Extending the Operating System to Support an Object-Oriented Environment*

José Alves Marques and Paulo Guedes

INESC - R. Alves Redol n°9 1000 Lisboa PORTUGAL
mcvax!inesc!{marques,guedes}

Abstract

Comandos is a project within the European Strategic Programme for Research on Information Technology - *ESPRIT* and it stems from the identified need of providing simpler and more integrated environments for application development in large distributed systems.

The fundamental goal of the project is the definition of an integrated platform providing support for distributed and concurrent processing in a LAN environment, extensible and distributed data management and tools for monitoring and administrating the distributed environment.

An object oriented approach was used as the ground level for the integration of the multidisciplinary concepts addressed in the project.

This paper starts by describing the basic model and architecture of Comandos, which results from a common effort by all the partners in the project. We focus then on the description of a first prototype of the system, which implements a subset of the architecture and is currently running on a set of personal computers and workstations at INESC. The prototype is a testbed for the architecture, providing dynamic linking, access to persistent objects and transparent distribution. Special attention is given to the performance aspects of object invocation in virtual memory.

1 Introduction

The fundamental motivation for Comandos is to define and implement an integrated platform able of supporting transparent distribution and persistent programming, together with a set of tools for monitoring and administrating the system. The platform can be viewed as an extension of the computational model offered by traditional operating systems towards a more comprehensive, uniform and integrated model.

The difficulties application programmers have when forced to interact with the operating system are widely known. They become faced with a totally different computational model and type model, normally more primitive than the framework of description available in programming languages. In a paper on the principles behind Smalltalk, Ingalls [17] expresses his view on the operating system "an operating system is a collection of things that do not fit into the language, there should not be one". Although sound in principle such radical opinion must be weighted against one major aspect in computer industry: compatibility with existing software. In particular all the Unix[1] environment must be considered in any proposed evolution of the operating system interfaces.

Although deeply influenced by the ideas proposed in object oriented programming, Comandos aims at defining an operating system platform that we hope will provide support for multiple languages and will be able to offer a Unix interface and interwork with standard Unix applications, rather then just defining a new programming language.

Some of the initial requirements for the Comandos platform were:

- *Transparent distribution.* To provide uniform access to any entity in the system, independently of its location and implementation.
- *Uniform treatment of volatile and persistent entities.* To provide uniform access to all entities in the system whether active in memory or stored in disk. To promote the integration of the usually dissociated environment for persistent programming. To have all persistent data in the system under a

*This work was partially supported by CEC under Esprit contract n° 834 and JNICT under contract n° 87634

[1]Unix is a registered trademark of AT&T

common model and management would allow considerable simplification in application development.

- *Upward compatibility.* The possibility of extending existing services or applications without having to modify working programs.
- *Consistent sharing of data.* Transparently enforcement of consistency rules on shared objects.
- *Intrinsic fault tolerance mechanisms.* Support for fault tolerance will be increasingly more important as systems become more complex and handle vital data. Fault tolerance mechanisms should be embedded into the system and not considered as an afterthought to handle some special data.
- *Support for heterogeneity.* Heterogeneity both in hardware and software must be supported. This means to deal with the classical issues in distributed systems, like different data representation and communication protocols.
- *Support for standards.* The platform should incorporate all relevant standards in the area of communications, but also standard operating system interfaces such as X/Open.

After the initial phase, that corresponded to the definition of the type model and the computational model, we launched amongst the partners several implementations, which were considered as experimental testbeds to evaluate the basic ideas of the architecture [21,12,16]. In this paper we start by giving an overview of the Comandos virtual machine and its supporting architecture. A particular emphasis is given to the structure of objects and its relation with the programming environment and execution environment. The prototype developed at INESC is presented and some preliminary performance results are discussed.

2 Comandos Virtual Machine

We decided to subdivide the model in three main entities which account for the definition of the Comandos virtual machine: a *type model* defining the entities the system is able to manipulate, a *computational model* that specifies the way objects become animated and interact, and a *virtual architecture* providing a functional decomposition of the main entities required to support the previous models.

2.1 Type Model

One of the major problems in designing a large software system is organizing the complex lattice of relationships between objects in applications. Types provide support for modularization and means to establish a classification of similar objects, which allows to capture their common characteristics [14].

A type defines the interface an entity presents to the exterior. A type definition may be considered as the protocol an object known by the system is able to handle. An initial decision of the project was to privilege, whenever possible, static type checking for early detection of programming errors and to improve code efficiency. Similar approaches are followed at the language level in Emerald [6], POOL-T [2] and Trellis/Owl [25].

