Implementing a modular object-oriented operating system on top of Chorus

This content has been downloaded from IOPscience. Please scroll down to see the full text.

1993 Distrib. Syst. Engng. 1 11

(http://iopscience.iop.org/0967-1846/1/1/002)

View the table of contents for this issue, or go to the journal homepage for more

Download details:

IP Address: 52.26.76.46
This content was downloaded on 17/07/2015 at 20:59

Please note that terms and conditions apply.
Implementing a modular object-oriented operating system on top of Chorus

Rodger Leat, Paulo Amaral and Christian Jacquemot
Chorus Systèmes, 6, avenue Gustave Eiffel, F-78182 Saint-Quentin-en-Yvelines Cedex, France

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 that 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. This approach allows us to build operating systems as collections of objects.

1. Introduction

Distributed 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 [3, 6, 12].

While this approach has demonstrated that object-oriented languages provide sufficient support for building distributed applications in many cases it has suffered from poor performance [7]. Often this is 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 operating 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 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, [5, 6].

While we feel that this approach is correct, it does not go far enough. System support needs to be both efficient and flexible, further 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 architecture such as Chorus [21], Amoeba [25] and Mach [14] 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.

(i) To provide a set of generic services that reduce the mismatch between system abstractions and language abstractions.

(ii) 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 (known as the Nucleus) with abstractions more suited to object-oriented systems.

(iii) To provide these services in a way that they can be tailored by higher levels, and can be replaced or extended in a dynamic and flexible manner.

This paper is structured as follows; we introduce the COOL v2 architecture, discussing its functionality and layered design. We then discuss the different layers concentrating on the lowest levels of the COOL system, detailing the distributed memory model and its implementation with particular reference to how we have used the basic micro-kernel functionality to build this operating system service. In section 4 we discuss related work and in the final section we conclude and discuss future directions.

2. COOL v2

The COOL project is now in its second iteration, our first platform, COOL v1†, was designed as a testbed for initial ideas and implemented in late 1988 [15, 19, 20]. Our initial platform implemented a simple notion of an object as a micro-kernel supported abstraction, with mechanisms to instantiate 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 distributed office environment [13] system during the development of which we identified several problems but gave us considerable insight into how such an object-oriented operating system can be built above the Chorus micro-kernel.

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 [24] and the Esprit Comandos project [8], both building distributed object based systems.

COOL v2 is composed of three functionally separate layers (see figure 1), the COOL base layer, the COOL generic run-time (GRT) and the COOL languagespecific run-time layer.

Our goals when designing this architecture were twofold; efficiency and flexibility. To ensure efficiency we built a basic set of system services directly above the micro-kernel. These services are generic enough to support higher level notions of objects without requiring that the lowest levels in the system supported very specific object semantics. In addition, since the services were implemented as a set of extensions to the Chorus micro-kernel we can use a feature of the Chorus architecture to dynamically load these services into system space thus reducing context switch and data copying time.

Secondly, we designed the architecture so that it would be easy to specialize both system components, and how these components were used. For example, the lower levels of COOL support three separate mechanisms for object interaction, a simple invocation mechanism, an ability to migrate objects, and an ability to share objects using a distributed shared memory mechanism. Each of these mechanisms have their advantages and their drawbacks, and each will be used in different circumstances. A key element of our work is that the three levels of the architecture interact to provide all three mechanisms, allowing policy to decide which mechanisms to be used for a particular object model, or even mixing their use depending on applications.

3. 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. It supports the following base services; a virtual memory abstraction where objects can exist which abstracts both remote machines and secondary storage; support for object sharing through distributed shared memory, message passing and migration; an execution model based on threads and a single level persistent store.

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 the 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 [1]. 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, regions and hence clusters, can be mapped into multiple contexts, termed a context space.

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 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 re-mapped 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.

Finally, a group of related clusters is collected into a container. A container provides a structure over cooperating clusters defining a group of objects whose external references are 'contained' solely within the clusters of the container.

It is important to note that although a cluster is mapped into several contexts at one time, it 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 un-map the cluster and map it back in at a different set of addresses.

3.1. Supporting Interaction

To allow objects to interact from within clusters we support three basic mechanisms.

The first is 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 light-weight invocation mechanism, whereas between clusters on different sites we use a traditional invocation model.

A second means of interaction is based on the ability to map clusters into and out of contexts on demand. This mechanism can be used to force 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 space. 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.

The third mechanism makes use of the virtual memory architecture to provide a distributed shared memory mechanism. Each container is ultimately mapped to one or more Chorus segments, the unit of secondary storage. When mapped, a container is said to have a view. The view represents this mapping from secondary storage segments to primary storage regions. More than one view may be managed by a context at one time, allowing multiple containers to be mapped into a single context (see figure 3). The management of distributed views of a container is carried out by the base level and allows more than one context to view a container, we refer to this as a context group. A context group is a set of contexts that support one or more containers. Each container is mapped at exactly the same set of addresses in each context in the group.

