COOL: an object support environment co-existing with Unix

Rodger Lea
Chorus-systèmes, 6 av. Gustave Eiffel, 78182 St.-Quentin-en-Yvelines, France

James Weightman
University of Lancaster, Lancaster, LA1 4YU, UK.

e-mail: rodger@chorus.fr

January 1991

1 Abstract

The COOL[1] system has been designed to satisfy two goals; to provide an efficient object oriented support layer built directly on the Chorus Micro-Kernel and to co-exist with UNIX by offering access paths, and by using Unix functionality.

In this paper we discuss how the COOL system has been designed to exploit features of the Chorus operating system model, to provide abstractions suited to object oriented systems. A significant feature of the COOL project is that we have been able to exploit the design methodology of Chorus to allow COOL to co-exist with the Chorus Unix implementation (MiX) [Armand89] yet still build our abstractions directly onto the Chorus Nucleus, exploiting its efficiency and functionality.

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 [Deshayes89]

We describe a number of developments with this C++ environment, including experiments to support a single invocation model based on the ANSA RPC package [Esprit89] and utilising the COOL object migration facilities.

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

[1] The Chorus Object Oriented Layer (COOL) is a joint project between Chorus Systems and INRIA.
[2] Service d'etudes Communes des Postes et Télécommunications

2 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.

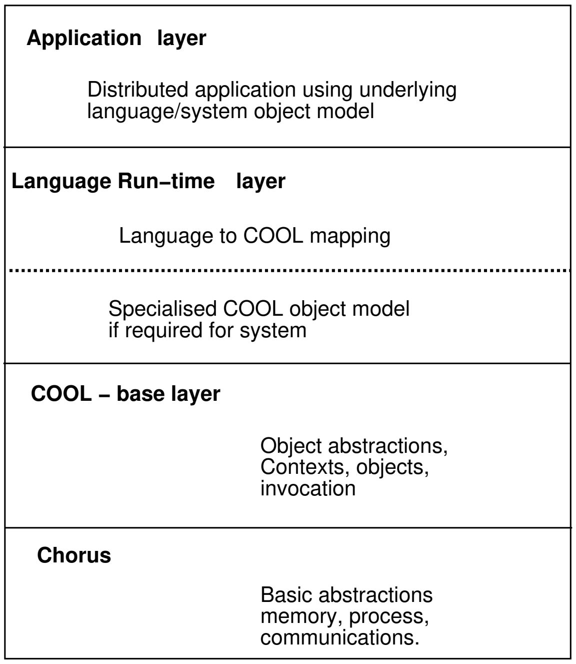

Figure 1: The COOL architecture

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 instanciate COOL objects. The COOL mapper that is responsible for managing the mapping between object representation in virtual memory, and the secondary storage representation. Lastly, the COOL activity monitor is used to manage communications directed to objects that are in transit.

3 The COOL abstractions

The COOL system is a set of abstractions that are designed to enhance those provided by the Chorus Nucleus [Rozier88]. 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 memory in which objects exist and models a traditional address space; the notion of an **Object** which is a system supported generic entity which offers 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**.

4 The COOL system interface

The COOL system and its externals have already been described in the literature [Habert90], below we give a brief resume of the COOL system interface along with a short description of each operation.

4.1 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 [Abrossimov89b]. 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 == (contextId, key)	creation
ctxtDelete(contextId)	destruction

4.2 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 Global or local unique identifier	creation
objDelete(Oid)	destruction
objCopy(Oid, options)→Oid	Create a new copy
objSelf(Oid)→Oid (global)	Return global id

[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.

Each object is instanciated 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 [Abrossimov89a] .

Like Chorus, as long as the mapper interface supports the operations expected by the Chorus Nucleus (pull-in, push-out etc), then 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.

4.3 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.

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.

Object intercommunication:	
objSend(target, msg, objectList)	Msg a target object
objReceive(receiver, delay)→(msg, objects)	recieve with associated objects
objCall(target, msg, delay, objectList)→(response, objects)	synchronous message
objReply(response, objects, objectList)	associated reply

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.

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

4.3.1 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, only one (choosen at random) or a particular object choosen by site or name.

Object group management:	
groupAllocate()→(groupCap)	create a communication group
groupInsert(groupCap, Oid)	insert a member
groupRemove(groupCap, Oid)	remove a member

4.4 Persistence

COOL allows system developers to detach objects from the context they reside in and move them to backing store. At a later point such objects can be re-mapped from store to a context.

Object storage:	
objectDetach(ctxtCap, Oid)→(persistentOid)	detach and store an object
objectAttach(cntxtCap, persistentOid)→(Oid)	attach an 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 occour 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 passivated, including all objects within that contexts. This form keeps global object references and thread 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:	
contextSave(cntxtCap, name)	save a named context
groupRestore(name)→(cntxtCap)	restore context

5 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. 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.

When the object create function is called, the COOL class manager is passed the class name of the object and asked to locate and instanciate 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.