In Comandos an object has an interface defined by its *user defined type.* A user defined type represents the external interface of an object and may contain operations and properties (variables) which are directly accessible from external objects. Types are abstract data types when no properties are present in the interface, or merely aggregates of data types when no operations are defined.

A user defined type is just an interface specification and may be associated with (possibly) several implementations. Each implementation is defined by an *implementation type* allowing to have different algorithms, or possibly different code, for heterogeneous machines associated with the same interface.

Objects are instances of implementation types and have an internal state, or instance data, and a number of operations. Instance data is subdivided in external visible properties and private data, which can only be manipulated by invoking operations on the object.

Objects are manipulated through references. A reference allows to access the properties and invoke the operations defined in the type to which it is associated.

2.2 Computational Model

Active Entities

In a system supporting persistent programming most of the objects will be passive data structures. Therefore, we choose to separate explicitly active and passive objects. The difference between the two being that active objects may change their internal state independently of any invocation. Active objects exist in the form of *jobs* and *activities.* A job may contain one or many activities executing in parallel. Activities are distributed threads of control, similar to processes in a centralized system. Jobs and activities are distributed objects which may span over several nodes. A job may be considered as a multi-process, multi-node virtual machine. During its execution a job is a varying collection of objects. In each node visited by any of its activities, the job maintains a local context containing the objects used at that node. The context is shared by all activities of the job executing at that node.

Transparent Invocation of Objects

A major objective of the project was to make object invocation uniform and transparent, providing: location independence, distributed access transparency, uniform access to persistent and volatile objects, concurrence transparency and fault transparency.

Location independence frees the programmer from having to establish explicitly the location of its target object and is usually considered in all object-oriented distributed systems [6,26,10]. In Comandos location transparency is directly supported by a low-level uniform reference mechanism, which attributes unique identifiers to all objects known by the system. A naming service allows to register and translate human readable names to object identifiers

Access transparency can be considered in two perspectives. The first one is to have a homogeneous invocation mechanism which handles distribution, heterogeneity in data representation and communication protocols. The second view is more general and considers the integration between short term and long term objects. To provide access to objects independently of their location, in mass storage or main memory, implies a run time environment able to detect object faults and support dynamic linking.

In Comandos all objects are potentially *persistent*. Persistency is not a static attribute but instead an object becomes persistent when a persistent object has a reference to it. All persistent objects are considered to be reachable from an eternal persistent root.

Objects may be shared by jobs and activities. *Concurrence transparency* means that objects keep internally their own synchronization rules, which are enforced during execution without previous knowledge by the caller. Synchronized objects in Comandos have internal mechanisms for synchronizing accesses.

The system provides *fault transparency* if it is able to continue working in the presence of faults in some of its functional units. A simple way for providing fault tolerance is to confine faults to domains where they can be concealed from the rest of the system. Atomic transactions are a well known technique for fault isolation and were included in the computational model of Comandos. Objects for which it is important to define an error containment domain are designated *atomic*. When used within an atomic transaction, the classical properties (failure atomicity, isolation, permanence and serializability) are enforced on operations executed on these objects. The model is intended to support both short and long duration transactions. To improve efficiency sub-transactions and intermediate checkpoints were incorporated in the model (further details in [7]).

2.3 Virtual Architecture

The system is considered to be composed by several machines designated nodes, executing in a distributed environment supported by local or wide area networks. The architecture was defined as an abstract implementation structure, consisting of a set of distributed functional entities which provide support for the Comandos virtual machine interface.

Two main decisions were taken in the definition of the conceptual architecture. The first one is related with the management of the system. We have considered the notion of a ***domain*** as a set of machines under a closely coupled management. Domains may be fairly large like a campus or a corporation network. Within a domain objects known by the system have a unique low-level reference designated *Low-Level Identifier* (LLI). LLIs form an object address space where object invocations are executed transparently. LLIs also provide a location independent and data independent reference to objects, an important requirement in data modeling applications.

The second important issue is the structure of the storage system. Although the user views the system as a simple address space, the architecture considers two storage levels. Objects are mapped in the virtual address space of a job on demand, when an operation is invoked on them. Objects are mapped out of virtual memory by the system, when a job terminates or when system resources are needed for new or incoming objects. This separation allows to have more efficient invocation mechanisms for already mapped objects and isolates contexts for costly operations such as garbage collection.

To provide a common ground for multiple implementations we defined a set of functional units which should be present in a full Comandos implementation.

