

Bull/Inesc development of a Programming Platform for DCE

J. Alves Marques, P. Sousa, M. Sequeira, P. Ferreira, A. Zúquete

INESC - R. Alves Redol n°9 1000 Lisboa PORTUGAL

marques@inesc.inesc.pt

November 1990

1 Objectives

Distributed applications are still a small market in spite of all the promises of technology. The reasons for this apparent delay are not related with unavailability or poor performance of the technology. The main limitation stems from the difficulty of programming and managing distributed applications.

DCE will undoubtedly present an opportunity to produce large distributed applications by standardizing a set of services and interfaces. However, DCE does not address the issue of the programming environment to support the development of applications.

The main objectives of Comandosi [2, 1] was the development of such a platform providing a simple coherent environment where all details concerning distribution, persistence and sharing are hide behind an easy to learn programming methodology.

One of the Comandos prototypes developed in Inesc [3] was implemented on top of Unix and used as programming language C++, a well established object oriented programming language.

A join effort by Bull and Inesc could profit from the Comandos work and the opportunity created by DCE by providing within a short time frame a Comandos environment upon which all future developments can be include.

This paper describes the main technical issues in the project. We start by an overview of the programming model followed by the actual implementation of our system called IK (Inesc Kernel for short). We summarize the main evolution points towards DCE and conclude by some remarks concerning future developments.

2 Programming Model

In spite of the growing interest in distributed systems and the widespread use of computer networks, the development of distributed applications is still a difficult task due to the lack of adequate support. Management is also becoming increasingly more complex reducing the real utilization of available resources. The development of platforms able of handling transparently distribution, heterogeneity and multi-language software in the framework of an object model has been perceived as a promising direction in several projects.

2.1 Object Model

The object model provides an unified way of designating the entities manipulated by the system and establishes a common platform to different languages and programming environments.

In our model an object has an interface exclusively composed by operations. Currently, we do not allow properties to be defined in the interface although higher level environments, such as the ones for data modeling, may provide it by transparently invoking operations that read and store values in the object's instance data.

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

Objects are manipulated through untyped references which identify them uniquely. References are long lived and independent from any context information, they can be stored persistently or used across machine boundaries. Classes are also objects and can have their own instance data.

Object Invocation

Object invocation is the basic primitive of the system reflecting all the features related with transparent handling of persistence, distribution and sharing. We tried to make this primitive as general as possible providing:

- **Location independence.** The programmer does not have to establish explicitly the location of the target object. Distributed access transparency implies an homogeneous invocation mechanism hiding distribution, heterogeneity in data representation and communication protocols.
- **Uniform access to persistent and volatile objects.** The system is capable of trapping invocations on objects stored on secondary storage, reading them to memory and supporting dynamic linking.
- **Transparent sharing of objects.** Objects may be shared transparently when mapped in different jobs. The system provides the basic mutual exclusion on the execution of an operation on an object. The computational model also provides mutexes and condition variables upon which more sophisticated synchronization mechanisms can be explicitly programmed.

Persistence

All objects are potentially persistent and uniformly addressed. Persistence is not a static attribute but instead an object becomes persistent when a persistent object has a reference to it.

When an object is stored in the storage system, all referenced objects are also stored. However, when an object is retrieved from storage system, referenced objects are retrieved only when required.

Object sharing

As persistence, sharing is also a potential feature of every objects. Whenever an object is invoked simultaneously from several users, it is transparently shared among the invoker by the system.

Object sharing can be considered according to two general models, to map the object on shared memory or to perform a cross-context invocation to the context where the object is already mapped. Obviously, these solutions imply different semantics, in the former the object is mapped on both address spaces while in the second a cross-invocation is performed between the two contexts. Differences concerning protection, the fault model and efficiency have been extensively discussed in the usage of these models in concurrent programming languages.

Memory management

Object creation requires management of application memory space. The system frees the programmer from such management by reclaiming memory from objects no longer needed to the application. This task does not introduce obstructive pauses during the interactive work and consequently is completely transparent to the programmer.

Incremental Loading of Classes

The conventional production cycle for a static programming language is a sequence of edit-compile-link-load-run. Incremental loading eliminates the link phase and reduces the loading time of the production cycle, because no executable are created, rather compiled classes (`.o` files) are loaded when the application requires.

Shortening the production cycle is an important issue, as static linking with a large library can take minutes, and the cycle is performed several times.

