CS/TR-91-102

Supporting object oriented languages in a distributed environment: The COOL approach

Rodger Lea (Chorus systèmes)
James Weightman (University of Lancaster)

approved by:

abstract: In: Proc. of TOOLS USA'91, Santa Barbara, July 1991

Contents

1 Abstract

Object oriented computing is now an established technology for software development. However, a number of challenges must be met before the topic can claim to be fully mature. One of the most demanding challenges is posed by the move from single workstation environments to the more general case of a distributed system.

To date, much of the work on distribution has been carried out in an ad-hoc manner, by bolting on distribution to existing language systems. Experience with the development of distributed operating systems has shown that this approach will not work. Instead, a more comprehensive approach whereby the system fully supports distribution at a level below the programming language is required.

We feel that the most efficient approach is to provide a distributed platform that supports a generic object model allowing existing object oriented languages to be layered above.

This approach retains the clean semantics of the language model yet extends their functionality across a distributed persistent environment. Additionally, it allows us to support multiple languages simultaneously.

We detail the development of the Chorus Object Oriented Layer (COOL), built as an extension to the Chorus micro-kernel, which provides a distributed platform for a number of language object models. This platform provides the facilities to support language objects with persistence, storage, remote communication and migration whilst retaining the existing language model.

We discuss the implementation of two programming models onto the COOL generic interface, that of C++; a fine grained non distributed model, and of the large grained ANSA distributed programming language. In particular we highlight the benefits and drawbacks of our approach and propose future directions of research.

2 Introduction

The COOL[1] system has been designed to satisfy two goals; to provide an efficient object oriented support layer built directly into the Chorus Micro-Kernel and to provide a generic object support platform that supports multiple object oriented languages.

Experience to date, both within the distributed systems community and within the object oriented programming community has concentrated on supporting distribution in one of three ways:

- By extending existing object oriented languages with support for distribution. This for example is seen in such work as the extensions to SmallTalk [Bennett 87] [McCullough 87] and to C++ [Shapiro 86].
- The second approach has been the development of distributed object oriented languages closely coupled with the system. A classic example of this is the work carried out by the Emerald project [Black 87].

[1] The Chorus Object Oriented Layer (COOL) is a joint project between Chorus systèmes, SEPT and INRIA.

- The last approach is the development of an underlying support environment that attempts to provide a distributed support platform for languages. Similar work is reported in [Lucco 90]; however, the tarmac system is concerned solely with virtual memory units, a lower level of semantics than COOL which deals with objects.

In general these approach suffer from a number of drawbacks, extending an existing language with support for distribution often is carried out in an ad-hoc manner, and more importantly, may change the fundamantal semantics of the language. In addition, the work carried out is specific to the language in question and often not re-usable.

The development of new languages that implicitly handle distribution is a more desirable approach but suffer from the drawback of a limited user community and poor support.

The most promising approach is to build a support substrate that supports an object oriented language in a distributed environment, however, it is necessary that this support is generic enough to support multiple languages, while at the same time being able to support the programming paradigm of the languages it intends to support.

This paper describes our experiences with the development of COOL. An extension to th Chorus Micro-kernel, that supports a generic notion of objects, their storage and their interactions in a distributed environment.

To demonstrate the feasibility of this approach and to gain experience of our abstractions, we have built a C++ environment on the COOL sub-system that enables programmers to develop applications consisting of standard C++ objects that can then be used in a transparent manner in a distributed environment. This has been carried out in conjunction with SEPT[2] as part of the CIDRE project and is reported in [Deshayes 89]

From this work we outline a number of drawbacks of the COOL model and present our future directions of research.

3 The COOL architecture

COOL can be seen as two distinct layers; the COOL base mechanisms which collectively provide a set of object support abstractions; and the COOL run-time system which specialises the system interface to support a specific object model. Our objective is to provide a set of base abstractions that can be used to support a number of different object based systems. System developers use the COOL base to build their own tailored run time, supporting their particular object model. Onto this, language models are mapped and then used by application developers.

This model is represented graphically in figure 1.

