CS/TR-93-68

COOL: system support for distributed object-oriented programming

Rodger Lea, Christian Jacquemot
Chorus systèmes
Eric Pillevesse
Service d'Etudes communes de La Poste et de France Télécom

approved by:

© Chorus systèmes, 1993

COOL: system support for distributed object-oriented programming

CS/TR-93-68

Chorus
SYSTEMES

CHORUS® is a registered trademark of Chorus systèmes.

All other products or services mentioned in this document are identified by the trademarks or service marks of their respective companies or organisations.

© Chorus systèmes, 1993.

All rights reserved. No part of this document may be reproduced, stored in a retrieval system, translated, transcribed or transmitted, in any form or by any means manual, electric, electronic, electro-magnetic, mechanical, chemical, optical, or otherwise without the prior explicit written permission of Chorus systèmes.

Contents

1 Introduction **1**

2 The Chorus object-oriented layer: COOL **2**

2.1 The COOL v2 architecture 3

2.2 The COOL-base 3

2.3 The COOL generic run-time 5

2.3.1 The activity model 6

2.3.2 Up-Call model 7

2.4 The language specific run-time 7

2.4.1 Implementation details 9

3 Performance **10**

4 The CIDRE intelligent document system **11**

4.1 Groupware as a distributed application exemplar 12

4.2 CIDRE 12

5 Using COOL to support CIDRE's requirements **13**

5.1 Experiences with CIDRE 14

6 Related work **15**

7 Conclusion and current status **15**

1 Introduction

Distributed systems are by their very nature, large and complex applications. They require the interaction of many individual components scattered throughout a distributed collection of hosts which are often physically dispersed.

Interaction is usually modelled on a message passing abstraction where services are requested by sending a request message to a service provider and receiving an asynchronous or synchronous reply. Service providers are often large grained encapsulated entities, whose interface is defined by its message protocol. Internal synchronisation of multiple competing requests is handled by the service provider either by message queuing, or language synchronisation primitives.

Such systems are natural candidates for the object-oriented model of software development simply because the way that the majority of such systems are built maps closely to the object-oriented model. Service providers are large grained, active objects; message protocols define an ad-hoc type interface and message passing is a low level mechanism that supports method invocation.

This obvious mapping has led many groups to attempt to extend existing object-oriented languages with support for distributed objects, either by adding remote message passing facilities (based on RPC) or supporting distributed objects [21] [22].

This approach has had mixed success. On the one hand it has demonstrated that the OO languages provide sufficient support for building distributed applications. However, because the efficiency of such an approach has been so poor it has served as a proof of concept but has failed to provide the breakthrough that many in the distributed systems community have hoped for.

This inefficiency is mainly caused by a mismatch between the services and abstractions that systems provide, and those that languages offer. System services are often generic, designed to support multiple uses and achieving this with a lowest common denominator solution. In addition, the majority of existing operating systems provide abstractions that were never designed to support modern programming languages and in particular, were never designed to support distributed applications.

For example, object-oriented languages deal with fine grained objects. The majority of modern systems provide an abstraction of an address space as the smallest system supported concept. It is the compilers job to match the fine grained language model to the coarse grained system model. For a single address space application this is fine, however, for distributed applications, spanning multiple address spaces, the compiler support breaks down because the compiler is not aware of the environment outside a single address space. Equally, some languages support lightweight activities or active objects, again most systems support a heavier notion, a process. Mapping between the two is a complex and often costly task. Lastly, current operating systems provide distributed inter-process communication using protocols designed for unreliable networks and often implemented as an "add-on" feature. These communication mechanisms

---

In: Communications of the ACM, Special issue on Concurrent Object Oriented Programming, September 1993, Volume 36, Number 9, ACM Press New York

are often too costly to support applications built of fine grained objects, working in a tightly coupled manner and making heavy use of inter-object invocation.

To deal with these kinds of mismatches a number of researchers have attempted to extend their underlying system with some support for their particular programming model, [23] [24].

While we feel that this approach is correct, it doesn't go far enough. System support needs to be both efficient, flexible and it must fit in with existing systems. Rewriting an operating system from scratch to support a particular programming model is a time consuming and difficult process often resulting in more time devoted to operating system engineering issues than the research goal. This has tended to force most people into the traditional solution of building their support environment above an existing operating system leading us back to the mismatch between language and system services.

However, with the availability of micro-kernel architectures such as Chorus[3], Amoeba[1] and Mach[2] which provide a basic set of abstractions designed to allow people to build operating systems, it is now possible to explore how operating systems can better support programming models.

Our goal within the COOL project has been threefold:

- to provide a set of generic services that reduce the mismatch between system abstractions and language abstractions.
- to provide these services at the operating system level so that we are not hampered by the inefficiencies of building above the existing operating system. To do this we wish to extend the CHORUS micro-kernel with abstractions more suited to object-oriented systems.
- to provide these services in a way that they co-exist with existing traditional operating systems; in our case UNIX. This allows new applications to use existing data and services and provides a way to evolve existing applications.

This paper is structured as follows; we first introduce the COOL v2 architecture, discussing its functionality and usage. We then outline details of its implementation and its performance. After this we introduce the CIDRE distributed document application that has been built using the COOL system, and explain both its functionality and its use of COOL. We use CIDRE to justify some of the design decisions and as a proof of concept for our research. In the final sections we discuss our experiences and outline future directions.