Concurrency

We chose to separate explicitly active and passive objects. The difference 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 just independent threads of control.

A job generally represents an application and is composed by several objects and one or more activities, running in parallel. The job is a distributed entity with a number of contexts, one or more per node visited. A context is a dynamically varying collection of objects located together at the same node. An activity is the fundamental unit of processing of objects which may be spread across several nodes of a distributed system.

These concepts are supported by two system classes denominated *Job* and *Activity*. When a job or an activity is created, an instance of the corresponding class is also created, providing a coherent interface to the programmer.

2.2 Programming Language - Extended C++

The interface to the virtual machine subsumes a set of primitives for handling classes and objects, and also for requesting services to the various components of the model.

Class and object handling comprise:

- class declarations,
- object instantiation,
- object invocation,
- accessing object data.

As preferred programming language we chose C++ because of its wide availability and growing user community. C++ is extended so that it supports the IK object model (described in the previous section) as well as normal C++ language objects.

IK objects are derived from a *IKObject* class, providing them with the minimal behavior common to all IK objects.

The system services are used through library objects offering an object oriented interface and hiding their implementation peculiarities.

The extensions maintain whenever possible the syntax and semantics of original C++. That is, a programmer using the extended capabilities compiles with a standard C++ compiler.

While we wanted to provide maximum compatibility with standard C++, we realized that complete support of its semantics would be very hard to implement and in a way incompatible with the underlying object model.

These restrictions imply that programs written in standard C++ must be to changed if they use the prohibited constructs (only applies if the objects are to become IK objects).

The construct restrictions imposed in C++, do not affect the process of class (programmer) design, and are a form of only allowing strict object-oriented concepts in the C++ language.

The restrictions are:

- instance data may only be accessed by member functions.
- IK objects are only accessed through references. That is an expression can never be typed after an IK class.
- IK pointer manipulations are prohibited.

Extensions to support persistence

Although persistence provides a single programming paradigm to handle both persistent and volatile data, programmers must select the data that should become persistent.

The system offers primitives to name objects, turning them persistent objects, as well as the graph of objects they refer. Later programmers can access to the persistent graph of objects by translating the name into reference. These services are provided by the *NameService* system class, requiring no dedicated lingual support.

Extensions to support concurrency

The paradigm followed to express synchronization is similar to the one preferred by Monitor based languages. A synchronization object is an instance of one of the system classes: *Lock* and *Condition*.

The interface offered for synchronizing threads is based on these classes.

These classes may be used by the programmer to build more sophisticated synchronization mechanisms like monitors or mediators or some other entity best adapted to a specific application.