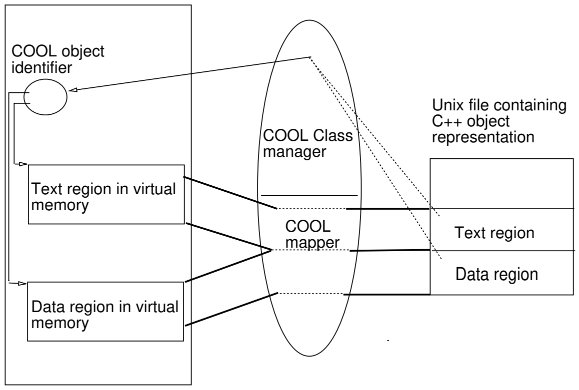

Figure 2: Mapping a C++ object into a COOL context

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

5.1 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.

5.2 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.

Each member object is actually implemented using the relocatable pointer described above.

5.3 Active objects

To provide a degree of parallelism, objects may be created as **active**, with a lightweight Chorus thread begining 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.

6 Co-existing with Unix

The COOL system is built as a layer above the Chorus Nucleus, thus directly exploiting its functionality. However COOL is not designed as a stand-alone environment. COOL co-exists with Unix in three ways; firstly, UNIX provides the development environment that will enable system builders to design and develop their individual object systems. As described above, COOL/C++ objects are standard C++ objects which are not fully linked and are loaded at a specific address. The COOL/C++ system is made aware of the structure of these Unix generated files and uses this information to allow the COOL class manager and the COOL mapper to operate correctly.

Secondly, we use Chorus/MiX to provide functionality such as the file system thereby releasing us from the implementation effort of building such functionality directly into the COOL system. This gives an extremely fast prototyping environment. In particular, COOL/C++

objects are actually held as normal Unix files, these are managed by the Unix file system but used by the COOL class manager and COOL mapper as data repositories.

Lastly, it is possible for COOL system objects to directly use the UNIX communication mechanisms to allow COOL objects to interact with UNIX applications. Thus, for example, a COOL application can gain access to an existing UNIX program. The COOL stdio object, which is system defined and always mapped into each COOL context contains a minimal set of Unix file I/O functionality. Thus a COOL/C++ object is able to create, read and write, and destroy normal Unix files.

7 Experiences

Through early experiments with COOL, both within the CIDRE project and internally at Chorus we have identified three areas that we feel need further work;

- firstly, we do not support a single invocation model, remote objects may be migrated to the local context and treated as normal C++ objects. However, communicating with a remote objects uses the underlying Chorus IPC model and requires that the programmer implement the invocation procedure. In effect, the programmer must write his own remote procedure call code.
- Secondly within the COOL C++ testbench we support traditional C++ passive objects and COOL active objects. Thus it is possible to define a C++ object that supports not only transitory invocation, but permanent activity. We have had difficulty understanding the semantics of this, in particular, who controls that object. For example, a request to delete an object may be made by any thread in that object. If that object has a permanent thread associated with it, it should only be that thread who can make such requests.
- The last area of difficulty is the relationship between activity, local object identification and object persistence. Our experience has been that object persistence makes little sense without the object's context. This context is contained as a set of virtual memory references within the object, the stack associated with any activity in the object and state held on behalf of the object. COOL does not have enough understanding of object format or stack format to allow it to convert persistent object refernces. Thus we have found that context persistence is a more useful mechanism because virtual memory references remain valid when the context is restored..

Whilst we are developing a successor to COOL to deal with the second two problems, we have already carried out some experimentation with supporting a single invocation model on COOL.

7.1 Invocation in distributed systems

One of our main aims has been to investigate invocation mechanisms in distributed, object oriented systems. In particular, we view invocation as being subdivided into three basic catagories.

- Invocation of objects on remote nodes in the distributed system.
- Invocation of objects on the same node but in different address spaces.
- Invocation of objects in the same address space.

To succesfully build distributed transparent mechanisms to support object based systems we must provide a single invocation interface for the programmer. However, we must map this single interface to the three cases described above. To do so, requires that we provide a set of mechanisms to support the three invocation cases, and choose the most appropriate mechanism at the time of invocation.

Our current experiments we have adapted a remote procedure call package to provide our single interface to object invocation of C++ objects.

7.1.1 COOL and ANSA RPC

The ANSA testbench [ANSA89], an implementation of the ANSA/ISA architecture provides an RPC package that is well suited to experimentation. The RPC package consists of two main functional components; an interface definition language (IDL) used to describe the functional interface to an object, and a distributed programming language (DPL) that is embeded into standard C or C++ code[5]. 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 in 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.

Referring to our three basic invocation cases listed above;

The first 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 equivilent costs for the results.

The second case we have mainly ignored because work has already been carried out into this case [Bershad89]. However, because we map communication onto Chorus IPC, the local case send is far cheaper than the remote case.

The last 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 3-4 times a normal procedure call. Whereas a same machine RPC costs between 20 and 60 times a simple procedure call [Schroeder89].

[5] The standard ANSA platform only supports C code. The RIBA platform, developed by STC/STL (ISA partners) has been enhanced to deal with C++ code.

7.1.2 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.