2 The Chorus object-oriented layer: COOL

The COOL project is now in its second iteration, our first platform, COOL v1[1], was designed as a testbed for initial ideas and implemented in late '88 [4] [5] [8]. Our initial platform implemented a simple notion of an object as a micro-kernel supported abstraction, with mechanisms

[1] COOL v1 was built as a joint project between Chorus Systèms, Institute national de research en informatique et automatique (INRIA) and the SEPT

to instanciate objects, migrate and store objects and make invocations on local and remote objects. COOL v1 was used to support a first version of the CIDRE system during the development of which we identified several problems. In particular, we found that our generic mechanisms were not easily used by the languages, partly because of the cost of our system level objects, and partly because our mechanism were too generic. Further details of this can be found in [9].

We began a redesign of the COOL abstractions in 1990. This work was carried out in conjunction with two European research projects, the Esprit ISA project [6] and the Esprit Comandos project [7], both building distributed object based systems. This work was specifically designed to address this issue of providing generic support mechanisms, yet allowing those mechanisms to be efficiently used by languages.

The result of this work has been the specification of the COOL v2 system and its initial implementation in late '91.

2.1 The COOL v2 architecture

COOL v2 is composed of three functionally separate layers, the COOL-base layer, the COOL generic run-time (GRT) and the COOL language-specific run-time layer.

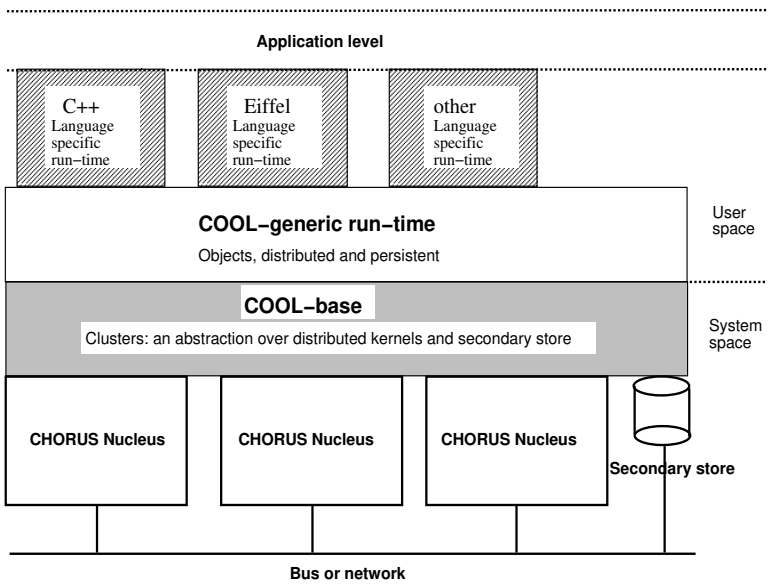

Figure 1: COOL v2 Architecture

2.2 The COOL-base

The COOL-base is the system level layer. It has the interface of a set of system calls and extends the CHORUS micro-kernel (or Nucleus) abstractions. It acts as a micro-kernel for object-oriented systems, on top of which the generic run-time layer can be built.

The COOL-base provides memory abstractions where objects can exist, support for object sharing through distributed shared memory *and* message passing, an execution model based on threads and a single level persistent store that abstracts over a collection of loosely coupled nodes and associated secondary storage.

In our initial work with COOL v1 our base level supported a simple generic notion of objects. This proved to be too expensive in terms of system overhead. In COOL v2 we have moved the notion of object out of our base layer and replaced it with two more generic abstractions, *clusters* and *context spaces*.

A cluster is viewed from higher levels as a place where related objects exist. When mapped into an address space, it is simply a collection of virtual memory regions [10]. The mapping may be on an arbitrary address. The collection of regions that belong to a mapped cluster is a set of CHORUS regions backed by segments, and forms a semantic unit managed by the base layer. By using a distributed virtual memory mapper[2], regions and hence clusters, can be mapped into multiple address spaces or contexts, which leads us to the notion of context space.