The COOL base level is built of four functional components each mapped onto the sub-system actor model of Chorus; the COOL *sub-system* object which provides the COOL system interface, the COOL *class manager* that is responsible for managing class representations, and using these to instanctiate COOL objects. The COOL *mapper* is responsible for managing the mapping between object representation in virtual memory, and the secondary storage representation.

[2]Service d'etudes Communes des Postes et Télécommunications

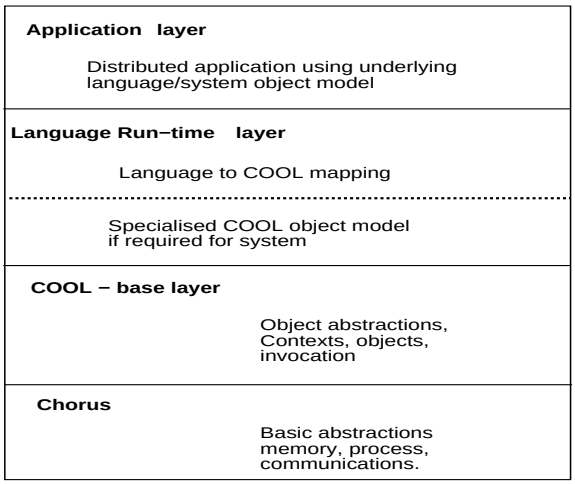

Figure 1: The COOL architecture

Lastly, the COOL *activity monitor* is used to manage communications directed to objects that are in transit.

These sub-system actors extend the Chorus micro-kernel interface to that of an object oriented micro-kernel. This work is carried out in the micro-kernel because of its manipulation of protected portions of a running programs address space. However, the language specific run-time is developed as a library linked in with normal application code and providing the interface to the COOL kernel.

4 The COOL abstractions

The COOL system is a set of abstractions that are designed to enhance those provided by the Chorus Nucleus [Rozier 88]. COOL exists as a set of system actors[3] that provide an object oriented interface above the Chorus Nucleus. The base abstractions that COOL offers are the notion of the **Context**, which provides an address space in which objects exist and models a traditional address space; the notion of an **Object** which is a system supported generic entity offering higher level object abstractions a target to map their object abstraction to; a **Message based** communication model that enables objects to transparently invoke other objects both locally and remotely. The message system is further enhanced to allow **Migration** of referenced objects between contexts; and finally a set of mechanism to support both object and context **Persistence**.

[3]Chorus supports a system model whereby traditional kernel functionality is encapsulated in 'actors' which communicate via the Chorus IPC mechanism and collectively provide the system interface.

5 The COOL system interface

The COOL system and its externals have already been described in the literature [Habert 90] [Lea 91], below we give a brief summary of the COOL system interface along with a short description of each operation.

5.1 Context management

Contexts are created as empty address spaces using the Chorus virtual memory abstractions. Each context can support a variable number of virtual memory regions, sparsely allocated and each backed by a secondary storage entity managed by an external memory mapper [Abrossimov 89b]. A unique identifier is returned when an object is created that can be used to denote the context and in particular used to destroy the context.

Context Management:	
ctxtCreate()→ctxtCap	creation
ctxtDelete(contextId)	destruction

5.2 Object management

Objects, consisting of an initial code region and data region are created according to a template (the type) and a set of attributes. Such attributes determine if this object will be globally known and whether it is a persistent object.

Object Management:	
objCreate(Class, options)→Oid	creation
objDelete(Oid)	destruction
objCopy(Oid, options)→Oid	Create a copy
objSelf(Oid)→Oid (global)	get global id

Each object is instantiated into the current context, occupying two regions of virtual memory, holding the code for that object[4] and its data region. The COOL system manages an object identifier: an internal structure describing the format of that object. This structure holds, among other things, the capabilities of the secondary storage entities, segments, that back the two virtual memory regions.

The COOL mapper is responsible for managing the relationship between the virtual memory regions and the secondary storage segments. This it does by responding to *pull-in* or *push-out* requests, generated by the Chorus kernel, in the normal course of object execution. The COOL mapper is an instance of the general model of user level mappers that Chorus defines as part of the virtual memory architecture [Abrossimov 89a] .

Like Chorus, as long as the mapper interface supports the operations expected by the Chorus Nucleus (pull-in, push-out etc), the mapper may be redefined by the system builder. This allows the language run-time layer to map its own objects to the underlying COOL object format.

