Implementing a modular object oriented operating system on top of CHORUS

Paulo Amaral, Rodger Lea and Christian Jacquemot

Chorus systèmes
6, avenue Gustave Eiffel
F-78182 Saint Quentin-en-Yvelines Cedex
France

Tel: +33 1 30 64 82 00
Fax: +33 1 30 57 00 66

ABSTRACT

Building distributed operating systems benefits from the micro-kernel approach by allowing better support for modularization. However, we believe that 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. It is built as a layered system where the lowest layer support only clusters and the upper layers support objects.

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 system 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 by offering 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[Bla92]

In this paper we discuss how the COOL 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 object oriented application, because 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.

We will introduce the basic COOL v2 architecture, and then concentrate on the Persistent Context Space model that we have developed to allow us to efficiently support distributed, shared objects at the lowest layer in the system.

2. COOL v2

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 [Hab90, Des89, Lea91]

Our early work with COOL (COOL v1) consisted of 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 and the Esprit Comandos project [Cah91]

The result of this work has been the specification of the COOL v2 system and its initial implementation in late '91, [Lea92, Ama92].

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.

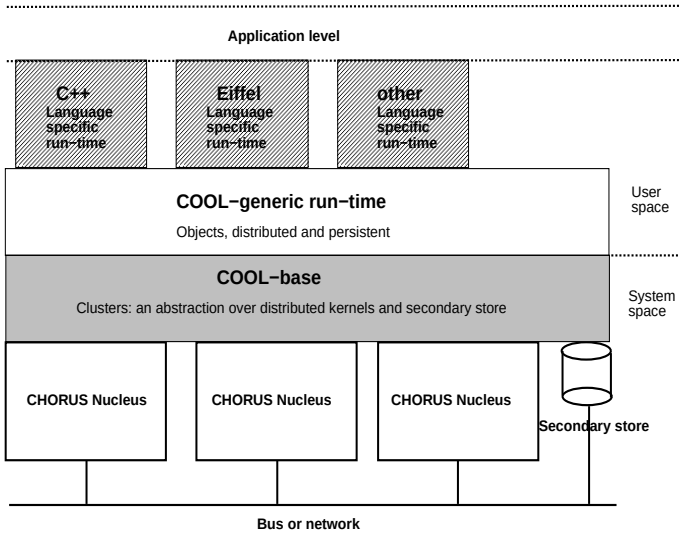

Figure 1: COOL v2 architecture

[1] COOL v1 was a joint project between Chorus Systemes, the SEPT (Service d'Etudes des Postes et Telecommunications), and INRIA (Institut National de Recherche en Informatique et en Automatique)

Our goal when designing this architecture was twofold, efficiency and flexibility. We wanted to support distributed interactions using a number of base mechanisms.

To allow objects to interact we can;

- support a communications mechanisms that will allow transparent invocation.
- allow objects to migrate between contexts by unmapping from one context and mapping into another, relocating internal pointers on the fly.
- use a distributed shared memory mechanisms that ensures object level faulting.

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 at which particular time.

In the following sections we briefly outline the functionality of the three levels and then return to the base level and explain further its support for a distributed virtual memory model and its implementation as a distributed system support layer.

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. It acts itself as a micro-kernel for object-oriented systems, 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 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 objects out of our base layer and replaced it with a more generic set of abstractions which we term the *Persistent Context Space* model (PCS).

The persistent context space supports a basic abstraction, the *cluster* which is a set of virtual memory regions and provide a repository for objects. Clusters, being persistent, are represented on secondary storage using the CHORUS abstraction of a segment, and are represented in memory using the CHORUS abstraction of regions.

Clusters are grouped together into *containers* which represent collections of objects whose references are completely contained, i.e. all references within clusters point into clusters within the same container.

A *context* abstracts the notion of an address space, and provides a place into which containers can be mapped for execution. To support distributed shared memory we define the *context group* which is a collection of contexts, on one or more sites, that map identical containers.

We will return to the PCS model and its implementation in section 4

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 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 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 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 [Sha86]. 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 with 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.

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 runtime provides an execution model based on the notion of *activities* which are mapped onto CHORUS kernel supported threads and *jobs* which models 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[2].

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

The generic runtime 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 located 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 an remote invocation which marshalls the parameters, issues an remote procedure call, and unmarshall the results. At 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 bounderies, again the upcall functions are used to support this.

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 GRT 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

---

[2] Subject to language level constraints.

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. The Base Level revisited.