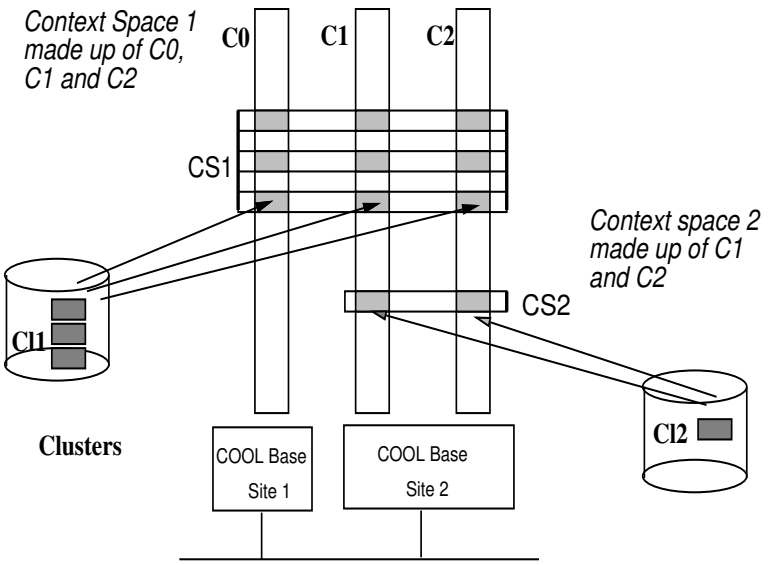

Figure 2: anatomy of a cluster

A context space is a collection of distinct address spaces on one or more sites. Any cluster belonging to a context space is mapped into all contexts of that context space. For example in figure 2, cluster Cl1 is mapped into the context space comprising contexts C0, C1 and C2, and cluster Cl2 is mapped into contexts C1 and C2. Note that cluster Cl1 is made up of three separate secondary storage segments and so is mapped by three virtual memory regions. In the case of Cl1 we must enforce that the cluster is always mapped at the same addresses in the contexts forming the context space. Therefore, a context space represents a distributed virtual address space, and can be used to share clusters among threads of execution of a particular context space. It is important to note that when a cluster is mapped into a context, it will be

[2] A mapper in CHORUS supports the relationship between virtual memory regions and the secondary storage segments that a region 'maps'

mapped into the same range of addresses in all contexts in the context space. However, the binding between a cluster and a range of addresses may be changed. Thus a cluster may be mapped out of a context space, and remapped at a later point at a different set of addresses. Of course, this will require that higher layers in the system deal with any problems of pointer relocation.

A significant difference between the distributed shared virtual memory (DSVM) model of COOL-base and more standard implementations of DSVM is that cluster spaces offer a means to structure the address spaces belonging to the contexts managed by the COOL-base. Although a cluster is mapped into several contexts at one time, is does not need to be mapped into **all** contexts managed by COOL-base. This approach means that we have more control over memory allocation and are not forced to allocate memory in all contexts. If a memory clash occurs when a context space extends into a new context, then we can unmap the cluster and map it back in at a different set of addresses.

Each cluster is uniquely identified in the system as the unit of persistence. Clusters can have references to other clusters and they are subject to garbage collection.

The COOL-base also provides a low level mechanism for communication between clusters. This can be used to implement invocation of objects that exist inside the cluster. Transparent remote invocation is achieved with a simple communication model which uses the CHORUS communication primitives and protocols. This model supports multiple mechanisms so that invocations among clusters on a local site may use a lightweight invocation mechanism, whereas between clusters on different sites we use a traditional invocation model.

The COOL-base maps-in clusters on behalf of the upper layers. It can be used to enforce an invoking thread to carry on execution in a remote address space. In addition, because clusters are persistent, the COOL-base provides a means to locate non-active clusters, i.e., clusters currently swapped-out on secondary storage and load them transparently into a cluster spaces. This model is similar to that of the Clouds v2 project [12]. We use the virtual memory mapper to store and retrieve passive clusters to and from secondary storage by performing load and flush operations on virtual memory regions. This model is similar to paging in a page based virtual memory system and provides an implementation of a single level store.

2.3 The COOL generic run-time

The generic run-time (GRT) implements a notion of objects. Objects are the fundamental abstraction in the system for building applications. An object is a combination of state and a set of methods. An object is an instance of a class which defines an implementation of the methods.

The GRT has a sub-component, the virtual object memory that supports object management including: creation, dynamic link/load, fully transparent invocation including location on secondary storage and mapping into context spaces.

Two types of object identifiers are offered by the generic run-time: domain wide references and language references. A domain wide reference is a globally unique, persistent identifier. It may be used to refer to an object regardless of its location. A language reference is a virtual memory reference and is valid in the context in which the object is presently mapped.

Objects are always created in clusters. Each cluster's address space is divided into two parts: the first one is used to store all the structures associated with the cluster used by the generic run-time, the second one is used to store the application objects.

The classes are structured in modules which are application defined clusters of associated objects. The generic run-time allows the code to be dynamically linked and offers a primitive to link a module. When an instance of a class is created in a cluster, the class descriptor is saved in the cluster. This class descriptor is used to retrieve the appropriate module and therefore the appropriate class when a cluster is remapped in another address space.

2.3.1 The activity model

The generic run-time provides an execution model based on the notion of **activities** which are mapped onto CHORUS kernel supported threads; and **jobs** which model distributed execution of activities.

An activity is a thread of execution and is created under application control and launched within a particular object. The activity is then capable of diffusing to other object, in other clusters or other machines by invoking other objects. When invocation is carried out, the GRT causes the thread of control to be passed from the invoking to the invoked object. In the remote case this will be carried out using a message to transport thread information and parameters to the remote machine.

A job gathers together a number of contexts, each supporting several clusters, which in turn have multiple objects. Each cluster can support multiple activities, with more than one activity capable of running within the same object at any particular time. The job is the unit of distributed control and will often represent a single distributed application.

Since the model supports multiple threads of control we have to provide support for synchronisation. Since COOL is a distributed system, we support not only the usual local synchronisation primitives, but also a set of distributed synchronisation primitives based on a distributed token manager.

The basic local synchronisation primitives are semaphores, mutexes and multiple reader/single writer locks. These are implemented as GRT services, which when needed will call into the COOL-base level. These services allow programmers to implement synchronisation as they need, so for example, by coding the synchronisation constraints explicitly into the object's method code. Alternatively, it is possible to use pre-compiler techniques to implement a language specific synchronisation model that is implicit in the code.