[4] Code regions may be shared if the language run-time layer wishes to.

5.3 Communication

Object interaction is based on the underlying Chorus communication model. Objects created with a **global** attribute are assigned a port for communications and use this as the end-point for messages. Messages, synchronous or asynchronous, are delivered to the target object.

Object communication:	
objSend(to, msg, objList)	Msg an object
objReceive()→(msg, objList)	recieve
objReply(msg, objList)	associated reply
objCall(to, msg, objList)	-
→(msg, objList)	synch call

5.3.1 Object migration

A significant feature of the COOL communications model is that the user is able to specify a set of local objects that will be **migrated** from the calling to receiving context when a message is sent.

When a user messages another object, and includes a list of object references, COOL uses this list of references to determine the layout of the objects by examining the associated object identifier. It then sends this object identifier, including the capabilities for its text and data region to the remote context. At the remote context, COOL receives the object descriptor, builds the appropriate region descriptors and uses the included secondary storage segment identifiers to ask the COOL mapper to map the objects' code and data region into the current context.

COOL ensures that a copy of the object being migrated is saved so that simple failures in the migration protocol can be recovered from.

The communication facilities are used simply to send the object descriptors, the actual migration of objects is achieved by the underlying virtual memory mechanisms unmapping the text and data segments of an object from the old context, and re-mapping them into the new context.

5.3.2 Object Groups

COOL also allows objects to be grouped according to communication categories. This allows groups of objects to be denoted that will be considered by the communication system as a single endpoint. The functional model of delivery allows messages to be delivered to all objects in a group[5], only one (chosen at random) or a particular object chosen by site or name.

Object group management:	
grpAllocate()→(groupCap)	group create
grpInsert(groupCap, Oid)	insert a member
grpRemove(groupCap, Oid)	remove a member

[5]The broadcast mode is supported directly by the Chorus microKernel, as such it is a best effort delivery and not atomic. It is left to higher layers of the system to build particular delivery semantics.

5.4 Persistence

COOL allows system developers to detach objects from the context they reside in and move them to backing store returning a persistent object identifier (pOid). At a later point such objects can be re-mapped from store to a context. This base functionality is not responsible for storage consistency, which is left to higher layers.

Object storage:	
objSave(Oid)→(pOid)	detach object
objSaveCopy(Oid)→(pOid)	store a copy
objRtrv(pOid)→(Oid)	attach an object
objRtrvCopy(pOid)→(Oid)	attach a copy
objDelSaved(pOid)	delete object

Because this mechanism for object persistence has no understanding of object format, it is unable to perform any form of reference conversion. Thus object references held internally to an object are saved in whatever form they are manipulated by the language. When an object is restored into a context, there is no guarantee that these references will still be valid. This problem does not occur for global object references, since these are generated by the Chorus system and are always valid. However, language generated, virtual memory references are not tracked by COOL. In addition, since activity in an object will have an associated thread stack whose format COOL does not know, we delete, or await termination of activity in an object, before storing it.

To provide some form of consistent persistence, we also support a mechanisms whereby an entire context is stored, including all objects within that context. This form keeps global object references and thread state information, and ensures all virtual memory references remain valid because it stores the entire state of the virtual memory. At a later point the context can be *restarted* by pulling in all its objects and restoring the state of activity.

Context persistence:	
ctxSave(cntxtCap, name)	save context
ctxRestore(name)→(cntxtCap)	restore context

6 The COOL C++ testbench

To demonstrate the feasibility of our approach we have built a run-time system that allows C++ objects to be mapped into the COOL environment and treated as system objects. Thus, C++ objects are capable of being created, messaged, migrated and stored as persistent entities.

Each C++ object that will be used as a COOL object is defined from a base class COOL, this is a syntactic measure only. The normal chain of C++ compilation is carried out, however, we require that each C++ class template resides in its own file. COOL/C++ objects are compiled and then partially linked at a static, user defined address. Such objects are left in executable files (with the same name as the class) but are not fully linked. A further constraint is that any COOL/C++ object that wishes to export a set of methods must make these object virtual. We

require this because we intercept normal C++ invocation and carry out COOL processing (See 6.3). This is done by replacing the standard v_table with a COOL specific one.