In section 3.1 we briefly outlined the abstractions that the base level provides, however the container/cluster mechanism is designed to support more than a simple grouping of objects.

Our goals when designing the base abstractions where:

- Support distributed, shared virtual memory so that we could efficiently support languages based on virtual memory references.
- Provide a form of memory persistence including the mechanisms for a single level store so that higher levels would not see a multi-tiered storage hierarchy.
- Provide a means to structure the distributed virtual memory space so that system builders can control their use of the distributed virtual memory.

The mechanisms that form part of individual language run-times and the GRT support distributed programming, however, in all cases they are costly. Object relocation requires pointer swizzling when clusters are mapped and unmapped; invocation using a message passing mechanisms need parameter marshalling and often break the semantics of object invocation.

Supporting a distributed, shared virtual memory is one solution that allows efficient transparent programming within a distributed environment. Although there are many costs and restrictions to a distributed virtual memory model, when combined with a complete system that supports other mechanism, such as mapping and remote invocation, it offers a powerful tool. A key difference between our work and others is that we offer a range of mechanisms to support distribution, not just one.

Our basic unit of distribution at the base level is the container, which, as described in section 3.1, is made up of several clusters.

Containers are lazily mapped from secondary storage, by the base level into a virtual address space, or context. This mapping my involve relocation, as the form held on secondary storage may store pointers in a global format[3].

[3] A vanilla language mapped onto the GRT, without language run time support will not be able to support relocation and will be constrained to always be located at a particular set of addresses. An extended language, that was designed to exploit the GRT would allow addresses to be relocated, thus allowing the system to relocate containers as required.

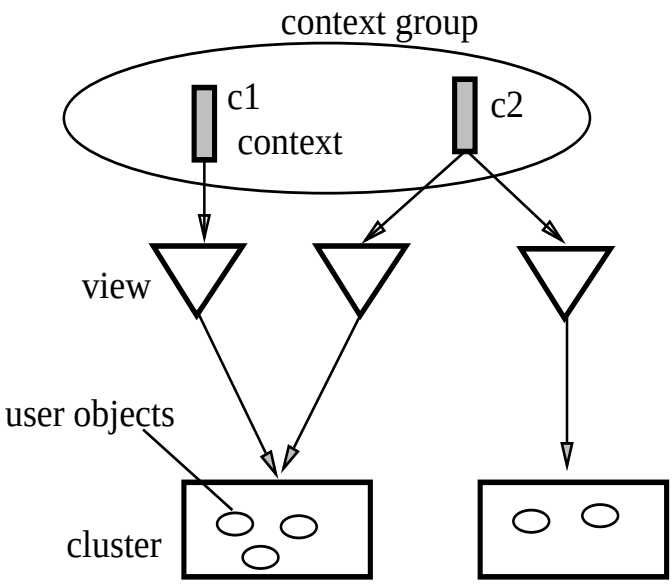

Figure 2: COOL-base fragmented objects

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 2. The management of distributed views of a container is carried out by the base level.

A container, once created will remain in the system as long as there are references to that container. This persistence is managed by the base level.

Once mapped, objects within the container can carry out invocation using virtual memory references. If that activity wishes to diffuse to other sites, for example to allow physical parallel activity, then we create 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 as 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 can not be carried out[4]. 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.

4.1. COOL-base implementation structure

To manage these distributed entities, the COOL-base is composed of several objects, or fragments[5] represented on each site and each using the underlying CHORUS mechanisms to implement a distributed algorithm. These objects are grouped into three major components:

---

[4] It may be possible to remap the container to a new set of addresses compatible with the new context.

[5] We use the term fragment, because each local representative, is a part of a global distributed, or *fragmented*, object.

- base object: it contains all state informations 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 (with one CHORUS thread per protocol):

- 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 upcall whenever these operations can influence the use of the data stored inside the
- Distributed cluster management creates, deletes, activates and de-activates *clusters;* it is also responsible for adding and deleting segments to/from *clusters.*