In addition to the basic local synchronisation primitives that are used to synchronise between activities running in a single address space, we also provide support for distributed synchronisation primitives that allow distributed objects to synchronise. Distributed synchronisation is based on a distributed token manager that is accessed via a set of GRT calls. The token manager implements a simple get/release token model, and relies on hint information to locate tokens. The maintenance of the distributed hint information uses the CHORUS message passing mechanisms and avoids the use of a broadcast primitive.

2.3.2 Up-Call model

One of the main problems with trying to use a single generic base to support multiple language level models is that of semantics. Most languages, and systems have their own semantics, each of which are subtly different. To allow us to build sophisticated mechanisms that support multiple models we have defined a generic *run-time to language* interface based on up-calls that enables specialisation of the GRT mechanisms.

The up-call information, and associated functions are used for a variety of purposes, including support for persistence, invocation and re-mapping between address spaces. In fact, any time where the generic run-time needs access to information about objects that only the language specific environment will know.

For example to support clusters persistence, and hence object persistence, we need access to the layout of objects to locate references held in the objects data. When a cluster is mapped into an address space all the objects are scanned by using the appropriate up-call function to locate the internal references (to external objects) and performing a mapping from the domain wide references (used when an object is on secondary storage) to address space specific references, this technique if often called pointer swizzeling.

Another example is for object invocation; invocations between objects in the same cluster are based on the standard method invocation of the language. Invocations between objects in different address spaces use the model offered by the COOL-base layer (CHORUS communication primitives). A type of proxy, called an **interface object** is used to trap the normal function invocation and replace it by a remote invocation which marshals the parameters, issues a remote procedure call, and unmarshals the results. At the receiver, a dispatch procedure, which is part of the up-call function associated with an object is used to call the appropriate method on the appropriate object.

Invocation may use the underlying cluster management mechanisms to map clusters into the calling address spaces for efficiency reasons, or locally to allow light-weight RPC but maintain protection boundaries.

2.4 The language specific run-time

The language specific run-time maps a particular language object model to the generic run-time model. This may be achieved through the use of pre-processors to generate the correct stub code to access the GRT functionality and the use of an up-call table to allow the GRT to access language specific information.

A pre-processor called COOL++ supports an extended version of standard C++ which is adapted to run on the COOL GRT. The extensions are minor, and reflect a programming convention that we choose to adopt rather than a required extension to the language.

The most significant difference is that we have chosen to use an interface definition language (IDL) to define objects, and that we use these to generate interface objects that are used to represent all objects in the system.

The programmer when defining a class, will also define an interface that the class exports.

We use a keyword, **interface** to denote this. In addition, we use a keyword **implements** to tell the pre-processor which set of methods implement a particular interface.

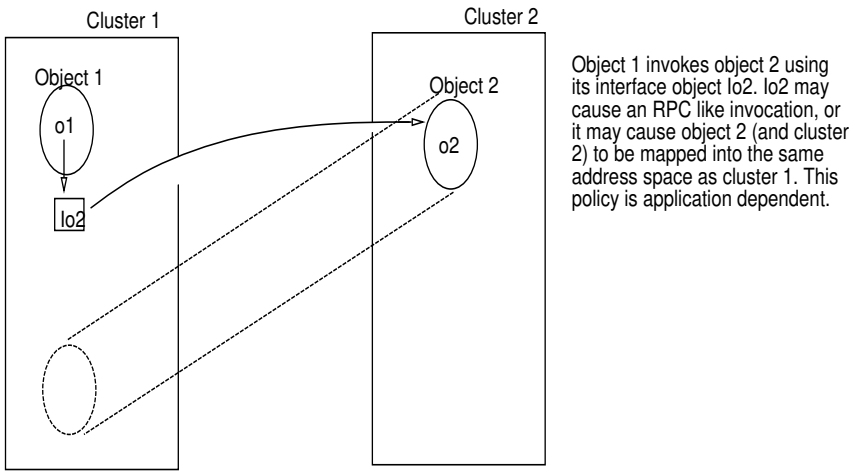

Figure 3: Using interface objects

Using this information, the pre-processor is capable of generating a special class, instances of which are known as **interface objects**. Interface objects are used in COOL++ to access all objects. The code for an interface object is generated automatically by the pre-processor and includes code to carry out remote invocation, or to map objects locally.

The user can use this interface object not only to access methods defined on the objects, but also to drive the decisions about where such an object will be invoked if it is to be mapped, or shared using distributed shared memory. To do this, the interface object contains a set of methods that allow application control over the invocation and mapping policy. The advantage of this approach is that the programmer only ever sees local memory pointers, and can choose to ignore the distributed nature of the application if required, simply using the default policies in the interface object.

There is a cost to using this model because all invocation goes through the local interface object. When an object is mapped locally then we can choose to update the interface object to optimise local invocation. However, we will always pay an extra level of indirection. We feel that this is acceptable for the ease of programming that it offers.

It should be stressed that this is a programming convention we have chosen to adopt, and one which hides much of the distributed and persistent nature of the programming environment. It is quite possible to use standard C++ and make explicit calls to the GRT to invoke and manage remote objects. Thus for example, a particular pre-processor may only generate upcall information and make visible global object references. In this case the programmer would not pay the cost of the local invocation we discuss above, but equally not see a transparent programming model.