Virtual Object Memory (VOM) - The VOM in conjunction with the storage system implements the virtual object memory. The VOM is primarily responsible for implementing object invocation and managing object creation. To support object invocation it keeps track of all currently mapped objects and knows how to fetch persistent objects from the storage system.

The VOM is also responsible for managing the virtual address space of jobs and conducting garbage collection of volatile objects in virtual memory.

Storage System (SS) - Secondary storage is managed by the SS and may be seen as a set of containers. A container is a logical entity with a unique identifier, which may be implemented by several physical containers. Each container is organized as a set of segments. A segment is a contiguous storage unit. Provision has been made for optimizing the access to related objects by grouping them in clusters. Objects and clusters are normally entirely stored within a segment but exceptionally large objects may be partitioned over several segments.

Garbage collection in distributed object oriented systems is a particularly difficult problem, it becomes even more complex in the storage system due to the expensive I/O needed to follow the object references.

Comandos tries to provide a good compromise between efficient resource management and algorithm complexity using aging in the storage system. The idea of segregating objects into generations proposed by Lieberman [20] is used to subdivide storage into several logical areas. When not used, objects age by moving to older generations, until they are eventually deleted or transferred to some off-line backup unit. If an object is invoked and therefore is brought to main memory, it will automatically be upgraded to the youngest generation. Some mechanisms were added to garbage collect temporary objects [7].

Activity Manager (AM) - The Activity Manager handles all functions related with process management. It controls the low-level kernel providing support for jobs, activities, semaphores, timers and triggers. The complexity of the AM may vary considerably depending on the functionality of the underlying kernel. In some cases the AM is just an interface layer with some internal management functions, while in others it may be responsible for implementing the distributed management of jobs and activities.

Communication Subsystem (CS) - All the architecture components are distributed and use the Communication System to communicate with their remote peers. The CS should offer two transport services, one dedicated to efficient remote invocations and another optimized for the transfer of large amounts of data.

Type Manager (TpM) - The TpM is responsible for maintaining information about types and the relationships between them. The TpM will be a fundamental component in an advanced programming environment. It may also be used for dynamic type checking during development of an application.

Transaction Manager (TM) - The TM is responsible for implementing the support for atomic objects, in particular concurrency control and recovery procedures for aborts and node failures.

Object Data Management System (ODMS) - The ODMS provides the functions related to management of classes and queries on objects in the object base. The ODMS is responsible for three main functions: support of the query language; management of classes; implementation of distributed location schemas for class members.

The first four units are the basis of the Comandos kernel, as they provide the underlying support for using objects.

This paper is focused on the implementation virtual object memory, which is the critical component of the system. Little is said about the other subsystems, which are being implemented by other partners in the project [3].

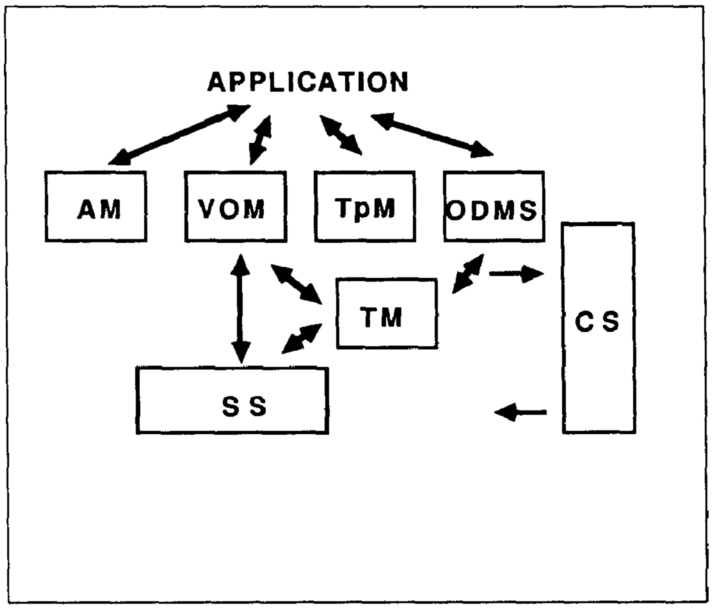

Figure 1: Comandos Architecture

3 Implementation Issues

The basic objective of the implementation prototype was to test the main ideas of Comandos and evaluate some performance issues, trying to identify the possible bottlenecks of the architecture.

3.1 Interface with the Programming Environment

Comandos extends the level of abstraction of the entities handled by the system into a domain normally of the single responsibility of the programming languages. The design issues associated with the implementation of the type model are particularly important, as they condition what type of languages may be used.

