
        Wait(foo)   { ... /*wait−condition attached to operation foo()   */ }
        Post(foo)   { ... /* post−condition attached to operation foo()   */ }

      ...
};
```

Figure 2: An atomic class

4 Modular concurrency control

As any other object, an atomic object contains a state, as well as a set of operations through which this state may be accessed. In addition, an atomic object contains three concurrency control conditions associated to each of its operations: a *pre-condition*, a *wait-condition*, and a *post-condition*. The conditions are defined within each atomic class.

Figure 2 shows an example of a definition of an atomic class. This class, named `AtomicClassExample` inherits from the basic class `AtomicObject`. Each operation defined in this class, i.e. `bar(), foo()` is bound to three conditions: a pre-condition, a wait-condition and a post-condition.

4.1 Concurrency control conditions

When a transaction accesses an atomic object, it first executes the conditions associated to the required operation. Pre-conditions and wait-conditions are boolean expressions which decide whether to abort, to block, or to accept the performing transaction under some termination constraints. The post-condition does not entail the transaction acceptance, but it allows to link the transaction with a set of completion constraints. More precisely:

- A transaction starts by performing the pre-condition. If the latter is evaluated to true, the transaction is aborted. Otherwise the transaction performs the wait-condition.
- If the wait-condition is evaluated to true, the transaction is blocked until the object receives a termination notification from another transaction (`commit()` or `abort()`. When it resumes, the transaction restarts the protocol by performing the pre-condition. When both the pre-condition and the wait-condition are evaluated to false, the transaction is said to be accepted.
- Before performing the required operation, the transaction executes the post-condition. The latter allows to accept the transaction under some completion constraints that may depend on a serialisation order, a transaction outcome etc.
- When a transaction terminates, its manager sends `ask-commit()` to all accessed atomic objects. Each object checks whether all the completion constraints linked to the transaction are evaluated to true. If one of them is false, the object replies with a negative answer, infering the abort of the transaction. Otherwise, if one of these constraints cannot be evaluated yet, the object blocks the certification protocol and resumes it when a termination notification arrives. If all the constraints are evaluated to true, the object sends out a positive reply.

Customising local concurrency control comes down to implement the conditions within each atomic class.

Pre-conditions and wait-conditions may be used to provide a pessimistic concurrency control by aborting or delaying conflicting transactions. Post-conditions may be used to provide an optimistic concurrency control by postponing the treatment of conflicts until commit time.

4.2 Linguistic supports

Conditions can be implemented using: (i) the object internal state; (ii) the identifier `t` of the incoming transaction; (iii) a list, noted `args`, containing the arguments of the requested operation; (iv) a list, noted `currentList`, containing the requests associated to all the active transactions currently accessing the object; (v) a list, noted `waitingList`, containing the requests associated to all the transactions pending on the object; (vi) a list, noted `accessedList`, containing the requests associated to all the transactions that have already accessed the object and have not terminated yet. These linguistic supports permit:

1. to determine for each incoming transaction T, whether a concurrent transaction T' has accessed the object, and how it is serialised with T (using the operation `order()`).

2. to know the operation that has been performed by T' as well as the associated arguments.

3. to link T with a set of completion constraints that have to be fulfilled at commit time of T.

Through these supports, programmers may implement concurrency control conditions in various ways.

```
class BankAccount1    : public  AtomicObject    {
 public:
   Operation(deposit)    { ... }
      Pre (deposit)  {  return(accessedList.in(after, withdrawal));   }
      Wait (deposit) { return(currentList.in() // accessedList.in()(withdrawal));    }
      Post(deposit)   { }
   Operation(withdrawal)   { ... }
      Pre (withdrawal)  { return(accessedList.in(after));   }
      Wait (withdrawal)  { return(currentList.in() // accessedList.in());   }
      Post (withdrawal)    { }
};
```

Figure 3: A pessimistic concurrency control

Figure 3 shows the definition of an atomic class providing a pessimistic concurrency control. This class, noted `BankAccount1`, is atomic as it inherits from the basic class `AtomicObject`. The associated concurrency control is said to be pessimistic because all conflicts are treated in an *a priori* manner. The pre-condition ensures that an incoming transaction is aborted if it is serialised before a conflicting transaction that has already accessed the object. It uses the list `accessedList`. The wait-condition ensures that two operations cannot perform concurrently, and that each incoming transaction is blocked until a conflicting transaction has terminated. It uses both `currentList` and `accessedList`. Using the commutativity property of operations, we eliminate the conflict between two `deposit()` executions [17]. As the post-condition is *empty*, no completion constraint is linked to an accepted transaction. Consequently, each transaction which terminates successfully is locally allowed to commit.