2.4.1 Implementation details

The Chorus micro-kernel supports an architectural model in which an operating system is made up of a set of *actors*. Each actor implements a subset of the overall system functionality, uses the micro-kernel to access resources and uses message passing to communicate with other actors. The overall operating system functionality is the sum of these actors. For example, in the Unix implementation, known as CHORUS/MiX, a set of actors implement the overall Unix system, see figure 4. The **process manager** (PM) acts as the interface for Unix processes and services all kernel traps. It implements the basic process management functions, such as *exec()*, but will communicate with the other actors for their services. In turn, the **object manager** (OM) implements the SVR4 file system services, the **streams manager** (StM) implements SVR4 streams and the **ipc manager** (ipcM) implements SVR4 inter process communication.

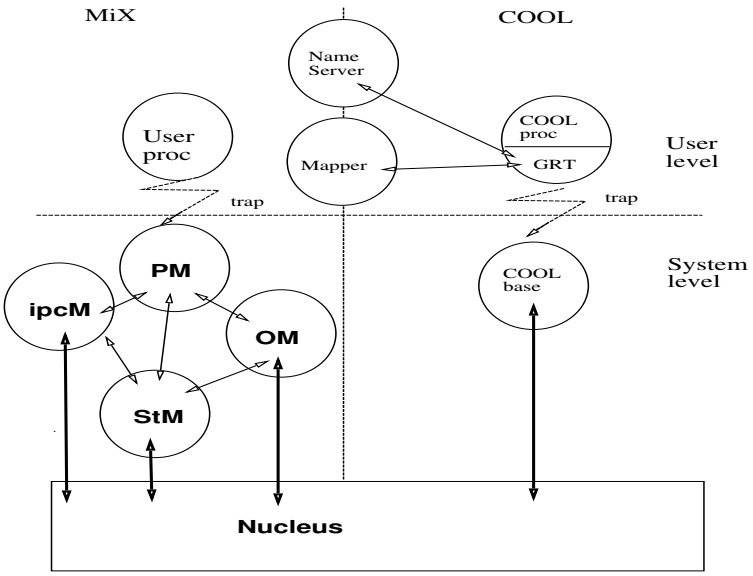

Figure 4: COOL v2 Implementation outline

The micro-kernel allows other actors to be statically or dynamically added to an existing system. New actors may be integrated into the system to provide extended functionality, or they may implement a separate system interface.

In the case of COOL, the COOL-base actor is dynamically loaded into the system address space when COOL is booted and provides its own system interface. This interface is accessed via a trap mechanism. Thus COOL provides an example of a second operating system personality running alongside the traditional Unix operating system.

However, as can be seen from the diagram, COOL as it is currently implemented is not a completely separate operating system, rather some of the COOL servers, the name server, class manager, and mapper run as Unix processes. The reason for this is to allow these servers to access Unix services which they use to provide COOL services. Thus, the class manager uses the Unix file store as its class repository. While it would be possible to run COOL as a self contained operating system personality we have chosen not to do so because of the work required

to implement device drivers and other basic system functionality.

In figure 4 we see that a COOL process has the GRT linked in as a library which in turn traps into the COOL base actor to access lower level functionality. In addition, the GRT will use the message passing mechanisms of Chorus to access functionality in the other COOL servers that run as application level servers.

3 Performance

To be able to efficiently support closely coupled applications which exhibit a persistent and dynamic nature it is necessary that we can move objects between address spaces, and between machines efficiently. A prime concern therefore is the cost of mapping and unmapping clusters of objects both between secondary storage, and between address spaces.

In the following tables we report figures for invocation and mapping of objects. These figures were recorded on a Compaq Deskpro, using an Intel 80386, running at 25MHz, with 8MB of memory and running Chorus MiX V3.2 r4.

Table 1 shows the basic cost of invocation using an interface object.

Table 1

Time for invoke(microseconds)

Standard proc call	3
Via an interface object	97
Activity create	321
Activity switch	64

Table 2

Time for UnMap operation (milliseconds)

Size of cluster	write	no write
1 * 0	103	11
32 * 0	152	11
64 * 0	137	14
128 * 0	156	14
256 * 0	157	15
1 * 128	127	13
32 * 128	187	14
64 * 128	215	14
128 * 128	214	16
256 * 128	227	19

Table 2 shows the cost of un-mapping clusters of various size. The two columns represent the cost when an un-map operation must write to disk, and when the operation simply leaves the data in physical memory.

In all cases, the table shows the cost for various size clusters, where both the number and size of the objects increase. As would be expected the overhead for mapping or unmapping a cluster will increase equally.

Table 3 shows similar data for the map operation, in this case the time taken is higher than the un-map operation because of the costs of; time to initialise internal cluster tables, time to

initialise all objects and time to swizzle pointers held internally to objects.

Table 3

Time for Map operation (milliseconds)

Size of cluster	read	no read
1 * 0	241	43
32 * 0	257	43
64 * 0	252	49
128 * 0	351	87
256 * 0	356	113
1 * 128	400	72
32 * 128	402	97
64 * 128	387	102
128 * 128	428	124
256 * 128	397	154

Table 4

Time for migrate (milliseconds)

Size of cluster	local
1 * 0	77
32 * 0	82
64 * 0	89
128 * 0	102
256 * 0	133
1 * 128	76
32 * 128	86
64 * 128	98
128 * 128	125
256 * 128	183

The final table shows the cost of migrating an object between two contexts on the same machine, in this case the operation is a combination of an unmap without writing to disk and a mapping without reading from disk.

