On Object Orientation as a Paradigm for General Purpose Distributed Operating Systems

Vinny Cahill [*], Seán Baker, Brendan Tangney, Chris Horn and Neville Harris
Distributed Systems Group, Dept. of Computer Science, Trinity College, Dublin, Ireland.

Proceedings of the 1992 ACM SIGOPS European Workshop

December 9, 1992

Abstract

In the Amadeus project we have been considering the construction of a general purpose distributed support environment for object oriented programming. In this paper we tackle a number of key areas whose interaction must be addressed in the design of such a general purpose object support system: 1) integration of support for (object oriented) database systems; 2) integration of security mechanisms suitable for objects; and 3) operating system support to allow object oriented applications exploit the inherent parallelism of the underlying distributed environment.

1 Introduction

A number of recent projects have considered the construction of operating systems – particularly distributed operating systems – which support the object oriented paradigm. Such systems have typically supported a single programming language and have been targeted at particular application environments. In the Amadeus project we have been considering the construction of a general purpose support environment for object oriented programming.

In this paper we identify a number of key areas whose interaction must be addressed in the design of such a **general purpose** system, and describe the approaches which we have taken in supporting each. The areas considered are 1) integration of support for (object oriented) database systems, including managing associative access to large collections of objects and the integration between programming languages and database systems; 2) integration of protection mechanisms suitable for large numbers of (potentially) small objects using only conventional hardware and 3) operating system support, in the form of load balancing and clustering of objects[1], to allow object oriented applications to exploit the inherent parallelism of the underlying distributed environment. We begin by giving a brief overview of Amadeus.

[*]email vjcahill@cs.tcd.ie
[1]Explicit allocation of processors is also supported but is not addressed here.

2 Amadeus Overview

The Amadeus model is based on passive objects which are accessed by distributed processes. An Amadeus system may include processes running on behalf of many different users sharing objects in a secure way. The basic unit of computation in Amadeus is a *job* which is a distributed process consisting of a set of *activities*. Activities are distributed threads of control i.e. analogous to lightweight processes but with the possibility of executing in several address spaces or *contexts* at the same or different nodes at different times.

In Amadeus, *clusters* are used to store groups of related objects. Clusters are the fundamental units of data managed by the system. In particular, clusters are the units of storage and mapping.

Finally, Amadeus supports atomic transactions as a means of ensuring consistency of data in the presence of concurrency and (partial) failure, through the use of the RelaX transaction manager and libraries [6, 5].

3 Security in Amadeus

An object support system such as Amadeus must provide access control for objects in order to ensure that only authorized users can invoke operations on a given object. Moreover, when (authorized) users invoke an operation on some object they have no guarantee that there will not be side effects if they did not write the code for the invoked object themselves. This will be a common occurrence in object oriented environments characterized by high degrees of code reuse and sharing. In effect, an object may act as a Trojan horse, allowing damage to be inflicted upon other objects accessible to the invoker and providing an unauthorized access path to confidential information.

Amadeus provides access control at the level of individual operations – using operation based access control lists and isolation of untrustworthy objects (code) at runtime. Key to the provision of security is the notion of an *extent*.

Abstractly an extent is a group of objects belonging to a common owner. Each user may own many extents but an object may belong to only one extent at any time. This mechanism allows objects which are not trusted to be isolated in a separate extent.

At runtime, each extent is represented by a set of contexts. Objects of a given extent are only mapped into the context(s) belonging to that extent and never into the context of another extent. Thus it is impossible for an object in one extent to read or interfere with an object in another extent without an authorization check being performed – since objects belonging to different extents are always separated by a hardware protection boundary.

An activity can access objects in its current extent directly without an access check, and potentially without using their code, that is, by direct access to virtual memory. However if an activity attempts to access an object in a different extent, it can only do so by performing a cross-extent invocation via the Amadeus kernel, resulting in an authorization check being performed by the target extent using the target object's access control list. If the invocation is accepted by the target object (extent) then, before the invocation returns, the activity can invoke further operations on objects in the same extent without an access check. In effect, objects with access control lists act as controlled gateways into an extent. Authorization checking can be based on either the identity of the invoking user or of the source extent (effective user identifier).

