The COOL architecture and abstractions for object-oriented distributed operating systems

Rodger Lea, Christian Jacquemot

approved by:

distribution: COOL

abstract: Building distributed operating systems benefits from the micro-kernel approach by allowing better support for modularization. However, we believe that we need to take this support a step further. A more modular, or object oriented approach is needed if we wish to cross the barrier of complexity that is holding back distributed operating system development. The Chorus Object Oriented Layer (COOL) is a layer built above the Chorus micro-kernel designed to extend the micro-kernel abstractions with support for object oriented systems. COOL v2, the second iteration of this layer provides generic support for clusters of objects, in a distributed virtual memory model. It is built as a layered system where the lowest layer support only clusters and the upper layers support objects.

The COOL architecture and abstractions for object-oriented distributed operating systems

CS/TR-92-80

Chorus
SYSTEMES

Contents

1 Introduction

Building distributed systems is difficult simply because the complexity of interactions among entities scattered on a collection of machines is enormous. The distributed systems community has long been wrestling with this complexity and has developed methods such as RPC, group communications, distributed shared memory etc. in an attempt to provide mechanisms that abstract over some of this complexity. However, in attempting to build systems that actively use these mechanisms we have run into two major problems, performance and integration. Performance because we have tried to add these mechanisms to existing systems, and integration because we have tried to do so in an ad-hoc manner without fully considering how these tools should interact, or how applications will use these services.

Work in the operating system community has tried to deal with these issues by re-visiting our existing operating systems and looking at the minimum abstractions necessary to build distributed operating systems. By combining these with a systems building architecture that stresses modularity, we can begin to address the performance and complexity issues. This approach, often called the micro-kernel approach, allows us to provide a minimum set of abstractions that can be used to build operating systems themselves.

We feel however, that while this is the correct approach, it is only one step in the right direction. We need to augment our basic mechanisms with a framework that allows system builders to glue functional components together in a coherent and performant way. In effect, we need to provide a system building environment that supports a programming model, tools and services needed to work within that framework.

The object oriented paradigm offers a solution to this problem as it offers a framework for building large complex applications, such as OS's in a way that is amenable to distribution. However, we must not repeat the mistakes of early distributed system builders by trying to impose a model on a set of mechanisms, rather, we must actively support the model at the lowest layers in our system, by making sure that our abstractions are suitable for supporting objects.

In this paper we discuss how the COOL[1] system has been designed to exploit the unique features of the Chorus operating system model to provide an efficient set of abstractions that are well suited to support the object oriented metaphor. We stress that this approach not only facilitates building distributed OS's, but any distributed applications as it reduces the mismatch between our OO services and the model we use to build distributed applications. Our goal is to provide a framework that will allow operating system builders to develop their applications, the operating system, in a well structured, flexible and coherent environment.

[1] Chorus Object Oriented Layer

In Proceedings of 'Fifth ACM SIGOPS European Workshop on Models and Paradigms for Distributed Systems Structuring', September 21-23 1992, IRISA INRIA-RENNES