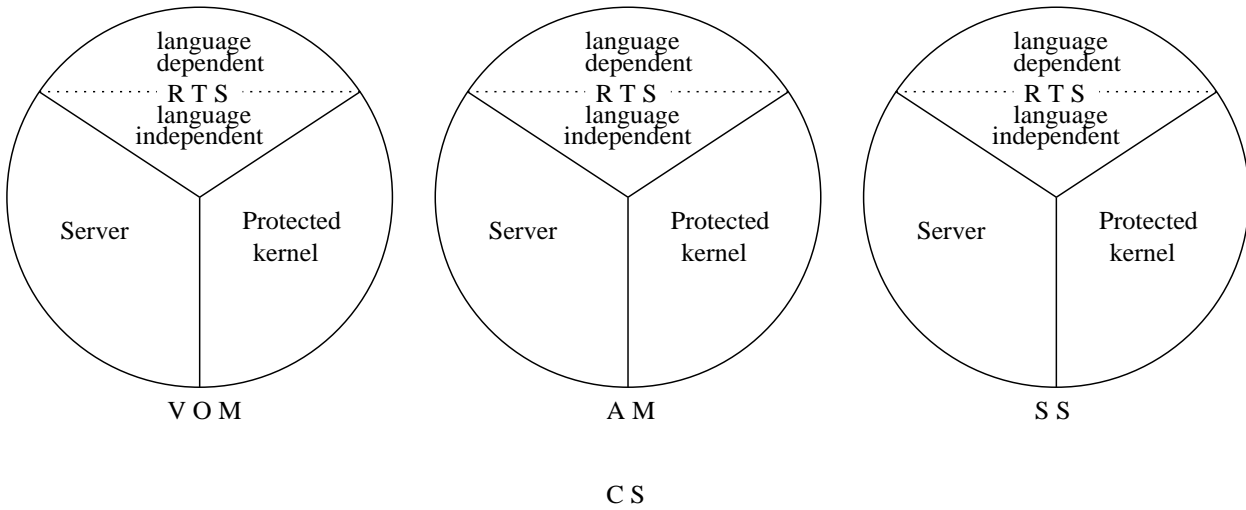


Figure 1: Architecture of the Object Kernel

3 Architecture

The system is considered to be composed by several machines, executing in a distributed environment supported by local area networks. The architecture was defined as an abstract implementation structure, composed by an Object Kernel providing the basic functions to manipulate objects and activities. The Object Kernel is internally composed of five components: the Virtual Object Memory (VOM) handling all operations related with the manipulation of the virtual address space of a context; the Activity Manager (AM) in charge of the active entities; the Storage Subsystem (SS) provides support for persistent objects; the Name Service (NS) implements a single hierarchical name structure spanning over machine boundaries; a Communication Service (CS) responsible for providing a generic RPC interface independent of the underlying stack of protocols.

Object Kernel is implemented as a set of servers running in user mode and a Run-Time Support (RTS) library executing in the address space of the application. For performance optimization all functions that do not compromise protection are executed by this library, which communicates with the servers only when necessary. The RTS library also contains a language-dependent portion providing the language constructs to interface the Object Kernel.

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 restricts the population of objects for costly operations such as garbage collection.

4 Actual Implementation

4.1 Code Production

The actual implementation is a modified ATT C++1.1 translator, in which the code generation phase is altered to make calls to the IK virtual machine interface when necessary. This implementation is to be regarded as a quick and dirty approach to the problem and has already fulfilled its role: providing a prototype to gain experience on programming IK in C++.

The final implementation is already underway. It consists on standard compiler technology (lex,yacc) pre-processor which will compile to standard C++ and to an IK interface also in C++.

4.2 Run-time System

4.3 Object Invocation

Normally the invocation of an object resumes to searching the method on the class of the invoked object. However, if either the object or its class are not mapped, one may step into one off the following situations:

- The object is stored on disk: Object Retrieval mechanism should retrieved the object from Disk.
- The Class of the Object is stored on disk: The Incremental loading mechanism should map the class object.
- The Object is mapped remotely: A Cross-context invocation should be forward to the remote node.

Object Retrieval, Dynamic Linking and Cross-context invocation are addressed in the following sections.

4.4 Dynamic Linking

Delaying the binding between procedures names and their code until execution time, makes the mapping of class objects a complex operation. The class object file (extension .o) is read and mapped. The internal references are relocated and the unresolved symbols are resolved against those in a global Symbol Dictionary, containing the global code, composed of IK and Libraries code. This global constitutes the platform executable. As it is used by all users it is transparently shared, by most Operating Systems, among them.

Dynamic loading of class objects is implemented without any system support. However it assumes a standard format of the object files. In the present implementation we only support BSD `a.out.h` format. Support for COFF format is being developed.

4.5 Cross context invocation

The implementation of cross-context invocations is subdivided in two modules:

Management of the stack frame - As remote invocations are only detected at run-time, no previous preparation is done at compile time. A significant part of the cross-context invocation implementation is dedicated to handle the invocation parameters. Any type of data can be passed as parameters, including pointers.

As parameters are not size limited, large messages are fragmented and reassembled at this layer.

Heterogeneity is solved by converting the format of the parameters when they are transferred to a heterogeneous machine. Again, this is implemented at the RTS level, so it is both system and machine independent.

Underlying RPC mechanism - The message passing is assured by the underlying RPC mechanism, from which we assume no particular semantics. In the current implementation, we use the Sun RPC package.

The major issue is avoiding dead-locks during the execution of the rebounding calls, and handling time-outs due to heavy network traffic.

4.6 Name service and Storage subSystem

The fundamental functionality of the NS and SS is implemented by a module that is linked to any system application that deals with IK persistent objects.

The IK is implemented on top of a system constituted by several node running Unix SunOS BSD 4.2 and with a SUN NFS distributed file system. System-wide configuration data, that is used both by management activities and user jobs, is available in maps of the Yellow Pages distributed service for permanent availability in all system nodes. The NFS file system is fundamental to access remote files but the Yellow Pages service may not exist if a global replication of configuration files is done manually. The IK uses a private YP domain.