These figures represent a non-optimised system. In addition, the times were subject to some fluctuations caused by the interaction with the COOL servers that ran as Unix processes[3]. Lastly, the increase in time when more than 128 objects are worked on is because more than 128 objects in a cluster cause the GRT to dynamically extend internal tables.

These figures are comparable to those reported in [12] and [16] but also show that although an order of magnitude more expensive than traditional RPC [1], mapping clusters of objects will provide a performance benefit when the cost of mapping plus subsequent local invocation is compared to multiple remote invocations.

4 The CIDRE intelligent document system

To justify some of our design decisions and to illustrate how a platform such a COOL v2 can be used the following section of this paper discusses the CIDRE intelligent document system.

COOL has been developed in conjunction with the S.E.P.T., a research laboratory of France Telecom. The S.E.P.T. is currently researching next generation services for public and private networks including distributed office groupware. CIDRE is an example of such groupware and is designed as a distributed platform for the creation and management of intelligent documents.

[3] A very significant optimisation could be achieved by running the mapper actor in system space, and accessing the Unix file store by messaging directly into the Object manager. This would save a context switch as we move from system to user space, a trap and context switch from mapper to Process Manager, plus the message costs. We have kept the Mapper in user space for the moment because debugging is significantly easier in user space.

4.1 Groupware as a distributed application exemplar

Groupware is software that supports virtually joined participants in structuring actions that have a common goal. By its very nature it is distributed and dynamic.

Examples of such groupware are mail, shared diaries, conference systems, electronic meeting systems, group editing and shared document preparation.

Groupware such as these can be usefully classified into two main categories, *exchange groupware* which is unscheduled interaction whereby there is a loose coupling between participants. Software such as mail or shared diaries fall into this category. These systems often use static entities which are updated only once, or infrequently, and typically exchange the entities but do not jointly work with the entities. Often these systems can be implemented in a centralised manner.

The second category is termed *production groupware*, whose role is the production of shared entities. For example, joint editing, or collaborative document management. Here the distinctive attributes are close cooperation, i.e. highly concurrent interaction on dynamic documents. Thus the application can not be modelled above a centralised system, it is by its very nature, decentralised and parallel.

4.2 CIDRE

CIDRE fits into the second category, it is a system that supports intelligent structured documents in a large scale distributed office environment.

The basic elements in CIDRE are *folders* which contain related documents and rules governing those relationships. Documents are structured ensembles made up of multiple components. Associated with documents are actions that direct the flow of documents within the network based on intelligent user or application provided scripts.

Documents are created dynamically, for example using joint editors and often comprise multiple media. Each media element is an object in its own right and may be located at different locations in the network. Hence a CIDRE document is a distributed entity.

Documents flow around the system according to control information associated with the document. For example, a draft of a component report will originate from the author, be passed to a secretary for initial proof reading and then will circulate amongst authors and interested parties for comments. Comments dynamically annotated to the document will return with the document to the original author.

To deal with unexpected events in the circulation, an intended recipient is on holiday for example, the document control script is capable of reacting and adjusting its circulation pattern. This is controlled by a Prolog program which is part of the document object and uses a simplified Petri net.

From the above brief description of CIDRE we can derive the following set of broad requirements:

- Dynamic construction: Objects representing documents can be constructed from a collec-

tion of existing smaller objects.

- Mobility: Objects need to be mobile in the distributed system, capable of moving between processing nodes according to usage patterns.
- Active: Objects need to react to their environment, evaluate new input and make decisions based on internal rules. To achieve this some objects must be active and must be capable of supporting multiple invocations.
- Sharing: Objects will be shared in a number of ways, hence requiring both concurrent access (and its control) and replication.
- Persistence: Objects, once created will exist independent of their original creator(s).
- Transparency: Objects, once in existence will move between sites and from primary to secondary storage. It is essential that transparency is available for accessing such documents, although transparency may be 'turned off' as required.
- Heterogeneity: Objects will consist of heterogeneous components, i.e. created under different systems. Further, the system will often comprise multiple heterogeneous nodes.
- Reactive: Objects must be able to deal with real world events and to change dynamically to accommodate these events.

5 Using COOL to support CIDRE's requirements

Objects in the COOL system are created in clusters. Clusters represent a form of dynamic grouping or construction. By ensuring that clusters contain related objects, a first level of support for dynamic construction is available. To support logical association, but physical dispersion we allow references between clusters which when used rely on the underlying mechanism to co-locate objects using the virtual memory. For example, when a document is being created it may consist of multiple objects, each object representing a chapter, or paragraph. Some of the text may have been created at an earlier date, or by a different author, in which case it will reside in a different cluster. When that document is being worked upon, or read, the COOL system will locate the set of clusters, and migrate them locally.

Each cluster is free to move among individual address spaces using either distributed virtual memory or re-mapping. The choice of technique depends on circumstances. For example, a shared document that is under a joint editing session is mapped into two address spaces corresponding to the two editing sessions. Strict coherency is supported by the underlying virtual memory, usually based on page granularity that ensures that individual writes are serialised.