Contexts are created for a given extent as required. The allocation of contexts to extents is determined by the *activation policy* for the extent. The default activation policy is that one context is allocated to each extent at each node at which objects of the extent are in use. Currently four different activation policies are supported,[2] as follows:

- **Distributed shared :** the default case. There is at most one context for each extent at each node. All objects of that extent which are in use at that node are mapped into that context.
- **Centralized shared :** the extent is represented by a single context (at one node). All objects of the extent that are in use are mapped into that context.
- **Distributed unshared :** each object of the extent that is in use is mapped into a different context. Contexts representing the same extent may be located on different nodes.
- **Centralized unshared :** as in the distributed unshared policy each object belonging to the extent that is in use is mapped into a different context, however all the contexts for a given extent are located on the same node.

For each of the distributed policies it is possible to specify a subset of the nodes in the system at which contexts can be created for the extent. For the centralized policies it is possible to specify the node at which the context(s) for the extent should be created.

The choice of activation policy for an extent is made by the owner of the extent and will typically depend on knowledge of the semantics of the objects belonging to the extent as well as local security policy. For example, a centralized policy may be used where an object is used to represent the interface to some device. An unshared policy may be used where an object represents a server providing a service to many clients. Local security policy may control the set of nodes available to a particular user.

4 Database support in Amadeus

One of the factors which determines the implementation techniques used within an object oriented programming language (OOPL) is whether or not it aims to support large volumes of data and large collections of objects, and in particular, whether or not it aims to support DBMS functions other than just storage and retrieval. If an OOPL is to support the implementation of a DBMS, or object level access to a database, then it must support access to volumes of objects in excess of the size of virtual memory, and it must efficiently handle the low locality of reference typical of these systems. In addition, it must manage collections of objects which can be queried, necessitating the maintenance of indices when objects are updated, and the acquisition of intent locks [1, 2] when objects are accessed. Since Amadeus aims to support a wide range of OOPLs, it must support such languages. The following are some of the ways in which the implementation of such languages differ from persistent OOPLs which access smaller, non-database, persistent stores:

- Where the locality of reference is high, there will be no significant cost in using logical *oids* – *oids* for which a lookup of a disk based table is required in order to locate and map in the target objects. The low locality of reference typical of database applications makes the use of physical *oids* attractive – allowing the page number of an object, or a cluster of objects, to be built into an *oid*, as a hint. Amadeus will support both mechanisms, allowing the

[2] We are currently investigating a replicated policy.

language level to choose whichever is more appropriate. The main component of the solution is a mechanism for preventing low level applications from fabricating physical *oids* in order to access other users' objects.

- To support access to large volumes of data, clusters of objects must be easily unmapped from virtual memory. This is difficult where a persistent OOPL swizzles its references into virtual memory pointers – since the unmapping of a cluster then requires a scan of the context to determine if any object is referencing an object in the cluster. This is very expensive and it mitigates against access to more data than will fit into the relatively small (by database standards) virtual memory size of a process. The Orion DBMS [3] swizzles each reference into a pointer to an indirection structure, allowing the target objects themselves to be easily unmapped. The E programming language [4] does not swizzle references, thereby allowing efficient unmapping of objects, and less overhead associated with the mapping in of an object. On the other hand, the swizzling of references to virtual memory pointers has efficiency advantages where virtual memory is rarely exhausted. Therefore, Amadeus provides the system level support needed to handle a wide range of swizzling policies within its OOPLs.

- To maintain indices and intent locks on collections of objects, operation invocations must be visible to a component of the system responsible for DBMS functions. Amadeus allows this to be done without changing the normal representation of an object – by using a code image which communicates with a *controller* object at the start and/or end of each operation, in addition to implementing the object's normal behaviour. This is optional for any object, so non-database objects are not affected, and different controllers can be provided by different DBMSs, rather than providing a fixed way to handle these important DBMS functions.

- For efficiency, the members of some DBMS collections should be stored by the collections themselves – so that they can be clustered, and possibly vertically and horizontally fragmented, to improve the efficiency of queries or scans. This cannot be done by the Amadeus storage system, since it must remain independent of application semantics, and different DBMSs may require different mechanisms. Instead, members of these collections are stored in such a way that they can only be accessed through an *access object* associated with their collection, and this access object can control and use specialised storage mechanisms. This is done transparently to the invoking object – without affecting its operation invocation mechanism. It is also transparent to the OOPL, since it is handled by the generic layers within Amadeus.