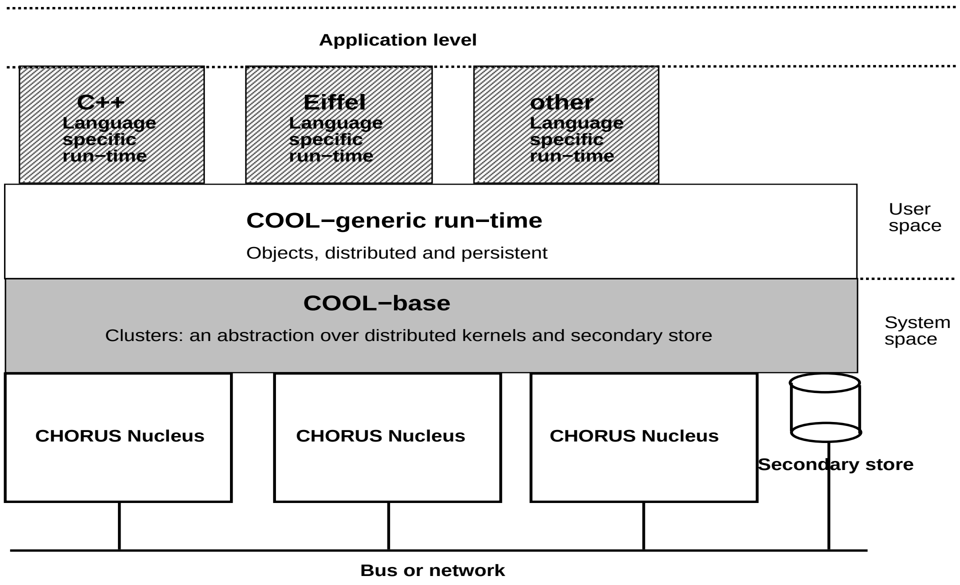
Application level
C++
Language specific run−time
Eiffel
Language specific run−time
other
Language specific run−time
COOL−generic run−time
Objects, distributed and persistent
User space
COOL−base
Clusters: an abstraction over distributed kernels and secondary store
System space
CHORUS Nucleus
CHORUS Nucleus
CHORUS Nucleus
Secondary store
Bus or network

Figure 1: The COOL v2 layered architecture

2 COOL v2

The COOL project is now in its second iteration, our first platform, COOL v1[2], was designed as a testbed for initial ideas and implemented in late '88 [2] [3] [6] [7].

Our early work with COOL (COOL v1) consisted with experimentation in the way that systems could be built using the object oriented model, and how this supported distributed applications. In an attempt to move the COOL platform from a testbed towards a full object oriented operating system we began a redesign of the COOL abstractions in 1990. This work was carried out in conjunction with two European research projects, both building distributed object based systems, the Esprit ISA project [4] and the Esprit Comandos project [5].

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

3 The COOL v2 architecture

COOL v2 is composed of three functionally separate layers, the COOL-base layer, the COOL generic run-time and the COOL language specific run-time layer (see Figure 1).

3.1 The COOL base

The COOL-base is the system level layer. It has the interface of a set of system calls and encapsulates the CHORUS micro-kernel [1]. It acts itself as a micro-kernel for object-oriented systems,

[2] COOL v1 was built as a joint project between Chorus systèmes, INRIA and SEPT

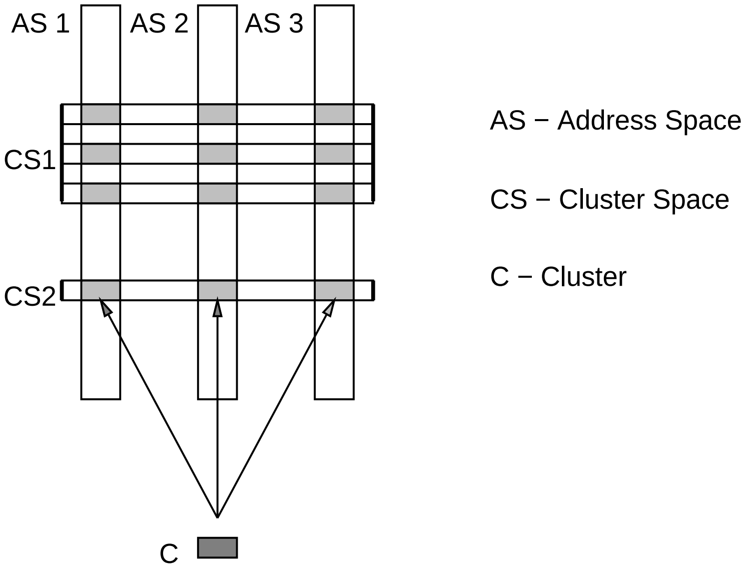

Figure 2: Clusters, clusters spaces and address spaces

on the top of which the generic run-time layer can be built. The abstractions implemented in this layer have a close relationship with CHORUS itself and they are intended to benefit from the performance of a highly mature micro-kernel.