It is possible, and indeed likely, that a context will support more than one container at a time. Hence, contexts may belong to multiple groups at any one time with parts of their address space 'allocated' to different groups. A group of contexts that map a particular container is said to support a persistent context space, a distributed, persistent address space from the containers point of view.

The management of these persistent context spaces requires some form of distributed control. There are
several aspects to this; containers which wish to diffuse to new contexts need to know if that context is capable of supporting the container, e.g. if addresses used by the container are already allocated then the diffusion cannot be carried out. When new virtual memory is added to a container, then allocation must be carried out across all containers in the group, this is managed by the distributed view control mechanisms. In both cases it is necessary to build mechanisms that work in the distributed system and our aim is to exploit the support that the Chorus micro-kernel provides for dealing with distribution.

### 3.2. Implementation Structure

To manage these distributed entities, the COOL base is composed of several objects, or fragments; represented on each site and each using the underlying Chorus mechanisms to implement a distributed algorithm. In figure 4 we outline the overall implementation architecture of the COOL system. We exploit the Chorus system building architecture to build our system as a set of cooperating servers. You will note that in fact, COOL runs as a second personality alongside an implementation of a more traditional operating system; UNIX. The multi-server version of UNIX, called Mix, is built of four main servers who export a traditional UNIX ABI. As can be seen, the COOL system is actually built from servers that run in both user and system space with the name server and mapper running as traditional UNIX processes. We have adopted this approach for ease of development, servers running in user space can access the UNIX ABI through the standard trap mechanism thus providing access to services like the files without needing to re-implement existing functionality. This approach can significantly reduce the development time of a new operating system because it allows the builders to concentrate solely on their abstractions. More details of the Chorus/MiX system can be found in [4].

The COOL base level is implemented using three servers all executing in system space, each server interacts with other servers, both locally and remotely through the Chorus IPC primitives. These three servers are:

- **base object**: contains all state information about the local site;
- **base proxy**: transparently addresses the correct base object whenever a request for some system action must be re-directed to another site;
- **base server**: transparently forwards incoming remote requests to the local fragment responsible for managing that resource.

Each COOL base server implements three protocols:

- **Distributed group management**: creates and deletes groups, attaches and detaches contexts from groups and controls address space allocation;
- **Distributed view management**: attaches and detaches views to and from clusters and informs the COOL base to raise an up-call whenever these operations can influence the use of the data stored inside the cluster;
- **Distributed cluster management**: creates, deletes, activates and de-activates clusters; it is also responsible for adding and deleting segments to/from clusters.

Each of these protocols are internally implemented as a separate thread of control using the Chorus kernel supported thread abstraction. Further, to allow communication between separate threads of a particular protocol which are running on different sites, we use standard Chorus IPC. In some cases, individual protocols will require information that only the higher levels of the operating system can provide. Since higher level code, i.e. the generic run-time, executes in user space, we implement an upcall mechanism, again based on Chorus IPC, that allows the base level to call into the GRT to access relevant information. One of the significant advantages of this approach is that we can upcall both locally, and if required, into the generic run-time of a remote node.
3.2.1. Persistency support. Persistent memory is organized in containers as explained above. Each container is further subdivided in clusters; a cluster being a set of persistent segments.

Each entity managed by the system layer is named using a Chorus capability, which uniquely names it in the distributed system. Capabilities are the means to manage system entities and are passed between servers, any server with a capability for an entity can carry out operations on that entity and can use the capability as an end point for communication. In the case of clusters and containers; both virtual memory based entities, capabilities are managed by mappers. Each mapper is designed to manage the relationship between secondary storage and main store. When a request to use a cluster is generated, the COOL base hands off the request for an unmapped cluster to the mapper managing that cluster. This hand off is again based on Chorus transparent IPC and works by allowing capabilities to represent not only single end points for a communication but groups. When a cluster is mapped out to secondary store its capability is associated with a group whose other members include a manager. Communications sent to the cluster via its capability are actually received on its behalf by the manager (the mapper) who then deals with locating and mapping in the cluster. This IPC mechanism thus enables us to distribute the mappers at any location in the system, and to move them between user and system space as required.

Persistency of clusters is also managed by the mapper. In conjunction with higher level tools such as garbage collectors, the mappers decide which clusters are referenced and will always ensure that such clusters are mapped out into secondary store where they will remain until referenced again.

The capability assigned to the cluster, and managed by the mapper is guaranteed to remain unique during the lifetime of the system. This guarantee is made by the underlying Chorus micro-kernel which is responsible for generating capabilities.

3.2.2. Cluster mapping. An application starts to run within a single context. Initially a single container will be associated with this application, with a minimum of one cluster mapped into the context.

Mapping clusters can be viewed as a three tiered process, at the highest level a container fault occurs which sets up a list of clusters that may need to be mapped. Access to a memory address will then cause a particular cluster to be mapped which will set up the relationship between the cluster and its secondary storage segment. Finally, in a page-based architecture, each segment is divided in pages, so it is only effectively read to in-core memory if it is really accessed (in a third level fault).