A subtask of the project is developing a complete programming environment called Oscar [8], which fully exploits all the capabilities of the type and computational models. However, other programming environments are needed, not only for testing and evaluating the system, but also because we consider as a prime objective to support traditional programming languages.

The environment used in the work described in this paper is C, enhanced with a set of macros to define the implementation types. The macros are responsible for specifying the operations and their parameters and the instance data associated with the object. The C preprocessor generates the operation table, the operation identifiers and creates the templates used to transfer data. Object invocation and control of the active entities of the computational model is provided by a system library. While we are concious that this environment is not adequate for application programming, it proved to be very valuable to define the system interface independently of any programming language. Support for a subset of C++ [27] is currently being implemented.

Operations are identified by a 32 bit value called *method identifier.* At each context, method identifiers have a one-to-one correspondence to method names. We achieve this with a variation of the scheme used by Objective-C [9], adapted to dynamic linking. The library routine *invoke* receives as a parameter a variable containing the method identifier. This variable is left unresolved at compile time.

```
extern char* aMethod;
invoke (aRef, aMethod, param1, param2, ...)
```

When the object is loaded, in consequence of an object fault, unresolved symbols like **aMethod** are inserted in a context-wide dictionary. Each entry in the dictionary contains a pointer to a string corresponding to the variable's name. In the example above, the entry now contains a pointer to the string **"aMethod"**. The dynamic linker resolves any reference to variable **aMethod** to the address of the corresponding entry in the dictionary. The method table, present in the implementation object, associates the method identifier to the virtual address of the corresponding method. The method identifiers of the method table are initialized when the implementation object is loaded in virtual memory, using the same algorithm described above. This assures that two methods with the same name will have the same method identifier, within a context. The method identifier is used when the target object is local. If a remote invocation has to be performed, the operation signature is sent across the network and converted to the corresponding method identifier at the remote node. The overhead of sending the operation signature is negligible, when compared to the costs of communication.

Inheritance is supported in the same way of Smalltalk-80 [13]. An implementation contains only the subset of operations defined by the type, the supertype operations are kept in different implementation objects, which are sequentially searched if the operation is not found.

3.2 Object Structure

Format of Objects

Conceptually an object is composed by its operations and instance data. However, at the implementation level we have two distinct entities: the *data object* containing the instance data of the object and the *implementation object* which contains the code shared by all objects of the same implementation type.

An object is represented by a *header* containing all the information needed by the system to map, invoke and enforce protection on the object. Protection is implemented by a simple access control list specifying the read, write, execute privileges of a given user. The header also contains a template with the description of the instance data in terms of the system basic data types. The template is used to linearize the object when it must be transferred between heterogeneous machines.

Data Objects

Data objects contain the instance data and a *reference list* with references to other objects. The system needs to distinguish references and local data because references have to be traversed in two situations: to conduct garbage collection and when a persistent object is transferred to secondary storage, as all the objects it references are made persistent. Data objects when created are volatile. They become persistent when a persistent object holds a reference to them.

Implementation Objects

Implementation objects contain the executable code associated with the implementation types. Implementation objects have the same conceptual structure of any other object, with an extra memory segment for the code itself. The data section has the operation table and references to other implementation objects.

Object References

Object references are 64 bit values that uniquely identify an object in the system. They are composed of a timestamp and the identifier of the logical container where the object is stored on disk. This information allows to locate the object at any time, as will be described later. Inside a context, the system hashes the LLI to obtain the virtual address of the object's header. This value may be cached for optimization. Only persistent objects have an LLI. As volatile objects cannot be referenced outside their enclosing context, the virtual memory address of their header is always valid, which makes the optimization important for the most frequently used objects.

3.3 Object Memory Management

An object is transparently mapped in a context when one of its operations is invoked. This applies both to data and code: if the data object on which the operation