When the object create function is called, the COOL class manager is passed the class name of the object and asked to locate and instantciate an instance of that class. The class manager locates the file containing the executable code for the object, and builds an object descriptor which is given to the COOL system.

The COOL system then uses the COOL mapper to map in the correct text and data segment. At this point the COOL/C++ object is dynamically linked by COOL with the system defined standard i/o library and with the COOL system interface itself.

This process is represented graphically in figure 2.

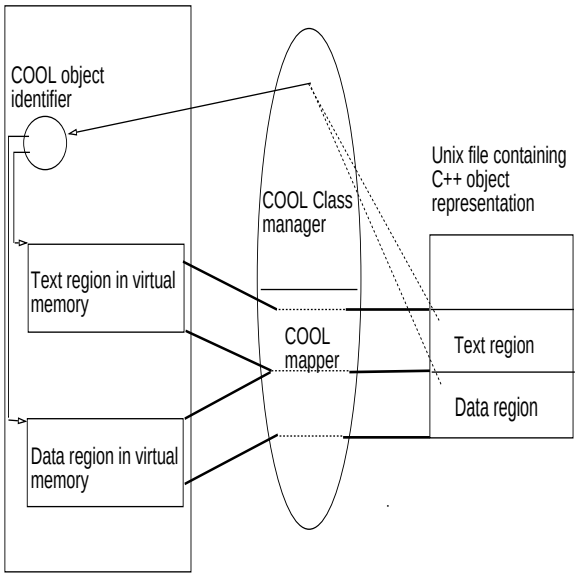

Figure 2: Mapping a C++ object into a COOL context

A number of additions have been made to the C++ object model to deal with the mapping of C++ to COOL. These are the notion of **relocatable pointer**, **member objects**, and **active objects**.

6.1 Relocatable pointers

A relocatable pointer is used to allow the C++ run-time to effectively relocate dynamic structures when their parent object is migrated. Each relocatable pointer is an instance of a class, paramaterised by the type of the pointer.

All dynamically allocated data is allocated in the same data region as the objects data. The data part of the pointer is an offset into this region. De-referencing of an object pointer computes the absolute address by simple addition.

6.2 Members

A second concept is that of *members* which allows us to dynamically build tree structures of related objects and to migrate the entire tree from its root object. This enhancement to the C++ object model is required to deal with distributed applications built from groups of related objects. In particular, in a system which supports dynamic objects, it is necessary for the programmer to provide hints on which objects belong to related structures. When the system migrates an object, it uses this information to allow it to migrate the group of related objects, thus aiding applications that make use of complex object relationships.

6.3 Active objects

To provide a degree of parallelism, objects may be created as **active**, with a lightweight Chorus thread beginning execution at a designated entry point in that object.

The data associated with a thread is stored in the data region of the object that created the thread. This allows a simple form of thread migration. When an active object is migrated, the thread is deleted and restarted from the entry point when the object is re-activated in the new context.

In addition, each thread manages a list of object identifiers that it has traversed; the invoked object list (IOL). When a thread makes an object invocation, it transparently adds the object identifier of the object to be invoked to its IOL. When the method call returns, the identifier of the returning object is transparently removed from the IOL.

An advantage of this approach is that we are able to make the system call interface specific to a context and not replicated in each object. When a system call is made, through the dynamically linked COOL system interface, the COOL system object is able to determine which object made the call by examining the IOL associated with the calling thread. The required system call can then be carried out on the correct object.

7 COOL and the ANSA distributed programming system

The ANSA testbench [ANSA 89], an implementation of the ANSA/ISA architecture provides a distributed programming package that is well suited to experimentation. Designed as a generic extension to current non-distributed and possibly non-object oriented languages, the package provides the facilities to encapsulate existing code within ANSA objects, to define the interface to these objects, and to carry out invocations amongst (possibly) distributed objects.

For our purposes, it is the tools to specify the functional interface to ANSA objects, and to carry out invocation amongst them that is important.