IK Name Space

The IK name space is a directory hierarchy build on top of the Unix file system. It hides the installation details of the underlying file system from the applications using IK persistent objects (namely the NS/SS run-time interface). Because the SUN NFS does not have a system-wide super-root for the distributed file system (each node has its own root), the root directory of the IK name space (**/IK**) is replicated in all nodes.

Name Service

The NS maps user-readable names into LLIs. Object names may exist in any directory of the entire distributed file system and are not detectable from persistent objects (like symbolic links from i-nodes of Unix files). Object names are subject to a versioning mechanism, which intends to avoid the involuntary overwriting of object names. The lookup of object names, without version, always fetches the one with higher version number. Object names can be freely manipulated by the users, including their elimination.

Storage Subsystem - SS

A persistent object managed by the SS is not simply a file in a traditional file system. An object is a container for a value of some kind (instance) and is associated to a set of operations, exported by its class, that controls its internal data manipulation. Therefore a fundamental difference is that the SS must keep track of the relations between each persistent data object and the code that implements its class.

The basic run-time functionality of the SS is to store and retrieve an object from and to secondary storage given its LLI. The SS is also responsible for assisting the mapping and unmapping of persistent objects and for retaining and update information about their **Object Managers**.

Persistent data objects are grouped in **clusters** to optimize the storage and retrieval of them. The mapping of a particular object implies the entire mapping of its cluster. Thus, the cluster policy should ensure a high correlation on the usage of all objects in a cluster. Clusters are also the unit of protection, i.e. there is a common protection for all objects in a cluster, both in secondary storage and virtual memory.

We envisage two possible ways to store clusters in a Unix file system: (1) using one file per cluster and grouping clusters in well known directories or (2) using simple data bases that maintain key/content pairs (like the one provided by the **ndbm** package).

The data bases approach seems the most interesting one for two reasons:

- Persistent objects are searched by LLI and not by a string of characters. Thus, the search of a persistent object is more efficient in a data base with a key-based search than in an Unix directory, where files are searched sequentially by name.
- For clusters small then the minimum block size of the file system the data base approach is more economic in the usage of secondary storage.

Clusters are, by default, stored in an original SS container. However, they can migrate to other SS containers for two reasons: improve locality of access or to spread SS containers load by other SS containers.

Clusters are the unit of protection for persistent objects. The protection is implemented by an ACL per cluster. The ACL describes access writes for three kinds of entities: owner, users and groups. The access rights are one of three levels of protection: MAP, INVOKE and NOT. The MAP right allows the mapping of the object, the INVOKE right allows its invocation in other context, which must have the MAP right, and the NOT right denies both its mapping or invocation. The Unix concept of file protection, using independent read, write and execute protections, is not well suited for persistent objects. If a persistent object can be mapped then it can be both read and write. The execute access no longer is important since the executable code is constituted by class objects and they can be executed only if they can be mapped.

4.7 Garbage Collection

The policy for managing the memory occupied by the objects is a fundamental facility in a platform such as the one we are describing. A fundamental decision was to exploit the different characteristics of volatile and persistent objects to isolate garbage collection in main memory from storage reclamation in the storage system.

Volatile objects have a high probability of dying while persistent objects are expected to live much longer. The confinement of volatile objects to a single context makes it easier to implement a garbage collector which deals essentially with well known problems.

On secondary storage we do not attempt, for the moment, to do garbage collection we merely use an aging policy. Objects are segregated into generations subdividing the storage system into several logical areas. Unused 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 into main memory, it will automatically be upgraded to the youngest generation.

Volatile objects are garbage collected using a generation scavenging algorithm which follows the the structure of the published algorithm [4]. As large objects tend to live longer, volatile objects with a size exceeding a threshold value are allocated in a special area where they are not subject to copy. A mark and sweep algorithm, triggered when the garbage collector fails to recover a minimal amount of space, is used to clean such objects.

The algorithm used is Generation Scavenging which was first implemented in the Berkeley Smalltalk. In our platform, this solution has to consider not only the existence of persistent objects but also multithreaded and distributed applications.

Persistent objects are not subject to automatic on-line garbage collection. They are only scanned for references to new objects which for that fact are also alive. When the Generation Scavenging runs, all threads of the context are stopped with exception of the one supporting the garbage collection.