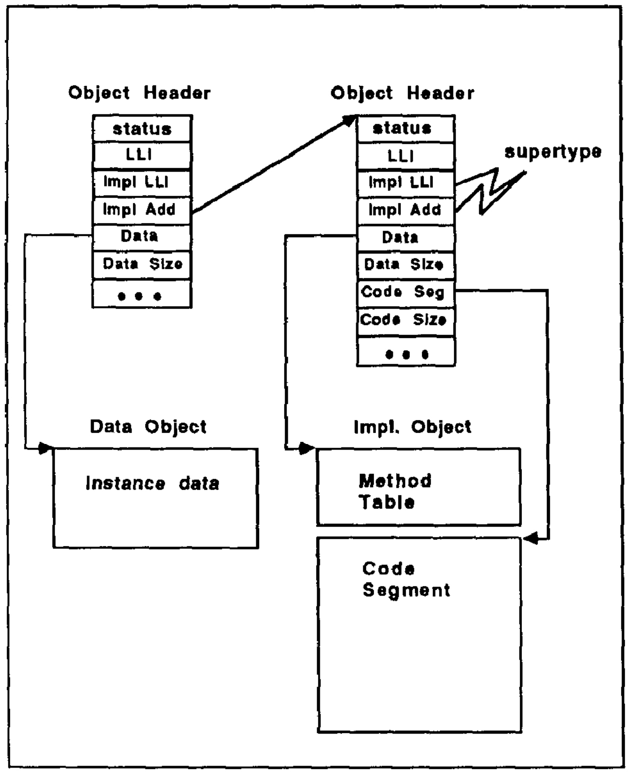

Figure 2: Structure of Objects

was invoked is not present in virtual memory, it will be mapped from secondary storage or from another node; if the implementation object is not present, it will be mapped and linked dynamically.

The management of the job's context is based on two tables, a *Context Object Table (COT)*, containing the references to the headers of all mapped objects, and a *Header Table* storing the object headers.

Exhaustion of the virtual address space of a job, and filling up of systems tables like the COT and Header Table, force to liberate resources for newly created or invoked objects. The VOM uses two techniques: garbage collection of volatiles, swapping of persistent objects.

New objects are created in a dedicated zone in the virtual address space. An object is considered to be volatile if there is no reference to it from a persistent object. Volatile objects have no LLI associated to them. Only these objects are susceptible to be garbage collected in the VOM. The current algorithm is reference counting, a simplified version of generation scavenging [26] is under study. If enough space cannot be found some persistent objects are swapped out to disk. A background task does the object swapping based on the information of memory usage.

All objects which may be referenced outside the context are made persistent and an LLI is attributed to them. This may cause potential garbage to end up in the storage system. The aging mechanism used in the storage system will eventually delete them. More experience is still needed to see the implications on the overall storage management.

3.4 Object Invocation

An important goal of the project was to provide transparent invocation of objects. Invocation is a system call in which the invoking activity presents the system with an object reference, the operation identifier and a list of parameters.

If the object is not mapped the system will fetch it from the distributed storage and map it in the address space of the activity, according to its access control rights.

Objects are transparently shared. In a single node sharing is implemented by mapping the object on the address space of the invoking activities. If the object is mapped remotely we decided to maintain the same model, by diffusing the activity to the node where the object is located. Therefore, invocation can always be considered local and performed within the address space of the invoking process. The protection rules are enforced naturally by the underlying virtual memory management system.

As object invocation is the most sensitive operation in the system, we tried to execute it in user mode, whenever possible. The COT and the Header Table are mapped in the VOM in the job's address space. The run-time support for object invocation is able to read these tables in user mode.

Operation Invocation in Virtual Memory

We can start by considering the simple case when both the target data and implementation objects are mapped in virtual memory.

First the VOM hint is validated. If the hint is valid, the next step is to locate the address of the operation to be invoked. A cache of the most recent invoked operations is maintained in the VOM. The cache is searched hashing the LLI of the implementation object and the operation identifier. If the operation is not in the cache, the implementation object is dereferenced and the operation table is searched.

Searching for Objects

If the hint is not valid, or is found to be outdated, the VOM uses the LLI to find the object. We start by searching the COT with a hashing function. If the object is already mapped in the job's context, the hint in the reference is updated and the call proceeds. If the object cannot be found in the current context, an object fault triggers the execution of the VOM location service, which may run in kernel mode.

The policy for searching an object considers the different categories of objects in the system. A basic distinction is made between data objects and implementation objects. For the latter we first try to locate the object in any other context of the current node, using the *Node Object Table (NOT)*, which keeps record of all objects currently mapped in the node. For optimizing the search, implementation objects are maintained in a separate section of the table, similar to a Unix text table [17]. If the object is not found the LLI is sent to the Storage System, which uses its internal storage location algorithms to search for the object.

Data objects belong to one of the following categories: unsynchronized, synchronized or atomic. For the last two categories only one copy of the object may be mapped and therefore the search, if successful, must always end up by locating the node where the object is active or stored. For unsynchronized objects several copies may be simultaneously active, but if one is already mapped in some context of the job, it must be used to maintain coherence, as the previous call may have modified the state of the object.

