Check for updates

System support for shared objects

P.-Y. Chevalier, D. Hagimont, S. Krakowiak, X. Rousset de Pina

Bull-IMAG, 2 av. de Vignate, 38610 Gières, France

{chevalier, hagimont, krakowiak, rousset}@imag.fr

Abstract. This position paper supports the view that a model based on shared objects is an attractive alternative to message passing for structuring distributed applications and that a distributed operating system should provide support for shared persistent objects. The paper reviews several recently proposed solutions, including our own, and outlines some current problems.

1. Motivation

Shared objects appear to be an attractive paradigm for communication in distributed systems. In this paper, the term "object" generally refers to a structuring unit which associates a piece of data and a set of access methods. The methods are the only means of access to the object. More specific definitions will be supplied when necessary.

There seems to be an interesting evolution in the definition of paradigms for distributed systems. In a first phase, "centralized abstractions" of a decentralized reality have been defined. For example, logical clocks provide a total ordering associated with a single (virtual) clock; location transparency aims to define a single uniform image of a distributed system. In a latter phase, there is an attempt to relax the constraints of a centralized model, which may be too strict for some applications while entailing unnecessary costs. For example, causal ordering has been recently proposed as an alternative to total ordering.

Communication through shared data is another example of a "centralized" paradigm for interprocess communication. This paradigm has materialized in two forms: distributed virtual memory and distributed shared objects. Distributed virtual memory is provided at the system level and is essentially the equivalent of a shared memory; it does not impose any structure to the information that it contains. On the other hand, distributed objects have a structure, which is usually (but not necessarily) embodied in a programming language.

It should also be noted that files have been another widely used means of interprocess communication, even in centralized systems (measurements show a high proportion of short-lived files, which are typically used for communication). The use of files for communication actually seems as least as important as their use for long term storage of information. Making shared objects potentially *persistent* is a means of reconciling the two functions of communication and storage in a single construction, thus avoiding the need of explicit storage and retrieval. A persistent object is one whose lifetime does not depend on that of the programs or processes that use it.

We may thus define the shared object paradigm in the following general terms: an application is composed of a set of objects, which may be predefined or created dynamically; objects are accessed through specific operations, and any object may be shared by several processes; depending on the model, processes may be associated with objects, or defined independently; the objects may or not be persistent; the objects are distributed on the different

machines (the distribution is usually transparent, but explicit location indications may exist). For the application programmer, the view of the system is that of a single integrated environment, as opposed to that of a set of components explicitly connected through communication primitives.

In section 2, we discuss some approaches to implementing different forms of the shared object paradigm, based on recent work. In section 3, we describe some of our experience and ongoing work in this area. In section 4, we outline some current problems.

2. Implementing the shared object paradigm

In this section, we attempt to characterize the main solutions for the support of shared objects in a distributed system, in order to identify the current research issues; a comprehensive review is well beyond the scope of this paper.

We are interested in a system in which there is a single, network wide, object name space spanning machine boundaries. We may identify two main approaches for object support.

In the first approach (used, for example by Clouds [Dasgupta 91] , Orca [Bal 90], Emerald [Jul 88]), individual objects are directly supported by the operating system (as in Clouds) or the run-time system (as in Orca or Emerald). The system provides a location services that allows to find an object when a call is made to one of its methods.

The second approach relies on distributed virtual memory (as implemented, e.g. in [Li 89]). This approach has not yet been widely experimented with for supporting distributed objects. However, it seems promising, because it separates the low-level memory management issues, which are relevant to the operating system, from the management of objects, which is done in the run-time system and may be language-specific. This separation was a primary motivation for the introduction of virtual memory in centralized systems.

An ambitious design is that of a single level distributed virtual memory, in which all information will be potentially persistent. This memory should therefore be garbage-collected, and should also provide some support for fault tolerance. Such a memory might eventually be supported by the future wide-address processors.

Whether the objects are directly managed by the system or supported by a distributed memory, the single most important primitive is object invocation, and it is essential to make it efficient. The whole range of possible solutions has been experimented: remotely calling an object, moving it to the calling site (this is the Emerald "call by move") or replicating objects, in order to have only local calls, with an underlying consistency-preserving mechanism. this mechanism could be part of the run-time system (like in Orca) or be part of the service provided by a distributed virtual memory. The optimal solution is not clear at the present time.

3. Experience and ongoing work with shared persistent objects in Guide

The paradigm of persistent shared objects was a central one in the Guide project. Guide is a distributed operating system that is designed to support applications written in an object-oriented language. Objects are passive and may be shared by a number of activities; activities may spread out on several nodes. The choice of passive objects was motivated by practical considerations, since we expected applications to need a large number of relatively small objects.

The first prototype version of the system [Balter 91] has been developed on top of Unix; it supports our own language [Krakowiak 90]. Several prototype applications have been

programmed, in the area of document circulation and cooperative editing of structured documents.

The experience of using persistent shared objects may be summarized as follows.