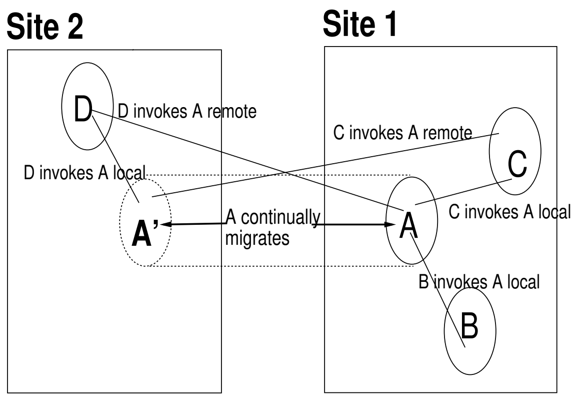

Figure 3: Performance degregation due to object thrashing

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 [Barak89] [Barak85] [Jul88] [Douglis87] [Smith88] [Eager86] [Horn89] 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 [Weightman90].

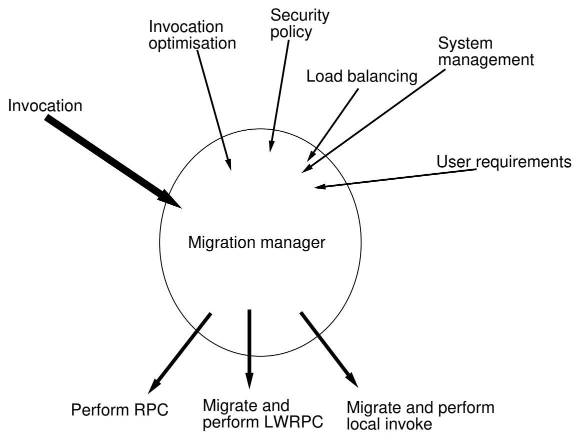

Figure 4: Conflicting requirements: Migration manager

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.

However, the problems highlighted in section 7, and our experiences with migration and invocation has led us to modify our design of a successor to COOL.

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++.

To overcome this problem we are extending our notion of an object to that of a container. A container is composed of multiple regions of virtual memory onto which a language can map its objects.

By adopting this approach we are able to extend our basic unit of memory from a simple two region entity to a multiple region entity. The base layer only manages this group of regions, it is the job of the run-time layer to map and manage objects within these regions. To allow the base-layer to carry out operations which require an understanding of the object format, we make extensive use of upcalls to the language specific run-time layer.

We then define a process of assembly on containers such that two containers with non-overlapping regions may be assembled in one address space. This subsumes our notion of object migration.

We are also defining a single level invocation mechanism, supported by the base COOL level, that will use this asssembly ability to assemble containers which hold objects that wish to invoke each other. Our aim with this work is to use our experiences with migration as input to the design of this invocation model.

In addition we are attempting to define a portion of the system (or language) run-time layer that will be common to all (or most) object models. This will allow us to explore to what degree we can support the Object oriented metaphor without restricting what can be built onto our platform. Lastly, we are addressing ways in which *alien* objects, in particular, Unix abstractions, can be modeled in the COOL environment, thus allowing us to interact more closely with UNIX.

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

[Abrossimov89a] 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.

[Abrossimov89b] 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.

[ANSA89] The ANSA testbench Reference Manual., The Advanced Networked Systems Architecture, APM ltd, Cambridge UK. November 1989.

[Armand89] 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.

[Barak85] Barak, A & Shiloh, A A Distributed Load-balancing Policy for a Multicomputer. *Software Practice and Experience Vol 15(9).* Sept. 1985.

[Barak89] Barak, A & Wheeler, R MOSIX: An Integrated Multiprocessor UNIX. in *Proceedings of 1989 USENIX Technical Conference.* Feb 1989.

[Bershad89] 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.

[Deshayes89] Deshayes, J., Abrossimov, V. and Lea, R., 'The CIDRE distributed object system based on CHORUS', *Proceedings of the TOOLS'89 conference*, Paris, 1989.

[Douglis87] Douglis, F & Ousterhout, J Process Migration in the Sprite Operating System. *Proceedings of the 7th International Conference on Distributed Computing Systems.* Sept 1987.

[Eager86] Eager, E et al Dynamic Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering.* May 1986.

[Esprit89] Integrated Systems Architecture ISA - Esprit project 2267.

[Habert90] Habert, S., Abrossimov, V. and Mosseri, L., "COOL:Kernel Support for Object-Oriented Environments" Proceedings of OOPSLA'90. Toronto, Canada, 1990.

[Horn89] Horn, C & Donnelly, A Architectural Aspects of the Comandos Platform. *Technical Report* Feb 1989

[Jul88] Jul, E et al Fine Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems. Feb 1988.*

[Rozier88] 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.

[Schroeder89] 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.

[Smith88] Smith, J.M. A Survey of Process Migration Mechanisms. *ACM Operating Systems Review.* July 1988.

[Weightman90] Weightman, J. and Lea, R. 'Migration requirements for object mobility' *Internal report* CS-TR-91-1, Chorus systems, France.