To search for active copies of an object the system maintains, in each node having storage devices, a *Mapped Object Table (MOT)*, which keeps record of the current location of mapped objects from that storage subsystem.

Diffusion

If the object is found on a remote node, the VOM must decide whether to diffuse the activity to the remote node, or if possible, to migrate the object to the current node.

Diffusion implies to freeze the local execution of the activity, to transfer the invocation context and to resume the activity on the remote site. An Inter-Kernel Message (IKM) protocol allows to transfer the invoking message and receive the reply, implementing the mechanisms of classical RPC systems [4].

The kernel on the remote node receives the invocation request and checks if this job had previously visited the node. If this is its first visit a local context is created, otherwise the existing one is used. A supporting task is created, or allocated from a pool, to support the invocation.

The parameters of the invocation message are either basic data types or references to objects. Heterogeneity must be taken into consideration when transferring data structures from one machine to another. The RPC protocol always sends the identifier of the initiating machine. The implementation type contains a template of the parameters in terms of the basic data structures. When needed a table driven conversion is performed in the receiving machine.

4 Basic Kernel

IK is a minimal kernel developed at INESC, which provides the basic mechanisms for the implementation of the Comandos platform. The choice of using a small kernel, resulting from a previous work [20], was to avoid dealing with restrictions imposed by the underlying kernel, particularly in what concerns memory management.

The basic kernel has an optimized task implementation, making context switching and interprocess communication reasonably fast operations. These facilities justified the decision of making each functional unit of the Comandos architecture an independent task.

Very simple machines based on INTEL 80286 and 80386 processors were used. Each object is mapped on a segment. This choice imposes some restrictions on the size of objects (in the 80286 machines) but simplifies considerably the management of the virtual address space. Protection is associated with each segment. As jobs share an object through different segment descriptors, jobs may have different access rights to the same object.

In the last months we have ported most of the run-time support for object invocation to SUNs 3 and 4 running Unix.

The prototype supports object invocation in virtual memory. Persistent objects are transparently mapped when an object fault occurs, code objects are linked dynamically. A simplified version of the remote invocation mechanism (without support for heterogeneity) provides transparent access to remote objects.

5 Performance

The following are some preliminary performance results. These figures were obtained on a Sperry IT with a 80286 at 8MHz, on a Sperry PW^2 with a 80386 at 16MHz, with 1 wait state and running in 80286 mode, on SUNs 3/50 and 3/60 running SunOS 3.5 and on a SUN 4/110 running SunOS 3.2, connected by a 10 Mbit/s Ethernet.

Table 1 presents the measured timing values for object invocation. The first value shows the best case, when both the object reference has a valid hint and the operation is in the cache. When the hint is not valid the LLI of the object is hashed using an open hashing table. The values presented in Tables 1 and 2 refer to the case were no collisions occur. The remote invocation measures the call to an object that is already mapped on a remote node.

Table 2 presents the various components of the object invocation time when the hint is valid and a cache hit occurs. The first factor accounts for saving the reference to the calling object and the call to the *invoke* library routine; then the hint is validated by comparing the LLI stored in the header with the one present in the object reference; next the address of the operation is obtained from the cache and a jump is made to that address.

In the best case of an invocation with a valid hint, object invocation is two or three times faster then similar operations performed by Smalltalk executing on equivalent machines [29]. On the other hand, object invocation is between five to ten times slower then a C call. Languages without persistence and distribution achieve ratios between 1.25 and 2.5 [27,23,9]. However, these ratios constitute a lower bound of what we may ever achieve, because objects in these languages are static and reside always in virtual memory. In our case, the effect of adding dynamic mapping of data objects, and dynamic linking of implementation objects, inevitably introduces some overheads in the invocation mechanism. In fact, the values obtained compare favorably with those achieved by other distributed systems, like Emerald [6] and Clouds [10].

We hope that further analysis, and statistical data resulting from some applications developed for the system, will provide a better insight of how to improve the basic system mechanisms.

6 Related Work

There are several research projects with similar objectives which influenced the definition of Comandos.

Eden [5] was one of the earliest projects which tried to integrate an object oriented environment with the basic mechanism of a distributed operating system. Objects have a type which defines the interface they present to the rest of the system. However, the programming model for constructing the objects is different from the one used to specify their interfaces. Emerald [6] took a language oriented perspective and produced an integrated programming environment. The computational model of Eden is based on the interaction between active entities using a distributed RPC. Object are expected to be large and are encapsulated in a process which handles all the interaction.