```
class  BankAccount2     : public  BankAccount1   {
  public:
    Pre (deposit)   { return(false); }
    Wait (deposit) { return(currentList.in() // accessedList.in(after, withdrawal))}
    Post(deposit)   { accessedList.cons(withdrawal, t, commit, before);   }

    Pre (withdrawal)    { return(false); }
    Wait (withdrawal)    { return(currentList.in() // accessedList.in(after))}
    Post (withdrawal)     { accessedList.cons(t, commit, after);   }
 };
```

Figure 4: An optimistic concurrency control

Figure 4 shows the definition of an atomic class providing an optimistic concurrency control. This class, noted `BankAccount2`, is atomic since it inherits from the atomic class `BankAccount1`, and thus indirectly from the class `AtomicObject`. The associated concurrency control is more optimistic than the previous one because some conflicts are treated *a posteriori*, i.e. at commit time. The wait-condition is used to ensure that two operations cannot perform concurrently, and that an incoming transaction T is blocked if it is serialised before a conflicting transaction T' that has already accessed the object. The post-condition creates dependencies between transactions. Indeed, the post-condition checks whether a conflicting transaction T'', has already accessed the object and has not terminated yet. If so, the post-condition creates a completion constraint linked to T, requiring that T'' must commit and be serialised before T. During the certification protocol, i.e. when executing the `askcommit()` operation, the object guarantees that all the completion constraints linked to T are satisfied. Otherwise, T is aborted. As in the previous example, the commutativity property is exploited to eliminate the conflict between two `deposit()` executions.

4.3 Modularity features

Since the concurrency control is encapsulated within a class implementation, it may be modified without effects on other classes. Consider for example, a class C using a pessimistic concurrency control. If the instances of C constitute concurrency bottlenecks, i.e. *hot-spots*, its concurrency control may be changed for a more optimistic one, without effects on clients of C.

As the concurrency control of C is confined within specific conditions and separated from operations, it may be refined within a subclass C', without effects on the operations inherited from C. This is shown on figure 4 since although refining the concurrency control conditions within the class `BankAccount1`, the `deposit()` and `withdrawal()` operations are directly inherited. In addition, since the concurrency control is dispatched among different conditions, a part of the concurrency control may be redefined without modifying the whole.

The concurrency controls showed on figure 3 and 4 do not assume a particular serialisation protocol. The only assumption made is that: there exists an underlying protocol, common to all objects. This protocol is implemented in the operation `order()`, within the basic class `AtomicObject`. It is thus possible to change the serialisation protocol of an application by redefining the operation `order()`, without effects on atomic object concurrency controls.

In [6], we describe a local specification property for atomic objects, named *orthogonal atomicity*, which allows modular and highly concurrent implementations, while still ensuring the atomicity of transactions.

5 Concluding remarks

Various linguistic supports have been presented in [15] [7] [14] [11] [17] to express concurrency control between transactions. The proposed models in [15] [7] [14] [17], permit the use of objects semantics to increase concurrency. Nevertheless, the associated mechanisms do not achieve our orthogonality requirement since they assume particular serialisation protocols. In [15] [14] [17], they assume a serialisation protocol based on the order according to which transactions access objects, while in [7], it assumes a serialisation protocol based on the commit time order of transactions.