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 (COOL v1) our base level supported a simple generic notion of objects. This proved to be too expensive in terms of system overhead so that 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 *cluster 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 [8]. The mapping can be done 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[3], regions and hence clusters, can be mapped into multiple address spaces, which leads us to the notion of cluster space (see Figure 2).

A cluster space is a collection of distinct address spaces on one or more nodes. The relationship between clusters spaces and address spaces is orthogonal, i.e., a cluster space can range over an arbitrary numbers of address spaces as well as contain many clusters. Any cluster belonging to a cluster space is mapped into all address spaces of that cluster space. In this case, we must enforce that the cluster is mapped always at the same address. Therefore, a cluster space represents a distributed virtual address space, and so can be used to share clusters among

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

threads of execution of a particular cluster space[4].

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 mechanism 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 mechanism 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 work described in [11]. The mapper is used to store and retrieve passive clusters to and from secondary storage.

Therefore, the COOL-base level supports a single-level, persistent cluster store with synchronous and asynchronous invocation between clusters, and distributed cluster sharing.

3.2 The COOL generic run-time

The generic run-time implements a notion of objects. Objects are the fundamental abstraction in the system for building higher level system servers or 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 generic run-time 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 cluster 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 address (a pointer in C++) and is valid in the context in which the object is presently mapped.

The generic run-time defines the primitives to convert one type of reference to the other one. When a domain wide reference to a remote object is converted to language reference a proxy associated to the object is created [12]. This proxy is used to transparently invoke the remote object.

Objects are always created in clusters. Each cluster's address space is divided into three parts: the first one is used to store all the structures associated to the cluster used by the generic run-time, the second one is used to store the applications objects, and the last one is used to store the proxies. A different allocator is associated to each part, this allocator is used to allocate and free space.

[4] Our initial implementation uses a memory mapper that supports strict memory coherency, we plan to investigate relaxed coherency later.

The classes are structured in modules (set of classes, unit of code). The generic run-time allows the code to be dynamically linked. The generic run-time offers a primitive to link a module. Each class contained in the module are store at the context level. 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.

The generic run-time provides an execution model based on the notion of activities which are mapped onto CHORUS kernel supported threads and job which model distributed execution of activities. Each cluster can support multiple activities, with more than one activity capable of running within the same object at any particular time[5].

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 mechanism that support multiple models we have defined a generic run-time to language interface based on upcalls.

The generic run-time maintains for each object a link between the object and its class. This link is used to find the upcall information associated with each object.

The upcall information, and associated functions is used for a variety of purposes, including support for persistence, invocation and re-mapping between address spaces. In fact, any time where a functionality of 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 upcall 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 is based on the standard method invocation of the language (C++ method). Invocations between objects in different address space use the model offered by the COOL-base layer (CHORUS communication primitives). The proxy 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. On the receiver, a dispatch procedure, which is part of the upcall function associated with an object is used to call the appropriate method on the appropriate object.

Invocation may also use the underlying cluster management mechanisms to map clusters into local address spaces for efficiency reasons, or locally to allow light weight RPC and maintain protection boundaries. Again the upcall functions are used to support this. This is further discussed in section 4.2.

[5] Subject to language level constraints.

3.3 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 and the use of the upcall table.

As discussed above, the generic run-time will, in the process of operations such as mapping or unmapping an object from an address space, upcall into the language specific run time responsible for that object by using the upcall table associated with the object and generated by the language specific run-time. This requires that the language run-time, usually the compiler, generates enough information to interface to the generic run-time. Currently we use pre-processor techniques to generate this information so that at run time objects can be managed by the underlying COOL system.

4 Main research areas

While the project covers a number of areas of interest in distributed, persistent systems, the architecture poses a number of problems at the lowest level.

4.1 Distributed memory model

Each cluster space represents a logical distributed address space, with each cluster mapped into a number of physical address spaces. The model makes a coupling between virtual memory addresses and object addresses only during the time that clusters are mapped. It makes no statement about the coupling between these addresses when a cluster is moved to persistent store. Thus we can support a model where a cluster always occupies a set of addresses and that range does not change when it moves between persistent store, or we may adopt a model whereby, the binding is only maintained while a cluster is mapped into a cluster space. Of course we need higher level support for relocation of objects within clusters (at the generic run-time level) if we adopt this approach.

