[ROZIER et al.-89] Marc Rozier, Vadim Abrossimov, Francois Armand, Ivan Boule, Michel Gien, C. Kaiser, Sylvain Langlois, Pierre Leonard, Will Neuhauser
The Chorus Distributed Operating System
Computing Systems, 1988

[SHAPIRO-86] March Shapiro
Structure and Encapsulation in Distributed Systems: the Proxy Principle
Proceedings of the 6th ICDS Conference, May 86

[SHAPIRO et al.-89] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, and C. Valot
SOS: An Object-Oriented Operating System - Assessment and Perspectives
Computing Systems, Vol 2, No4, Fall 1989

7 Acknowledgments

We would like to thank Marc Guillemont for his helpfull comments on this paper.

References

[BALTER et al.-91] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A, Freyssinet, S, Krakoviak, 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, Vol4, No1, Winter 1991]

[CAHILL et al.-91] V. Cahill, C. Horn, G. Starovic, R. Lea, P. Sousa
Supporting Object Oriented Languages on the Comandos Platform
Proc. of ESPRIT'91 Conference, Brussels, Belgium, November 25-29, 1991

[CHASE et al.-89] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield
The Amber System: Parallel Programming on a Network of Multiprocessors
ACM SIGOPS, Litchfield Park, AZ, December 1989

[DASGUPTA et al.91] P. Dasgupta, R. Ananthanarayanan, S. Menon, A. Mohindra, R. Chen
Distributed Programming with objects and Threads in the Clouds System
Computing Systems, Vol 4, No 3, Summer 1991, USENIX Association

[ISA] The Integrated Systems Architecture project ISA
Esprit Project 2267, The ISA Consortium, Castle Park, Cambridge, UK

[Habert et al.-90] Sabine Habert, Laurance Mosseri, Vadim Abrossimov
COOL - Operating System Support for Object Oriented Systems
Proceedings of ECOOP/OOPSLA'90 Conference, volume 25 of SIGPLAN Notices, Ottawa, Canada 1990 ACM

This operation, translates into a *COOL2-base* operation to unmap the cluster containing the object and to map it into another context. When the cluster is mapped into a new context, all internal references are updated depending on the new location. This technique, often know as pointer swizzeling (address translation) is common in database operations.

5.3 Migration from disk into a context

A final possibility is mapping an object that resides in a cluster which is currently inactive and which *COOL2-base* has moved to secondary storage. In this case the cluster is stored in an address space neutral format until it is invoked again. Invocation implies allocating a virtual memory address space and relocating the cluster the object if needed. This activation is done after an object fault, hence it is transparent to the application. This basic mechanism supports the single level persistent store that the *generic runTime* provides.

6 Concluding remarks and status

We presented briefly the *COOL2* architecture that implements a model of cooperation between contexts to allow the use of virtual memory identifiers that are kept valid on different sites of a network. It supports single address space semantics between groups of contexts.

This model has an impact on object migration and on the transparency at programming level of object use over different sites.

Our current *COOL2* prototype implements the described functionality and runs above the CHORUS micro-kernel on a network of IAPX386 based machines. We are currently experimenting with its use to refine the base mechanisms.

We believe the benefit in our work is not in its novel nature, but in the combining of several techniques, distributed virtual memory, proxies, and a single level store to provide a base set of mechanisms that can be use to support multiple object oriented languages. Further, we believe that only by providing these services in a coherent form, and at a low enough level in the operating system can they be both useful and efficient.

these cases we resort to the use of local proxies[6] for remote object. For example, we would create a local proxy for object D and place it in cluster 1 in context 2. Any invocation from object C to object D would use a virtual memory pointer to a local proxy that would invoke the remote object through standard IPC.

5 Object and activity migration

In *COOL2* the local visibility of objects is not exclusive to one context and it may grow dynamically with time spanning through different contexts, possibly in several sites. Objects may also be passive on disk. Objects move between the disk and contexts with the piece of memory that supports it (mapping and unmapping). We have to be aware that each time objects are mapped to different addresses they have to be relocated which can be time consuming.

The only reason to make an object locally visible to a context, if it is already visible to another, it is for performance enhancement because invocations will then be simple function calls (instead of remote calls). So, if an activity in a context tries to invoke some object, the object is mapped locally if possible.

Objects do not, by default, migrate explicitly, nor is the application aware of object migrations. If objects can not migrate locally to be invoked, the activity migrates instead.

We identify three forms of migration that COOL-base supports:

5.1 Object migration inside a context group

As described in section 3, when possible, a reference to a remote object cause that object to be mapped into the local address space using a distributed virtual memory model. Inside a context group we are sure that the target context has free space for the object exactly at the same address as the source context.

5.2 Object migration between context groups

If an object migrates between contexts that are not part of the same context group, it is unmapped from the first, mapped by the second and relocated.