• On the positive side, using shared objects allowed a programmer to develop an application as centralized and then port it on a distributed environment without significant additional work. Persistent objects also vastly simplified information storage by removing the need for explicit storage and retrieval.

• On the negative side, the system was essentially designed to support the run-time system of a single language, which limited its flexibility (even if existing applications could be easily reused by encapsulating them into Guide objects). There was no means for the programmer to define "lightweight", non-persistent objects, which limited the ability to use very small objects.

Based on this experience, we are now developing a second version of the system and its programming environment. This new version, whose design has been completed [Freyssinet 91], should provide generic support for object-oriented languages that satisfy the following set of minimal assumptions (as seen from the operating system):

- The language is class-based; a class describes the methods applicable to its instances and the internal organization of the instances; there is an explicit link from an instance to its class.
- Classes are organized in a hierarchy by the *is-a-subclass-of* relationship; there is no assumption about the organization of the inheritance graph (e.g. single or multiple, etc).
- Objects are named by universal references (i.e. references which are independent from any addressing context); this allows persistent objects to be defined.

The system has been designed as a hierarchy of abstract machines; for the purpose of this presentation, we are only concerned with two of those: the *segment machine*, which provides a segmented persistent memory; the *object machine*, which uses segments to build objects.

The segment machine uses indirect method calls in order to allow dynamic binding of the code of the methods, as required by objects models. The mechanism must support persistence, sharing, and distribution. Therefore, we must uniformly resolve references from different segments to a shared segment, which may simultaneously be mapped in several contexts (i.e. virtual address spaces), possibly on different machines, and at different addresses.

The proposed solution relies on a linkage section (à la Multics). This section is created each time a segment is mapped in a context, and it contains all context-dependent, non-persistent informations associated to the segment, including its mapping address. When the segment is unmapped or deleted, its linkage section is deleted. The main cost is incurred at binding time, when the segment is mapped in a context. This occurs at the time of the first call on a method of the object in the current context; subsequent accesses to the mapped segment are not interpreted, and use indirection through the linkage section. The additional cost is one level of indirection.

This scheme has the additional advantage that the internal representation of a persistent segment does not depend on whether it is mapped or not, thus avoiding the "pointer swizzling" costs incurred if references to other segments must be changed. References are universal (i.e. system wide) internal identifiers. They are interpreted by the system at binding time (i.e. at first call). The system locates the segment using its reference; the segment is loaded from the storage system if not mapped in the current context. The segment may already be mapped in another context, in which case the execution policy determines whether it should be remotely called or

remapped. Non-persistent segments, which are only used for communication, may also be defined.

With the advent of 64-bit wide addresses, the relocation problem is likely to disappear, since the machines in a local area network share a single address space. The identification of an object will now contain its (immutable) address, allocated at binding time.

4. CURRENT PROBLEMS

Shared objects provide a seemingly attractive paradigm for programming distributed applications. Efficiently supporting shared objects in a distributed system is still an active research area. We briefly summarize what we believe are the main current issues:

Consistency vs performance. Replication is a way of improving object call performance. However, consistency of the replicas must be maintained. A possible way of investigation is to define "degrees of consistency" which would allow to trade "freshness of information" against performance. The work on released consistency in distributed virtual memory is relevant at that point.

Generic support. While object management may be optimized for a specific language, a system must support several languages. There are two main questions. What is the degree of generic support for objects that may be provided? How are the services split between kernel and generic run-time?

Garbage collection. Garbage collection is an essential service for object support, especially if most object are potentially persistent. Should it be left to (generic) run-time support or integrated at a lower level?

Synchronization. Shared objects must enforce synchronization constraints on the concurrent activities that use them. The expression and enforcement of such constraints is still an open question, especially in relation with other language features such as inheritance.

Fault tolerance. Should transaction management be part of the basic services of a shared objects system? In that case, how can transactional mechanism coexist with synchronization mechanisms?

Acknowledgments.

Andrzej Duda, André Freyssinet, Serge Lacourte, Michel Riveill and Miguel Santana have contributed to the design of the Guide object support system. This work was partly supported by the Commission of European Communities under the Comandos ESPRIT project (nr 2071).

5. REFERENCES

[Bal 90]
H. Bal, M. F. Kaashoek, A. S. Tanenbaum, Experience with distributed programming in Orca, *Int. Conf. on Computer Languages*, 1990

[Balter 91]
R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. Le Dot, H. Nguyen Van, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville, G. Vandôme, Architecture and implementation of Guide, an object-oriented distributed system, *Computing Systems*, vol. 4, 1, pp. 31-68

[Dasgupta 91]
P. Dasgupta, R. J. LeBlanc Jr, M. Ahamad, and U. Ramachandran, The Clouds distributed operating system, *IEEE Computer*, 24,11 (nov. 1991), pp 34-44

[Jul 88]
E. Jul, H. Levy, N. Hutchinson, A. Black, Fine-grained mobility in the Emerald system, *ACM Trans. on Computer Systems*, 6, 1 (1988), pp. 109-133