A project which influenced the evolution of our computational model is SOS [26]. The type model of SOS is totally based in C++ with some extensions for dynamic linking and for controlling inheritance mechanisms. The computational model is based on the server/proxy model. The remote invocation mechanism is not transparent. The programmer must explicitly request a proxy before interacting with the server. The first invocation maps an object which becomes the principal and will be able to reply to *give_me_proxy* messages. The interesting idea is that different proxies can be tailored to the characteristics of the invoker, allowing to optimize invocation or use different protocols. The granularity of objects handled by the system is assumed to be large. Objects are stored persistently and a name manager translate names into unique OIDs, which can be used to invoke the object.

Clouds [10] has very similar goals to Comandos. The computational model also makes a distinction between active entities (threads) and the objects themselves, which are passive and are brought in the context of the thread when invoked. Persistence is handled differently from Comandos as all objects are persistent and exist forever unless explicitly deleted. The design solutions are also considerably different from the ones used in Comandos.

Some generic principles are common to all projects: system wide identifiers, clear definition of interfaces, transparent handling of distribution. Majors differences have to do with the relation between the system and the programming environment, the granularity of objects, the way persistence is handled, the implementation of invocation and protection mechanisms. Finally all have different type of supporting kernels, which account for some of the design decisions.

7 Evolution

The work is continuing in two directions. The first one is an evaluation of the design solutions, in the light of the experience with the first implementation. The second general line of work is focused on providing implementations of the system in kernels which can simplify the interaction with the Unix environment.

Intra and Inter Context Invocations

To consider that invocation is always executed in the context of the invoking job may be a too restrictive approach. An alternative is to consider inter-context invocation, or cross-invocation, which allows to keep both contexts separate, and in some cases can be more efficient, as no remote context or activity have to be created. Cross-invocation simplifies incremental development, by maintaining separate protection domains between entities used within the same program. However, most of the times a persistent object is used by a single activity. Systems exclusively based on a client/server model assume a high granularity of objects, which is not the case in Comandos. Therefore we will maintain our basic invocation model while adding a new category of objects which will be handled as servers.

Clusters

To manage the possible large number of small objects we have considered the possibility of clustering them together. Clusters help to adjust the variable granularity of objects to the size of virtual memory pages, avoiding

		80286 8MHz time (μs)	80386 16MHz time (μs)	SUN 3/50 time (μs)	SUN 3/60 time (μs)	SUN 4/110 time (μs)
C call		7.6	3.9	2.2	1.7	0.8
Object invocation						
Valid hint	Cache hit	54.0	28.0	20.3	13.2	5.1
	Cache fail	98.0	58.0	45.2	35.1	10.7
Invalid hint	Cache hit	84.0	40.0	34.0	29.4	8.7
	Cache fail	128.0	64.0	64.6	45.7	13.0
Remote invocation		15 ms	10 ms	6 ms	6 ms	4 ms

Table 1: Object Invocation Times

	80286 8MHz time (μs)	80386 16MHz time (μs)	SUN 3/50 time (μs)	SUN 3/60 time (μs)	SUN 4/110 time (μs)
Call RTS	11.0	5.9	4.9	4.2	1.3
Test hint	9.5	4.6	1.8	1.0	1.1
Access cache	5.8	3.4	6.0	6.3	1.0
Jump to object	27.0	14.1	7.6	1.7	1.7
Total	54.0	28.0	20.3	13.2	5.1

Table 2: Partial Times of Object Invocation with Valid Hint

excessive internal fragmentation and define the unit for protection.

A cluster is a group of objects with a global identifier. When one object from a cluster is referenced the whole cluster is mapped in a contiguous virtual memory region. The invocation mechanisms for objects remain identical to the ones described previously. Some special management mechanisms must be implemented to handle the extraction of an object from a cluster and to garbage collect the space left free.

Implementation on Other Kernels

The two areas of evolution described above are currently being implemented on the IK prototypes in order to evaluate their impact on the global system performance.

The bare machine prototype allowed us to make experiments without being to much worried about restrictions imposed by existing kernels, in particular in what concerns memory management. However, as mentioned earlier, compatibility with Unix is one of our main objectives. Therefore, we are currently concentrating in the implementation on top of kernels which could enhance Unix functionality, while providing Unix compatibility. We are currently evaluating Mach [1,24] and Chorus [15]. Both provide efficient communication mechanisms and Unix compatibility. They offer the possibility to manage virtual memory by processes outside the kernel, opening some very interesting opportunities for experimenting different invocation strategies and object sharing between different machines.