Only [11] proposes a mechanism based on object histories which hides the underlying serialisation protocol. However, this proposal lacks in flexibility since objects have to follow a uniform pessimistic algorithm. A concurrency control protocol, based on conditions bind to operations, has been proposed in [2]. The proposed mechanism is similar to ours in the way of dealing with encapsulation and inheritance. Nevertheless, transaction concurrency control is done in an *ad hoc* manner and does not allow neither to mix pessimistic and optimistic approaches within a same application, nor to modify the serialisation protocol.

Our proposed concurrency control model has been designed and implemented in C++, on top of a Unix plateform [4] [5]. However, since it does not map to any particular object model, we believe that it may be successfully used in many object-oriented distributed systems. Among various extensions of our model, we are investigating within the GARF project, a way to attach concurrency control mechanisms to individual objects instead of classes. Another possible extension would concern recovery. Indeed, we have only considered a particular "Do/Undo" protocol. A dual work would lead to customise the recovery protocol at a programmer convenience, as it is done for concurrency control.

Acknowledgements

I am very grateful to Jean Ferrié and Bill Weihl for their constructive comments, and to Karim Mazouni for reading and discussing preliminary drafts of this paper.

References

[1] A.J. Bernstein and P.M. Lewis - "Concurrency in Programming and Database Systems" - ed Jones and Bartlett - 1993.

[2] D. Decouchant, P. Le Dot. M. Riveill, C. Roisin and X. Rousset de Pina -"A Synchronisation Mechanism for an object-Oriented Distributed System"- *IEEE Proceedings of the International Conference on Distributed Computing Systems* - May 1991.

[3] J.N. Gray - "Notes on data base operating systems" - Operatings Systems: An Advanced Course - LNCS, Springer - 1978.

[4] R. Guerraoui, R. Capobianchi, A. Lanusse and P. Roux -"Nesting Actions through Asynchronous Message Passing: the ACS protocol" - *Proceedings of the European Conference on Object Oriented Computing* - June/July 1992.

[5] R. Guerraoui, R. Capobianchi, A. Lanusse and P. Roux -"Atomic Asynchronous Objects Invocations" - *IEEE Proceedings of the International Workshop on Fault-Tolerant Distributed and Parallel Systems* - July 1992.

[6] R. Guerraoui -"Modular Atomic Object" - Internal Report DI-LSE 93-2 - Ecole Polytechnique Fédérale de Lausanne - June 1993.

[7] M. Herlihy and J.M. Wing -"Avalon: Language support for reliable distributed system" - *IEEE Proceeding of the International Symposium on Fault-tolerant Computer Systems* - July 1987.

[8] D. Kafura and K.H. Lee -"Inheritance in actor based concurrent object-oriented language" - *Proceedings of the European Conference on Object-Oriented Programming* - July 1989.

[9] J.E.B. Moss -"Nested Transactions: An approach to reliable distributed computing" - Ph.D, thesis, MIT, MA - 1981.

[10] C. Neusius -"Synchronising Action" - *Proceedings of the European Conference on Object Oriented Computing* - June 1991.

[11] T.P. Ng - "Using Histories to Implement Atomic Object" - *ACM Transactions on Computer Systems* - November 1989.

[12] C.H. Papadimitriou - "The Serializability of Concurrent Database Update" - *Journal of the ACM* - October 1979.

[13] M. Papathomas - "Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming" - Ph.D Thesis, Université de Genève - 1992.

[14] G.D. Parrington and S.K. Shrivastava - "Implementing Concurrency Control in Reliable Distributed Object-Oriented System" - *Proceedings of the European Conference on Object Oriented Computing* - August 1988.

[15] W. E. Weihl and B.H. Liskov - "Implementation of Resilient, Atomic Data Type" - *ACM Transactions on Programming Languages and Systems* - April 1985.

[16] W. E. Weihl - "Commutativity-based Concurrency Control for Abstract Data Types" - *IEEE Transactions on Computing* - December 1988.

[17] W. E. Weihl - "Linguistic Support for Atomic Data Type" - *ACM Transactions on Programming Languages and Systems* - April 1990.