The package consists of two main functional components to achieve this; an interface definition language (IDL) used to describe the functional interface to an object, and a distributed programming language (DPL) that is embedded into standard C or C++ code. An associated stub generation tool uses the interface definition language to create remote procedure call stubs, which, via the use of a pre-processor, are embedded into the C++ objects.

In our experimental work we have adapted the ANSA stub-generation code to work above the COOL environment, thus providing us with a simple remote procedure call package layered onto COOL.

Our particular area of work has been concerned less with the use of RPC's and more with the notion of a single invocation mechanism. By using the ANSA RPC as the entry point into the invocation mechanism we model all COOL objects as remote, and hence all invocation as an RPC.

We distinguish between three basic invocation cases; local invocation within an address space, invocation across address spaces but local to a site, and the standard remote procedure call across sites.

The last case is dealt with by the ANSA RPC package and represents the most expensive invocation mechanism. The cost of the call lies in the marshalling of parameters, the actual message send, the receiving and unmarshalling, the dispatch to the correct method and the equivalent costs for the results.

The second case we have mainly ignored because work has already been carried out into this case [Bershad 89]. However, because we map communication onto Chorus IPC, the local case send is far cheaper than the remote case.

The first case, the local procedure call is the cheapest. To deal with this, we have adapted the RPC package so that a local object is recognised and the normal C++ invocation step is performed. This, though more costly than a standard procedure call is still far less expensive than a local RPC. Typically, recognising and converting an RPC to a local call costs 6-8 times a normal procedure call. Whereas a same machine RPC costs between 20-60 times a simple procedure call [Schroeder 89].

7.1 Optimising Invocation

Assuming that we view invocation costs as a criteria for efficiency in a distributed system, then we would attempt to always co-locate objects that communicate, and use local object invocation.

Most systems support static objects, thus the only time that co-location can be carried out is at initial configuration time. However, since, in many applications, communication patterns change over time, the initial configuration often becomes invalid.

However, in COOL we have the ability to dynamically migrate objects within the distributed system. Thus, by combining the RPC package with the migration mechanisms we are able, when an RPC request is made, to migrate the remote object to the local context. This allows us to convert the RPC into a local procedure call.

So far we have only performed simple experiments with this mechanism.

In particular we have developed a test application that allows experimentation with a number of communication parameters.

These include the pattern of interactions, the length of interactions, the time between interactions and the amount of computation each interaction causes. When combined with an ability

to designate whether objects are free to migrate or whether they are fixed to a particular site, we are able to investigate the applicability of object migration to support efficient invocation.

Whilst we have only performed preliminary investigation into this, we have, in some simple minded cases, found an order of magnitude speed up for a particular application.

However, our work has also highlighted an extremely crucial issue; that of **conflicting mobility constraints**.

In its simplist manifestation; the decision to migrate an object is driven by the requirement to optimise a particular invocation. In a large dynamic system this approach represents a single reference or decision point based on a single piece of information. It precludes a more global view of the system, and may in fact be detrimental to the application.

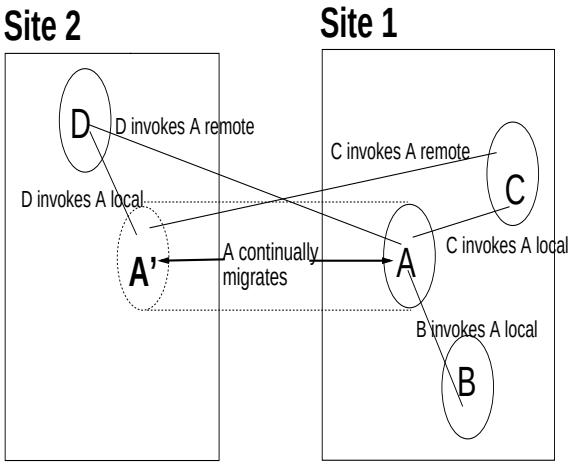

Figure 3: Performance degregation due to object thrashing

Consider a simple case as represented in figure 3. If object A is being invoked by B and C (both local) and D (remote), then in our simple case, and with a repeated invocation pattern of B D C, object A is migrated from site 1 to site 2 between each invocation of B and C. The cost of this exercise is dependent on the type of invocation, however, for simple computation on behalf of A, we have a degradation of performance of the application from a static case where A is fixed, to a dynamic case where A can be migrated.