The last two points support alternative mechanisms for implementing a collection: either it can hold references to its members, or it can act as a *storage container collection*.

5 Exploiting parallelism in Amadeus

This section looks at how load balancing is incorporated into Amadeus. As a concrete example it shows how a program to perform ray tracing was structured to run on Amadeus. Before going on to discuss the details of load balancing, it must be noted that our system does not (currently) support the replication of objects in memory, nor does it allow objects to be migrated from one node to another once they are mapped into memory.

As a first and obvious step it can be seen that the conventional technique of load balancing on process creation used in many systems, maps onto load balancing on activity creation in the case of Amadeus.

If all of the objects used by an activity are mapped at the node, to which the activity was assigned, then straight forward load balancing on activity creation will be effective. However, in the case of activities which share objects with other activities, extra steps are required. Take for example an application which is structured as a number of co-operating activities. For the most part, the activities operate on private objects but occasionally, for example for synchronisation purposes, they need to invoke on other objects. In the absence of any other action by the application, all of its objects will be assigned to the same cluster. Thus, no matter what placement decisions it makes about the activities it creates, these will effectively all return to the parent node to execute – thus eliminating any possibility of performance improvement due to parallel execution. Accordingly, it is necessary for the application to explicitly partition the data (objects) it uses during its execution, as well as functionally partitioning its activities.

The solution adopted in Amadeus is for the application programmer to explicitly use cluster management primitives[3] to partition the objects into clusters. Load balancing assigns a (`preferred`) node to each activity when it is created. When the activity subsequently faults objects (clusters) into memory they are placed at the preferred node. When an activity invokes on objects that are already mapped, these objects are not moved, but instead the activity diffuses to the appropriate node.

Thus Amadeus assigns activities to nodes uses load balancing and puts the responsibility for partitioning the problem on the application programmer. The following section shows how a substantial real world problem is structured to run on Amadeus.

5.1 An Example Application

Ray tracing is essentially an exercise in data parallelism with the program being structured along classic master slave lines. The image can be broken up into a number of subsections (rectangles) each of which can be processed in parallel (by a slave worker) without any reference to its neighbours. Once a worker completes its rectangle, this can be sent back to the master to be stored in file – or as was done in our case displayed, directly on a colour graphics screen.

```
Initialize;
FOR I = 1 to NumberOfWorkers
  NewCluster();
  SetDefaultCluster();
  Create worker I and associated objects;
END

FOR I := 1 to NumberOfWorkers
  launch worker I;
END

Display results;
```

As workers activities are created, load balancing will spread them around the system. The clustering of objects will ensure that activities do not have to diffuse too often.

[3]Namely, NewCluster() and SetDefaultCluster()

6 Summary and Conclusions

In a short paper such as this it is impossible to cover the central ideas and their ramification in any great detail. However, it can be seen even from this brief overview that the Amadeus philosophy of aiming to be a general purpose object oriented platform is a fruitful area of research, and one which shows some of the strengths and generality of the object oriented approach.

References

[1] J.N. Gray, R.A. Lorie, G.R. Putzolu, and I.L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *Proc. IFIP Working Conf. on Modelling of Data Bases Management Systems*, pages 695–723, Freudensadt, Germany, January 1976.

[2] Gray J.N. Notes on database operating systems. In *Operating systems an advanced course - LNCS vol. 60*, chapter 3, pages 393–481. Springer-Verlag, 1978.

[3] Won Kim. *Introduction to Object-Oriented Databases.* Computer Systems. MIT Press, 1990.

[4] J.E. Richardson and M.J. Carey. Persistence in the E language: issues and implementation. *Software - Practice and Experience*, 19(12):1115–1150, December 1989.

[5] R.Kroger et al. The RelaX transactional object management system. In *Proceedings International Workshop on Computer Architectures to support Security and Persistence of Information.* Springer-Verlag, May 1990.

[6] R.Schumann et al. Recovery management in the RelaX distributed transaction layer. In *Proceedings* 8^{th} *Symposium on Reliable Distributed Systems*, pages 21–28. IEEE, October 1989.