8 Acknowledgements

The design of the Comandos virtual machine was performed jointly with all the other partners in the project: Trinity College (Dublin, Ireland), BULL (Grenoble, France), LGI (Grenoble, France), IEI (Pisa, Italy), ARG (Milan, Italy) and Olivetti (Pisa, Italy).

In particular the authors would like to express their recognition for the contributions made by Chris Horn, Roland Balter, Elisa Bertino, Vinny Cahill, Alexis Donnelly, Andre Kramer, Sasha Krakowiak, Marie Meysembourg, Xavier R. de Pina, Michel Riveill, Cecile Roisin and Gerard Vandôme.

The authors also wish to thank all the implementors of Comandos at Inesc: Pedro Sousa, José Monge, Pedro Pereira, Manuel Sequeira, André Zúquete, Paulo Ferreira and Paulo Pinto.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of Summer Usenix*, July 1986.

[2] P. America. POOL-T: A Parallel Object-Oriented Language. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220, The MIT Press, 1987.

[3] E. Bertino, R. Gagliardi, and G. Mainetto. An object data management system for supporting complex, advanced applications. In *Proceedings of Es-*

prit Technical Week, North-Holland, Brussels (Belgium), November 1988.

[4] A. Birrel and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1), February 1984.

[5] A. Black. Supporting Distributed Applications: Experience with Eden. In *10th. ACM SIGOPS*, December 1985.

[6] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object Structure in the Emerald System. In *Proceedings of OOPSLA*, Portland, Oregon, September 1986.

[7] Comandos. *Object Oriented Architecture*. Technical Report D2-T2.1-870904, COMANDOS Project, September 1987.

[8] Comandos. *OSCAR: Programming Language Manual*. Technical Report, COMANDOS Project, November 1988.

[9] B. Cox. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1986.

[10] P. Dasgupta, R. LeBlanc, and W. Appelbe. The Clouds Distributed Operating System. In *Proceedings of Distributed Computing Systems*, 1988.

[11] D. Decouchant, A. Duda, A. Freyssinet, M. Riveill, X. R. de Pina, R. Scioville, and G. Vandôme. GUIDE: an implementation of the COMANDOS object-oriented distributed architecture on UNIX. In *Proceedings of EUUG Conference*, Lisbon (Portugal), October 1988.

[12] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[13] D. Halbert and P. O'Brien. Using types and inheritance in object-oriented languages. In *Lecture Notes in Computer Science*, pages 20–31, Springer-Verlag, 1987.

[14] F. Herrmann, F. Armand, M. Rozier, M. Gien, V. Abrossimov, I. Boule, M. Guillemont, P. Leonard, S. Langlois, and W. Neuhauser. Chorus, a new technology for building UNIX systems. In *Proceedings of EUUG Autumm*, Cascais (Portugal), October 1988.

[15] C. Horn and A. Donnelly. Architectural Aspects os the Comandos Platform. In *Proceedings of the Second Workshop on Distribution and Objects*, Decus, Karlsruhe, April 1989.

[16] D. Ingalls. Design principles behind Smalltalk. *BYTE*, 286–298, August 1981.

[17] S. Leffler, M. McKusick, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[18] H. Lieberman and C. Hewitt. A real-time garbage collection based on the lifetimes of objects. *Communications of the ACM*, 22(6):419–429, June 1983.

[19] J. A. Marques, R. Balter, V. Cahill, P. Guedes, N. Harris, C. Horn, S. Krakoviak, A. Kramer, J. Slattery, and G. Vandome. Implementing the COMANDOS Architecture. In *Proceedings of Esprit Technical Week*, North-Holland, Brussels (Belgium), November 1988.

[20] J. A. Marques, P. Guedes, J. P. Cunha, N. Guimarães, and A. Cunha. The Distributed Operating System of the SMD Project. *SOFTWARE-Practice and Experience*, 18(9):859–887, September 1988.

[21] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[22] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California, October 1987.

[23] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. An introduction to Trellis/Owl. In *Proceedings of OOPSLA*, Portland, Oregon, November 1986.

[24] M. Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *Proceedings of the 6th. International Conference on Distributed Computer Systems*, May 1986.

[25] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[26] D. Ungar. Generation Scavenging: a non-disruptive high performance storage reclamation mechanism. *ACM Sigplan Notices*, 19(5), May 1984.

[27] D. Ungar and D. Patterson. Berkeley Smalltalk: who knows where the time goes? In G. Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, pages 189–206, Addison-Wesley, 1983.