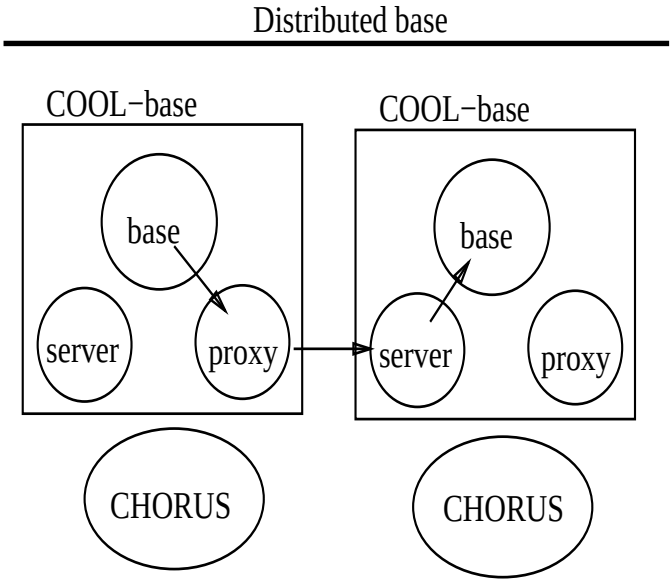

Figure 3: COOL-base fragmented objects

The protocols are mapped directly on to CHORUS IPC. Some operations have to upcall the generic run time to update upper layer state information. This also uses the CHORUS IPC, allowing us to upcall both locally and remotely.

4.2. 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. 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 system hands of the request for an unmapped cluster to the mapper managing that cluster. The mapper is responsible for locating the secondary storage representation of the cluster, and will understand enough of this format to allow it to map the cluster into primary store.

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.

4.3. The exception mechanism for 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[6].

An exception mechanism exist that uses memory faults to map the correct memory at the right place. The first container mapping can be considered the highest level fault. It makes visible the next level in the structure, that is, clusters.

The second exception level is the segment fault: upon an access to some unmapped memory address, the exception handler verifies if there is some existing cluster, part of the current complete memory space, that contains a segment with the needed address when mapped. It then maps the right cluster. 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 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.

4.4. The upcall mechanism for unmapping

The 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 unmapped 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 unmap 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.

4.5. Combining exception and upcall mechanisms to assure mutual exclusion

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

During memory fault handling, if the system sees that a cluster is being used by another persistent context space it upcalls all contexts in that context space to force the unmapping, 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 remapping 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.

---

[6] Remember that a container is made up of one or more clusters. Clusters are the unit of mapping.

4.6. Shared memory coherency

Memory mapped in a context space needs to be assured single-writer multiple reader coherence between all distributed contexts that have it mapped into its address space. A distributed shared memory system such as proposed by Li[App91] 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.

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.

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 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 limited COOL platform running above the CHORUS micro-kernel, running native on 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 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.

6. Acknowledgments

We would like to thank our colleague at CHORUS systems for their valuable input to this work, in particular, Peter Strarup Jensen and Adam Mirowski.

References

Ama92.
Paulo Amaral, Rodger Lea, and Christian Jacquemot, ‘‘A model for persistent shared memory addressing in distributed systems,’’ in *Proceedings of the International Workshop on on object orientation in operating systems*, IEEE Computer Society, Dourdon, France, September 1992.

App91.Andrew W. Appel and Kai Li, ‘‘Virtual Memory Primitives for User Programs,’’ in *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 96-107, Santa Clara, CA USA, April 1991.

Bla92.Gordon Blair and Rodger Lea, ‘‘The impact of distribution on the object-oriented approach to software development,’’ *IEE Software Engineering Journal*, vol. 7, no. 2, March 1992.

Cah91.Vinny Cahill, Chris Horn, Gradamir Starovic, Rodger Lea, and Pedro Sousa, ‘‘Supporting Object Oriented Languages on the Comandos Platform,’’ in *Proceedings of ESPRIT’91 Conference*, Brussels, Belgium, November 1991.

Des89.Jean-Marc Deshayes, Vadim Abrossimov, and Rodger Lea, ‘‘The CIDRE Distributed Object System Based on CHORUS,’’ in *Proceedings of TOOLS’89*, p. 8, 1989.

Hab90.Sabine Habert, Laurence Mosseri, and Vadim Abrossimov, ‘‘COOL: Kernel support for object-oriented environments,’’ *SIGPLAN Notices*, vol. 25, pp. 269-277, 1990.

Lea91.Rodger Lea and James Weightman, ‘‘COOL: An object support environment co-existing with Unix,’’ in *AFUU convention UNIX ’91*, p. 13, July 1991.

Lea92. Rodger Lea and Christian Jacquemot, ‘‘The COOL architecture and abstractions for object oriented distributed operating systems,’’ in *Proceedings of the 5th ACM European SIGOPS*, Mont Saint-Michel, France, September 1992.

Sha86. Marc Shapiro, ‘‘Structure and Encapsulation in Distributed Systems: the Proxy Principle,’’ in *Proceedings of the 6th ICDS conference*, May 1986.