The more general case not only considers conflicting invocation constraints but a number of other factors.

- The load on any particular site,
- the size and interaction pattern of long lived objects,
- the need to support system management policies,
- the need to handle system specified security policies,
- application requirements and

- user specified constraints.

In fact, at any point in time, the decision to migrate an object is the result of several (possibly conflicting) factors. Most work to-date on process, or object mobility [Barak 89] [Barak 85] [Jul 88] [Douglis 87] [Smith 88] [Eager 86] [Horn 89] has simply considered one of these factors, usually either load balancing constraints, or invocation costs. We feel that this approach has been too restrictive, and that for anything other than toy applications all of the above factors must be resolved.

To deal with this we have begun specification of a migration manager to be incorporated into the COOL system that will provide a focal point for these policy decisions and will drive the underlying COOL mechanisms [Weightman 90].

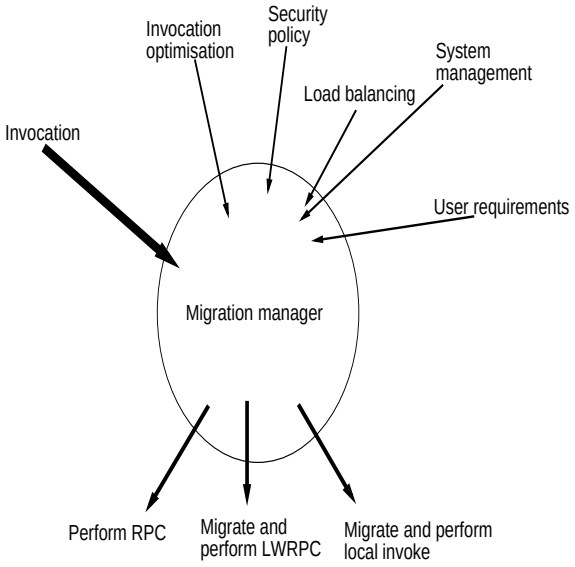

Figure 4: Conflicting requirements: Migration manager

8 Conclusion and current work

We feel that our experiences with COOL have been useful. In particular our ability to quickly map the C++ object model onto COOL bears out our original design aim of a set of generic object support mechanisms.

By adopting this approach we have been able to support a restricted set of C++ semantics in a distributed environment.

This has been further enhanced by our work with the ANSA distributed programming package. The ANSA object model supports a cleaner object interface than C++ and is already adapted to deal with distribution. By mapping the ANSA language onto COOL we enhance the ANSA language functionality with support for persistence and mobility. However, it is the use

of the COOL migration mechanisms to support the ANSA remote invocation mechanism that is the most interesting.

We have shown that by using COOL migration we are able to optimise remote invocation in a number of simple distributed applications. Our current work is a better understanding of this requirement for object mobility and its performance benefits.

A second aspect of our experiences has been with the suitability of the language support approach embodied by COOL. We have encountered a number of problems with our initial design that we are addressing in our current work.

In particular, we feel that it is not plausible for the COOL kernel to manage fine grain objects as defined by a language such as C++. The overhead of system management of a large number of fine grained objects reduced efficiency.

To overcome this problem we are extending our notion of an object to that of a cluster of objects. A cluster is composed of multiple regions of virtual memory onto which a language can map its objects. Because the granularity of clusters if far greater than single objects we are able to reduce the amount of information managed by the system.

A second aspect of our original design which we feel needs addressing is the tension between the semantic knowledge needed to perform sophisticated operations, and the use of a generic object support platform. This was discussed in the section on object persistence that highlighted the problems of making objects persistent without an understanding of their internal format.

To address this problem we are re-designing the generic run-time layer to include a sophisticated up-call mechanism that allows the generic run-time to call into the language specific run-time to gain semantic information about particular objects.

9 Acknowledgements

Vadim Abrossimov, Sabine Habert and Laurence Mosseri were the original implementors of the COOL system. March Shapiro, Marc Rozier and Marc Guillemont contributed both time, and experience, in the early design of the COOL system. The CIDRE group at SEPT were also involved in the design of the C++ run-time environment.