Both of these models impose a criteria for distributed memory allocation, since allocating a new cluster requires that all machines in the cluster space allocate the same space. Currently we adopt a simple model where portions of an address space are initially allocated to different machines. Creation of clusters initially uses this space and uses a standard distributed virtual memory to ensure that the allocation is propagated to all machines represented in the cluster space. When a machine exhausts this initial space, it must arbitrate with others to allocate space from a common pool as is done in [10].

4.2 Single invocation model

The base level abstractions include an invocation mechanism that works between clusters. Invocation falls into one of three cases. Local invocation, ie that which stays within an address space. Invocation local to a machine but between address spaces, and standard remote invocations (RPC). In a persistent, distributed system, there are a number of possibilities, when invocation takes place, concerning the location of the object.

In particular, the interaction between the invocation model and the cluster model provides us with the ability to optimize invocation:

- For a cluster that is held in persistent store, the cluster is mapped into the calling cluster space.
- For clusters mapped into an existing cluster space, instead of using an RPC call, we are able to de-map the cluster and re-map it into the calling cluster space, or into a cluster space on the same machine allowing us to use the light weight form of the standard RPC call.
- Finally, if no address space or protection clashes exist we can extend the cluster space to incorporate the cluster in its current location. The distributed virtual memory will then make the cluster available to the calling cluster.

COOL-base is capable of using this range of mechanism to carry out the invocation. The choice of mechanism will be dependent on higher level policy, but a simple approximation, using invocation efficiency as a criteria allows us to build a lightweight, default policy into the base level.

4.3 System building model

We have tried in the COOL system to provide a minimal set of abstractions that are suited to building applications using the object oriented model. One of our stated goals is also to build a platform that allows operating system builders to construct system services in the same way as end user applications would be constructed. Thus for example, elements of the generic run-time are built using objects who interact using serv¡ices provided by the core of the run-time. Some aspects of this are similar to work carried out in [9]. However, Choices concentrates more on the functional objects that make up the operating system and their relationship at build time. Our emphasis is how those objects are supported and interact at run-time.

This provides a great benefit in that operating system services can be invoked, migrated between address spaces, moved to persistent store, etc, in the same way as user objects. This model provides us with the ability to dynamically add services to an existing operating system and to reconfigure our operating system at run-time to exploit hardware or to adapt to specific user needs.

5 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.

Our premise is that the abstractions we provide at the lowest level will support both the model for constructing operating systems, and for developing applications via intermediary run-time levels. Our goal is to provide a flexible dynamic environment which allows operating system builders to easily build and add new functionalities into the operating system, in a coherent and fully modular approach.

We currently have a limited COOL platform running above the CHORUS micro-kernel, native on 386 based PC's. This platform implements the basic cluster level including the distributed virtual memory support but still lacks light weight RPC. The COOL generic run-time offers full support for object distribution and for persistence. In addition we have built a pre-processor environment which generates 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.

References

[1] Marc Rozier, Vadim Abrossimov, Francois Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, Will Neuhauser
CHORUS Distributed Operating Systems
Computing Systems Journal, Vol 1,No 4, December 1988, USENIX Association

[2] 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.

[3] 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.

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

[5] Vinny Cahill, Rodger Lea and Pedro 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.

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

[7] Lea, R. and Weightman, J. 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.

[8] 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.

[9] Campbell, R. H. and Madany, P. W. 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).

[10] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield
The Amber System: Parallel Programming on a Network of Multiprocessors
ACM SIGOPS, Litchfield Park, AZ, December 1989

[11] Partha Dasgupta, R Ananthanarayanan, Sathis Menon, Ajay Mohindra, Raymond Chen
Distributed Programming with objects and Threads in the Clouds System
Computing Systems, Vol 4, No 3, Summer 1991, USENIX Association

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