[6] A proxy is an object representative that performs a Remote Procedure Call to forward object invocations; see [SHAPIRO-86] for details.

coherency mechanism has to exist for the set contexts[5] that use the same set of objects, i.e., that share memory. This memory has to be the same and mapped also at the same addresses.

So, we defined two more abstractions that allow the system to control both the set of objects being shared by a group of contexts and the group of contexts itself: these are *clusters* and *context groups*.

Clusters are sparse chunks of persistent memory. They hold groups of related objects that can be potentially shared and turned persistent at the same time. *Clusters* are used by *COOL2* both to control persistent memory (i.e. objects) sharing between contexts, and to group persistent memory in arbitrary sized chunks.

Context groups are a *super-context* that groups one or more contexts on one or more sites into a single entity. All contexts of the group share the same address space. Applications that use the same persistent memory are automatically inserted by the system in the same *context group*.

The main benefit of clusters and context groups is that inside a context group, the objects part of a cluster are always assured the same addresses because the address space is the same for this set of contexts. Thus, using figure 1 we can see that objects A, B and C residing in cluster 1, all belong to context group 1. At this particular time, objects A and B reside on site 1 and object C resides on site 2. A reference in object B to object C would use a virtual memory pointer that reffered to a location in context 1. Any attempt to de-reference this pointer would cause the underlying *COOL2-base* to map object C into context 1 using the distributed virtual memory model.

4 Local identifiers are global identifiers

As shown above we are using the distributed shared memory model to support the use of local virtual memory pointers within a group of clusters. There are however, circumstances where this model will not, or should not work. Again reffering to figure one, an attempt by object C to invoke object D, which is not part of the context group 1 would, under normal circumstances cause context 4 to become a member of context group 1. However, we see that part of the cluster 2 overlaps addresses valid in cluster 1. A merge of these two clusters would cause a overwrite of part of the address space. In such a case, using shared virtual memory would not work. In

[5] Amber does the same for all contexts of a distributed application.

The *COOL2-base* supports four further abstractions that abstracts the distributed programming model. These are *contexts*, *sites*, *context groups* and *clusters*.

Contexts are a collection of resources and provide a linear and contiguous address space for applications. Objects ocupy a portion of the context's address space.

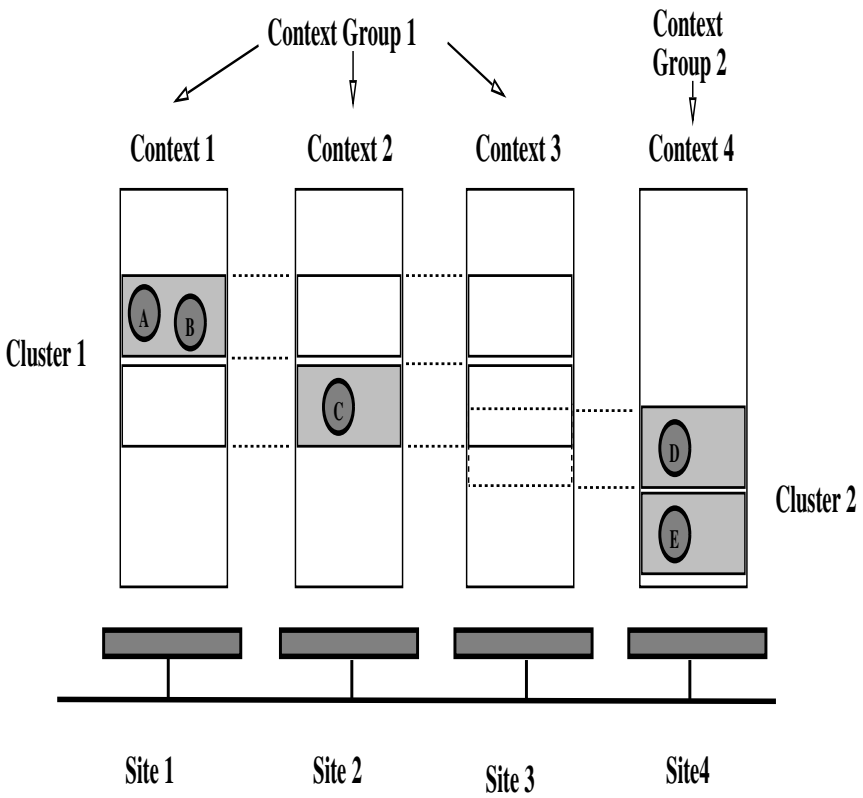

Figure 1: The distributed programming model of COOL-2

A *COOL2* objective was to identify objects in a location transparent manner, so we experimented the use of direct virtual memory pointers to identify both local and remote objects. *COOL2* relies on shared memory among different contexts in different sites. Shared memory provides a single address space semantics for a distributed application. To achieve this, a

a distributed virtual machine given by the *COOL2* system.

The *COOL2* system works on a network of sites. Each site is a set of tightly coupled processors. A *COOL2* operating system is composed of a set of *COOL2* kernels, each running on each site.