After mapping, memory contents may need to be relocated. This is dependent on the semantics of memory contents and only application levels are aware of it. The relocation itself is based on symbolic information and only known symbols can be relocated. The problem is that there may be pointers that have no correspondent symbols generated normally by the compilation chain. So, special high level run-time code has to exist in order to access intrinsic semantic information of memory contents at user level in a transparent manner. The base level causes the run-time level to carry out any relocation required when the cluster is mapped into a context for the first time by upcalling into the run-time.

3.2.3. Cluster un-mapping.

This upcall mechanism can also be considered an exception (but a distributed one). As we saw, cluster mapping and relocation is done automatically by the base and the run-time system when a memory fault occurs. In the simple case, mapping is carried out from secondary storage on an inactive cluster. However, it is likely that a cluster is already in use as part of another persistent context space. Hence it is necessary to force that cluster, and its container, to be un-mapped from one persistent context into the faulting one.

An upcall has to be issued from the kernel to the run-time system in order to un-map the cluster transparently from the application contexts. This upcall is performed using the Chorus communication mechanisms allowing the upcall to work in the distributed system.

3.2.4. Mutual exclusion. With the exception and upcall mechanisms in place it is straightforward to assure mutual exclusion of clusters that need to be mapped at different addresses, i.e., that belong to different active persistent context spaces.

During memory fault handling, if the system sees that a cluster is being used by another persistent context space it up-calls all contexts in that context space to force the un-mapping, and proceeds. Later on, if any one of the other contexts needs that cluster again, it will do exactly the same in the inverted sense.

After re-mapping a cluster, the system has to verify if the container information about that cluster is still valid. The container may have a different set of clusters or belong to another persistent context. This has to be done immediately after cluster mapping because it may now directly reference another cluster after being changed in the persistent context space where it was previously mapped.

3.2.5. Shared memory coherency. Memory mapped in a context space needs to be assured single-reader multiple-reader coherency between all distributed contexts that have it mapped into its address space. A distributed shared memory system such as proposed in [2] is used. This is a strict coherency algorithm but is well suited to the semantics of languages such as C++. We are currently investigating weak coherency support.

† Remember that a container is made up of one or more clusters. Clusters are the unit of mapping.
3.3. The COOL generic run-time

Given that the COOL base supports a set of generic mechanisms that form a basis for the development of an operating system that supports an object model, we can now discuss how we use these mechanisms to provide object support. This support is provided by a run-time layer that is sufficiently generic to support multiple languages at the application level.

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 re-mapped in another address space.

3.3.1. 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 created under application control and launched within a particular object. The activity is then capable of diffusing to other objects, 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.

3.3.2. The up-call mechanism. As discussed above, we use an upcall mechanism to allow the base level to access semantic information in the GRT. In addition to this we also use an upcall mechanism to allow the GRT to access information in language specific run-times. This mechanism is 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 cluster persistence, and hence object persistence, we need access to the layout of objects to locate references held in the object 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 swizzling.

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. 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 un-marshals 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.

3.4. 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 to generate 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 interested reader is referred to [18] for more details of the language specific run-time for C++.

4. 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 synthesize 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 [11] and Monads [16]. 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 be 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 [26]. Our experiences with the COOL v1 system led us to re-design the generic run-time to allow it to be specialized by language run-times and to allow interaction in both direction using down and up-calls.

Our work has many similarities to that of the Apertos project [23] although we have eased the development task by building above an existing micro-kernel rather than from scratch. One of our stated aims was to support flexibility throughout the system, and part of our approach to this has been the use of the up-call model. Apertos has taken a more comprehensive approach, that of the meta-space architecture which are now evaluating to understand if we can combine the two.

COOL v2 was heavily influenced by the work carried out in the Comadex project, other implementations [9, 10, 17] have approaches similar to this work and illustrate the way in which the generic run-time can support multiple languages. However, our work differs in that we have concentrated on using the object oriented model as a means to build not only applications, but the operating system itself.

5. Conclusion and current status

The Chorus micro-kernel is a set of low level functionality on which higher level systems can be built. After four years of experience using it to build object oriented operating systems we are convinced that micro-kernels are a sensible approach to reduce system complexity and the development cycle.

Our experience showed that much of the work in implementing a distributed system goes into the maintenance of distributed state. We used an object-based system to describe distributed state with fragmented objects. The use of the Chorus micro-kernel allowed the implementation of these fragmented objects in a natural manner using a set of protocols over Chorus IPC based on a distributed capability-based naming scheme that Chorus supports.

We currently have a initial COOL platform running above the Chorus micro-kernel which is in turn running native on a networked 386 based machine. This platform implements the basic cluster level including the distributed virtual memory support. 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 preprocessor tools that can be used to extend existing languages such as C++ to take full advantage of the COOL v2 operating system interface.

Acknowledgments

We would like to thank our colleagues at Chorus Systèmes for their valuable input to this work, in particular, Peter Starup Jensen and Adam Mirowski.

References

platform on top of the Mach microkernel Proc. 3rd Usenix Mach Symposium (Santa Fe, NM) (Berkeley, CA: Usenix Association)


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