References

[Abrossimov 89a] Abrossimov, V., Rozier, M. and Shapiro, M., Generic virtual memory management for operating system kernels. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 123–136, Litchfield Park AZ (USA), December 1989. ACM.

[Abrossimov 89b] Abrossimov, V., Rozier, M. and Gien, M., Virtual memory management in Chorus, Proceedings of the Progress in Distributed Operating Systems and Distributed Systems Management workshop, Berlin, April, 1989. Lecture notes in Computer Science, Springer Verlag.

[ANSA 89] The ANSA testbench Reference Manual., The Advanced Networked Systems Architecture, APM ltd, Cambridge UK. November 1989.

[Armand 89] Armand F, Gien, M., Hermann, F. and Rozier, F., 'Revolution '89 or Distributing UNIX brings it back to its Original Virtue', *Proceedings of Distributed and Multiprocessor Systems*, Ft Lauderdale, USA 1989.

[Barak 85] Barak, A & Shiloh, A A Distributed Load-balancing Policy for a Multicomputer. *Software Practice and Experience Vol 15(9).* Sept. 1985.

[Barak 89] Barak, A & Wheeler, R MOSIX: An Integrated Multiprocessor UNIX. in *Proceedings of 1989 USENIX Technical Conference.* Feb 1989.

[Bennett 87] Bennett. J.K., The design and implementation of distributed SmallTalk. *OOPSLA'87 proceedings, pp318-330* October 1987. FL USA. ACM press.

[Bershad 89] Bershad, B et al. Lightweight Remote Procedure Call In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 102-113, Litchfield Park AZ (USA), December 1989. ACM.

[Black 87] Black, A., Hutchinson, N., Jul, H., Levy, H., Carter, L., Distribution and abstract types in Emerald *IEEE transactions on software engineering* Vol SE-13, no. 1. January 1987.

[Deshayes 89] Deshayes, J., Abrossimov, V. and Lea, R., 'The CIDRE distributed object system based on CHORUS', *Proceedings of the TOOLS'89 conference*, Paris, 1989.

[Douglis 87] Douglis, F & Ousterhout, J Process Migration in the Sprite Operating System. *Proceedings of the 7th International Conference on Distributed Computing Systems.* Sept 1987.

[Eager 86] Eager, E et al Dynamic Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering.* May 1986.

[Esprit 89] Integrated Systems Architecture ISA - Esprit project 2267.

[Habert 90] Habert, S., Abrossimov, V. and Mosseri, L., "COOL:Kernel Support for Object-Oriented Environments" Proceedings of OOPSLA'90. Toronto, Canada, 1990.

[Horn 89] Horn, C & Donnelly, A Architectural Aspects of the Comandos Platform. *Technical Report* Feb 1989

[Jul 88] Jul, E et al Fine Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems.* Vol6 1 Feb 1988. pp. 109-133.

[Lea 91] Lea, R., Weightman, J. COOL: an object support environment co-existing with Unix. *Proceeding of the AFUU Unix Convention'91* CNET, Paris, France. March 1991.

[Lucco 90] Lucco, S., Anderson, D., Tarmac: A language system substrate based on mobile memory. *Proceedings of the 10th International Conference on Distributed Computer Systems* Paris france, June 1990. IEEE press.

[McCullough 87] McCullough, P., Transparent forwarding: First steps. *Proceedings of OOPSLA'87* pp 331-341, October 1987. ACM press.

[Rozier 88] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser C., Langlois, S., Léonard, P., and Neuhauser, W., Chorus distributed operating systems. *Computing Systems*, 1(4):305–367, 1988.

[Schroeder 89] Schroeder, M. and Burrows, M. 'Performance of firefly RPC.' In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 83-90, Litchfield Park AZ (USA), December 1989. ACM.

[Shapiro 86] Shapiro, M., Structure and encapsulation in distributed systems: The proxy principle. *Proceedings of the 6th International Conference on Distributed Computer Systems* May 1985. IEEE press.

[Smith 88] Smith, J.M. A Survey of Process Migration Mechanisms. *ACM Operating Systems Review.* July 1988.

[Weightman 90] Weightman, J. and Lea, R. 'Migration requirements for object mobility' *Internal report* CS-TR-91-1, Chorus systems, France.