[Krakowiak 90]
S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin, X. Rousset de Pina, Design and implementation of an object-oriented, strongly typed language for distributed applications, *Journal of Object-Oriented Programming*, 3,3, (sept.- oct. 1990), pp. 11-22

[Levy 91]
H. M. Levy and E. D. Tempero, Modules, objects and distributed programming: issues in RPC and remote object invocations, *Software — Practice and Experience*, 21, 1 (jan. 1991), pp. 77-90

[Li 89]
K. Li and P. Hudak, Memory coherence in shared virual memory systems, *ACM Trans. on Computer Systems*, 7,4 (1989), pp. 321-359

[Organick 72]
E.I. Organick, The Multics system: an examination of its structure, *MIT Press*, 1972

[Freyssinet 91]
A. Freyssinet, S. Krakowiak, S. Lacourte, A generic object-oriented virtual machine, *Proc. Int. Workshop on Object-Orientation in Operating Systems*, Palo Alto, october 1991, pp. 73-77

2 Comparison by Issue

Of the many issues that arise in distributed systems, five have been selected for consideration. This list is not intended to be exhaustive, but does provide examples of both the most interesting differences between the two models and some shared features.

2.1 Location Transparency

Both of these models exhibit location transparency. That is, in each case the references can be manipulated and invocations made without knowledge of whether the indicated object is local or remote. The assumption must be, however, that all invocations are remote (rather than all being local), as otherwise the additional failure modes introduced by network problems and failed remote nodes cannot be adequately handled [11]. In the UTOR model this means that additional exceptional returns may occur. Treating all invocations as potentially remote can lead to inefficiencies, since large amounts of unnecessary error handling code may be produced. It follows that optimisations which ensure that calls are local have benefits in terms of program simplification as well as run-time efficiency. In the U-TOR model special support, such as the Bellerophon colocator references [7], is required to achieve this.

The fragmented object model permits the designer of an FO class to handle such problems within the fragment, however this will not *always* be possible. If network errors cannot be entirely masked, the FO designer must include the additional exceptions in the interface of the object, but no special support mechanisms are otherwise required. So, in general, both models require that application programmers consider the possibility of failures due to distribution. However, the FO model ensures that such problems are made explicit and significantly restricts the circumstances under which such care is required of the application programmer.

Only if the reference mechanism offers network fault transparency, that is if the entire system exhibits fail-stop semantics, can this problem be disregarded. Such a system is, however, better characterised as a parallel system, a multi-processor, rather than being viewed as a distributed system or multi-computer.

2.2 Replication Transparency

The *raison d'être* of the FO model is to support distributed shared objects. One of the most valuable forms this can take is that of the replicated object. Both fully replicated objects and those which are perceived as replicated, but with weakened consistency models, can be implemented using fragmented objects.

Replication is handled internally by the fragments of an FO and the associated hidden components, such as the connective objects. A variety of consistency and sharing strategies can be implemented in this way, without visible consequence in the way in which the object is used. Recall that the model presented to the client code is always that of a local object which is shared with other remote clients.

Unfortunately the same cannot be said for the U-TOR model. The essence of that model is of a reference to a single object at a single location. It is, of course, possible to build a replication mechanism over such references, either by working with sets of references or by using references to group managers. One such attempt was, however, the initial motivation for the work on fragmented objects, and thus indicates the added power of the FO approach. The UTOR model is not aimed at support for replication.

2.3 Persistence

A superficial encapsulation of persistence can be achieved in the UTOR model if an object store manager intercepts messages sent through references to persistent objects. In a simple approach this manager could transfer the indicated object into memory before delivering the message. In itself, however, such an approach is inadequate as it fails to address the issue of checkpointing. Care must be taken to ensure that the object really is persistent, that is that it maintains consistency across processor failure and recovery. In particular, some form of transaction mechanism is essential. This should be integrated with the uniform references, superseding or supplementing the message passing or invocation mechanism.

Managing persistent objects with the FO model is somewhat cleaner. Rather than relying on the availability of a transaction mechanism this can, if necessary, be provided by a connective object. That is, the transaction mechanism can be layered over the underlying communication primitives without exposing it to the client. A persistent fragmented object will usually include the stable image of the object as an additional fragment, in conjunction with one or more memory resident copies. Within the FO the required consistency properties can be maintained, with checkpointing of changes and any necessary logging. This issue is closely related to replication as the reference to a persistent object is, in effect, a reference to a group of replicas: one or more on disk and others in memory. The BOAR library [21], now being developed, includes support classes for general persistent objects.

2.4 Garbage Collection

Distributed garbage collection is a difficult problem, since all known algorithms that can collect cycles are either not fault tolerant [13,1], can be slow to recover garbage [17] or are not scalable [25]. Supplementing such algorithms with an efficient fault-tolerant mechanism that collects all, or almost all, acyclic garbage [24, 6] offers significant advantages in reducing the work required of these complete techniques. Such benefits rely, however, on distributed cycles being relatively rare. Alternative approaches, such as [5,14], are more appropriate if small cycles are common and predictable.