COOL2 explores base operating system mechanisms to support multiple object oriented models and its architecture reflects this with an intermediate generic interface. *COOL2* has three levels of interface:

- a language specific run-time interface, composed the run-time code generated by the *COOL2* pre-processor; this level is language dependent and uses generic underlying mechanisms;
- a *generic runTime* interface which provides the basic abstractions: objects, invocations and activities and is implemented as library code linked with the application; this interface level is intended to support multiple object oriented programming systems;
- an operating system interface: *COOL2-base* , with system calls that provide a base set of mechanisms that support the *generic runTime* in a distributed environment;
- and a microkernel: CHORUS that provides a minimal base for building distributed operating systems; in can be used concurrently by other independent subsystems, e.g., the CHORUS/MIX UNIX subsystem; this enable the *COOL2* subsystem, as seen by the CHORUS microkernel, to cohabit with a UNIX environment that provide I/O, although this is used simply to avoid re-writing device drivers and other low level code.

3 The programming model

COOL2 has developed out of our work with *COOL1* , COMANDOS [CAHILL et al.-91] and the ISA project[ISA]. From the programming model point of view, *COOL2* explores the model of a centralized computing system on a distributed environment.

The *COOL2* main programming entities are objects and activities. Objects are passive and represent persistent memory for programs and activities abstract execution. These two entities are orthogonal. Like CLOUDS [DASGUPTA et al.91] an activity may span several objects and an object may have several activities executing concurrently as invocations.

distributed systems. *COOL2* explores the use of local object identifiers at language level, hiding distribution at system level.

In other systems (e.g. our previous work in *COOL1* [Habert et al.-90][4]) objects are addressed differently in the local and the remote cases which means that the application programmer is presented with a two tiered naming model with both local and global identifiers. Also, in systems like Guide[BALTER et al.-91], ANSA[ISA] and SOS[SHAPIRO et al.-89], all objects have global identifiers. In this respect, our work follows the ideas of Amber [CHASE et al.-89] which extends local object identifiers to the network. Although, unlike Amber, *COOL2* uses the shared memory model to maintain coherency between shared objects in a distributed environment.

Our reasons for this are threefold: to maintain the traditional, and familiar programming model in a distributed system; to provide a network transparent invocation model; and to compare the performance penalties of a solution that completely avoids global identifiers in the local case.

We also wanted to study the impact of a single level object identification scheme on object migration both from the application and the the system point of view. An application that always uses local object identifiers, disregarding object location, will not have to be aware of migrations, i.e., these migrations are transparent.

COOL2 implements a persistent object oriented programming system. The application does not need to access a second level store like a file system. Objects are automatically persistent and remain present in persistent virtual memory until deleted.

This paper will overview our architectural model and then explore virtual memory and the object identification support to achieve single address space semantics in an object oriented distributed environment. We also present the generic *COOL2* architecture and explore the effects of the above functionality in object migration.

2 *COOL2* architecture

Applications are generated with *COOL2* programming environment. In order to interface to the system, they are parsed by a *COOL2* preprocessor and through the standard C++ compilation chain. Then, they execute on

[4] *COOL1* was a joint project between Chorus Systèmes , SEPT (Service d'Etudes communes de la Poste et de France Télécom) and INRIA (Institut National de Recherche an Informatique et en Automatique)

Transparent object migration in COOL-2[1]
Position Paper

Paulo Amaral[2]
Chorus Systèmes and Université Pierre et Marie Curie (Paris VI)

Christian Jacquemot, Peter Jensen, Rodger Lea, Adam Mirowski
Chorus Systèmes

Abstract

COOL2 is a distributed object oriented computing system which extends the traditional single address space programming model to a distributed environment. *COOL2* sits on the top of the CHORUS[3] microkernel and can be used on a local network of workstations. The current *COOL2* implementation supports a standard C++ 2.0 programming interface.

The C++ programmer is provided an object oriented distributed computing system which looks centralized from the outside. *COOL2* objects are manipulated by the application always through their local identifiers even if they are not local. Object distribution is transparent, that is, objects are accessed uniformly irrespective of their location. Objects can also migrate transparently.

We focus on the distributed transparent use of standard C++ objects in *COOL2* , explaining why we kept the traditional local object identification with virtual memory pointers and we present the base architecture of *COOL2* that allows us to achieve this.

1 Motivation

Our motivation when designing *COOL2* was to investigate base abstractions for a distributed environment that would support multiple object oriented programming systems. We wanted to understand the functional layers of such a system and work towards a generic model. Besides, we wanted to investigate how to provide single address space semantics in distributed environments using the techniques of distributed virtual memory and single level object identification.

The *COOL2* programming model offers the traditional programming style of a centralized computing system that works transparently on a distributed architecture. We focused the problem of object identification on

[1] *COOL2* The second version of the CHORUS Object Oriented Layer

[2] email: paulo@chorus.fr

[3] CHORUS is a registered trademark of Chorus Systèmes