In contrast, when a CIDRE document needs to be moved between nodes as part of its circulation pattern then it is physically unmapped from one address space and later re-mapped where needed. As is often the case, there is a time delay between usage and so the document is automatically moved to stable store until re-accessed. It is then dynamically mapped into the new user's node and may be placed at any free address. As discussed above, because a document may consist of multiple clusters and because this group of clusters will contain

mutual references, mapping an initial cluster will cause the other clusters to be lazily mapped-in as they are accessed.

Clusters are persistent entities and when not currently active, i.e. there is no thread of control executing within a cluster, it is eligible to be mapped out to stable store. The COOL system implements a local garbage collection algorithm to ensure that unreferenced clusters are garbage collected. Clusters, once created, are automatically maintained within primary memory while they are referenced. Hence, the programmer is not forced to worry about the storage of important parts of a document, or the location of those documents when they have been stored for a period of time.

Access in COOL++ is fully transparent, objects are referenced through local virtual memory pointers irrespective of their physical location, the underlying GRT and base level deal with the actual location and invocation of objects. To allow applications to break that transparency, for example, if they need to physically move objects to a removable storage device we allow migration to be explicitly requested.

Each object may support multiple simultaneous invocations. In the CIDRE application synchronisation is achieved by explicitly using the underlying COOL synchronisation primitives. However, because COOL supports standard C++, a set of base synchronisation classes are generally inherited from to reduce the burden of explicit control.

The COOL exception mechanism provide a first level of support for reactive systems, exceptions can be associated with objects using an ad hoc exception extension to C++. We will incorporate 'standard' C++ exceptions when they are standardised.

5.1 Experiences with CIDRE

Using CIDRE to drive the development of COOL has resulted in several decisions that would otherwise have been different. The major influence CIDRE has had on COOL has been related to transparency. In our original design we had tried to produce a completely transparent programming model, thus location was hidden from programmers. During usage we found that although location transparency is often a benefit, there are cases where it needs to be broken. For example, physically co-locating an object with a piece of hardware it manages or requires. A further example concerns the usage of machines. In a typical computer science environment machines are never switched off, in a business environment they are often switched off at the end of the day[4]. In our original implementation of COOL, because of the distributed nature of the system, an object could be active and mapped at one machine, but its physical data stored on another machine. Switching off the storage machine would decouple the in-memory version from its stored version resulting in unpredictable behavior.

CIDRE also places requirements on the closed nature of the system. Since CIDRE objects can often interact with existing applications, including data bases and spreadsheets a means is needed to allow them to access other environments. While it would be possible to provide gateway objects between CIDRE and other systems, we benefited from the fact that COOL runs alongside Unix to allow us to use Unix functionality. For example, CIDRE objects can access Unix files by ensuring that application contexts run in actors that are both COOL actors,

[4] In fact in some cases, machines must be switched off overnight if the fire insurance regulations are to be met!

and Unix processes. In effect, a COOL context can trap into both COOL, and Unix. Thus for example, CIDRE can access X11 services providing it with an existing graphical interface service.

6 Related work

The scope of the COOL system is large, ranging from low level system mechanisms to high level language work. Our goal in the project has not been solely to investigate new techniques, but rather to synthesise existing techniques into a coherent platform. As such our work draws from many sources and has similarities to many systems. In this section we outline similar systems and discuss the differences.

At the virtual memory level, the original COOL v1 system, and some of the COOL v2 base system has similar features to Clouds[12], Amber[13] and Monads[14]. In particular the Clouds kernel, Ra, implements a simple micro-kernel that offers memory and threads in a similar way to Chorus and Mach. Memory is persistent and provides a single level store as in COOL-base. The essential differences between Clouds and COOL at the VM level is that COOL supports DSVM **and** an ability to re-map clusters into different parts of the address space. Further, COOL-base allows clusters to mapped simultaneously into multiple contexts, but does not force clusters to be seen in all contexts supported by the COOL system. The implementation of Clouds on Ra uses a simple object model similar to the COOL v1 approach. As reported in this paper, we abandoned this approach due to the costs and difficulties in supporting language level objects within the kernel.

The notion of a generic run-time is similar to the Portable Common Run-time [15]. Our experiences with the COOL v1 system led us to re-design the generic run-time to allow it to be specialised by language run-times and to allow interaction in both direction using down and up-calls.

COOL v2 was heavily influenced by the work carried out in the Comandos project, other implementations [17], [18], [19] have approaches similar to this work and illustrate the way in which the generic run-time can support multiple languages.

7 Conclusion and current status

The COOL project is building an object-oriented kernel above the CHORUS micro-kernel. Its aims are to provide a generic set of abstractions that will better support the current and future object oriented languages, operating systems and applications.

We currently have an initial COOL platform running above the CHORUS micro-kernel, running native on a network of ix86 based machines. This platform implements the basic cluster level including a minimal distributed virtual memory system. The COOL GRT offers full support for object distribution and for persistence. In addition we have built a pre-processor environment that allows us to generate pre-processor tools that can be used to extend existing languages such as C++ to take full advantage of the COOL v2 operating system interface.

Our premise is that the abstractions we provide at the lowest level will support both the model of construction for operating systems, and that of application level via the intermediary run-time levels. Our goal is to provide a flexible dynamic environment that will allow operating systems builders to easily build and add new functionality allowing the operating system to develop, in a coherent fashion, over time. To some degree we have achieved this goal, our experiences with COOL v1 and COOL v2 have led us to conclude that efficient system support is key if we want to encourage programmers to build distributed applications.

However, our experience has also shown that one of our major problems, that of reducing the mismatch between system mechanisms and language models still remains unresolved. The use of the up-call mechanisms between generic and language specific run-time has attempted to address the problem, but is still ad-hoc. A more promising approach, that of using meta information that can be used to control the behavior of classes, has been adopted in the Apertos project [16] and shows great promise. We hope to investigate how we can improve our system by adopting some of these ideas.

References

[1] R. Van Renesse.
Amoeba. In proceedings of Usenix Micro-kernels and other kernel architectures. April 27-28, 1992, Seattle, Washington. Usenix Association.

[2] R. Draves.
Mach. In proceedings of Usenix Micro-kernels and other kernel architectures. April 27-28, 1992, Seattle, Washington. Usenix Association.

[3] M. Rozier.
Chorus. In proceedings of Usenix Micro-kernels and other kernel architectures. April 27-28, 1992, Seattle, Washington. Usenix Association.

[4] Sabine Habert, Laurence Mosseri, and Vadim Abrossimov.
COOL: Kernel support for object-oriented environments. In ECOOP/ OOPSLA'90 Conference, volume 25 of SIGPLAN Notices, pages 269–277, Ottawa (Canada), October 1990. ACM.

[5] Deshayes, J.M., Abrossimov, V. and Lea, R.
The CIDRE distributed object system based on Chorus. Proceedings of the TOOLS'89 Conference, Paris, France. July 1989.

[6] The Integrated Systems Architecture project. ISA - Esprit project 2267. The ISA consortium, APM ltd, Castle Park, Cambridge, UK.

[7] V. Cahill, R. Lea and P. Sousa.
Comandos: generic support for persistent object-oriented languages. Proceedings of the Esprit Conference 1991. Brussels, November 1991. also Chorus systèmes technical report CS-TR-91-56.

[8] R. Lea and J. Weightman.
COOL: An object support environment co-existing with Unix. Proceedings of Convention Unix '91, AFUU, Paris France. March 1991.

[9] R. Lea and J. Weightman.
Supporting Object-oriented Languages in a Distributed Environment: The COOL approach. Proceedings of TOOLS USA'91, July 29-August 1, 1991. Santa Barbara, CA. USA.

[10] V. Abrossimov, M. Rozier and M. Shapiro.
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.

[11] R. Campbell and P. Madany.
Considerations of Persistence and Security in Choices, an Object- Oriented Operating System. Procs. of International Workshop on Computer Architectures to Support Security and Persistence of Information. May 1990, Bremen (Germany).

[12] 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

[13] J. Chase, F. Amador, E. Lazowska, H. Levy and R. Littlefield.
The Amber system: parallel programming on a network of multiprocessors
In proceedings of the 12th ACM symposium on Operating Systems principles, litchfield park, USA. December 1989.

[14] F. Henskens, J. Rosenberg and J. Keedy
A capability based distributed shared memory Proceedings of the 14th australian computer science conference, Sydney, Australia, 1991.

[15] M Weiser, et al.
The portable common runtime approach to interoperability
In proceedings of the 12th ACM symposium on Operating Systems principles, litchfield park, USA. December 1989.

[16] T. Tenma, Y. Yokote and M. Tokoro.
Implementing persistent objects in the Apertos operating system
In proceedings of the 2nd International Workshop on object orientation in operating systems (IWOOOS) 1992. Dourdan, France, Sept 1992. IEEE press.

[17] S. Krakowiak, A Freyssinet and S. Lacourte.
A generic object oriented virtual machine
In proceedings of the International Workshop on object orientation in operating systems (IWOOOS) 1991. Palo Alto, October 1991. IEEE press.

[18] M. Castro, N. Neves, P. Trancoso and P. Sousa
MIKE: A Distributed Object-Oriented Programming Platform on top of the Mach Microkernel
In Proceedings of the 3rd Usenix Mach Symposium April 19-21, 1993. Santa fey, New Mexico, USA.

[19] V. Cahill, Chris Horn and Gradimir Starovic.
Towards generic support for distributed information systems
In proceedings of the International Workshop on object orientation in operating systems (IWOOOS) 1991. Palo Alto, October 1991. IEEE press.

[20] M. Shapiro
Structure and Encapsulation in Distributed Systems: the Proxy Principle
Proceedings of the 6th ICDS Conference, Paris, France, May 1986

[21] Henry E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. A distributed implementation of the shared data-object model. In Proceedings of the Workshop on Distributed and Multiprocessor Systems (SEDMS), Fort Lauerdale FL, USA, October 1989. USENIX Association.

[22] P. Dasgupta, R. Ananthanarayanan, S. Menon, A. Mohindra, and R. Chen. Distributed programming with objects and threads in the clouds system. Computing Systems, 4(3), 1991.

[23] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter
Distribution and Abstract Types in Emerald, IEEE Transactions on Software Engineering Vol: SE-13 No.: 1, 1987, Pages: 65-76.

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