The algorithm is purely local. All objects known outside a context are turned persistent, so they are not garbage collected on-line as already mentioned. A distributed garbage collection would imply noticeable pauses with doubtful improvements.

4.8 Jobs and Activities

The AM provides operations to create, exit, suspend, resume and kill a job. Besides this, it also offers the possibility to create, start, suspend, resume, detach, join, exit, yield, check the state and set or get the name of an activity. It also supports the instances of the systems classes used for synchronization.

Jobs are formed by one or more Unix processes (tasks in Mach, actors in Chorus) and activities are co-routines inside them (threads in Mach and Chorus). The co-routines used were adapted from the C Threads package used in Mach.

A job is formed by several contexts and is represented by a job object (instance of the system class *Job*). Every activity is supported by one or more threads and is represented by an activity object which is an instance of the system class *Activity*. Thus, an activity is the conceptual flow of execution that can be spread by several nodes being physically supported by several threads.

Jobs

A job object is always mapped in the context that invokes it. As a matter of fact, its data is never changed so there is no problem in keeping its coherence in case a context crashes. To achieve this, is always kept in the Storage Subsystem the information that the object is always unmapped.

Thus, when a job is invoked, there is first a local invocation and then the AM primitives are called in the instance methods. These primitives originate a remote procedure call to a specific AM server. The AM server that is called is the one that exists in the node where the context is to be created or already exists.

Activities

An activity is supported by one or more threads, implemented based on a co-routine package.

Most usual blocking calls to the operating system were redefined in order to yield automatically the calling thread allowing others to use the processor. The Unix calls *read*, *recv*, *recvfrom*, *recvmsg*, *accept* were redefined in a way that the activity first calls *select()* on the relevant descriptor.

The scheduling of threads is completely transparent to the programmer and are managed in a round-robin fashion.

4.8.1 Synchronization

The synchronization objects are based on mutexes and condition variables. Thus, each lock object is supported by a mutex and a condition object by a condition variable. Associated with a condition variable is a queue of the threads blocked on it. There are primitives that allow the signaling of only the first blocked thread or all threads.

4.9 Sharing Objects

Our decision was to let the application designer choose the best policy. We also tried to make invocation as independent as possible of the sharing policy. We can summarize our model as follows:

- All objects are potentially sharable.
- The basic sharing mechanism is cross-context invocation .
- Whenever possible memory sharing provides a more efficient implementation which is totally coherent with the semantics of our invocation.

To have this separation we needed another notion in our model corresponding to the place where decisions concerning sharing are made. Although this component had to be integrated in our invocation mechanisms it had to be visible at the user level as any other object.

We considered that all persistent objects are associated with another object, the *Object Manager* which is responsible for defining the policy to follow when there is a request for mapping an object.

By default the Object Manager of an object is the Storage Subsystem where the object was originally created. When the object is mapped the control is passed to a standard Object Manager mapped at each context. Contexts always contain a dedicated thread waiting for requests concerning the objects they currently manage. The programmer may also explicitly assign an Object Manager to a given object (or objects).

5 PORTING TO DCE

The actual implementation is based on Unix BSD (SUN OS) uses the SUN/RPC, NFS and shared memory.

The port to a DCE environment will consist essentially in the substitution of some of SUN underlying mechanisms by the ones proposed in DCE.

- SUN RPC \Rightarrow NCS RPC
- Unix authentication \Rightarrow Kerberos
- Unix Naming \Rightarrow DNS
- NFS \Rightarrow NSF+ AFS

Next sections present the expected implications of these modifications on each platform component.

5.1 Object Invocation

Object Invocation is performed by the RTS accessing its own data structures and thus should be fairly independent of both the Operating System and machine architecture. Although some of these structures are generated at compile time, they require no particular skill of the C compiler. Further, currently we are using the GNU gcc compiler which is also available on DCE.

The porting the RTS to the DCE platform should present no major problem.

5.2 Dynamic Linking

As already mentioned, the major dependency of the Dynamic Linker is the format of the object file. As DCE also uses the standard BSD `a.out.h` format for object files, porting the Dynamic Link mechanism should not present major problems.

5.3 Cross context invocation

As we handle long messages and Heterogeneity ourselves, we accept RPCs package with limited size messages and no heterogeneity support.

The choice of using the Sun RPC was only due to practical reasons. Although not ware of the full specification of the DCE RPC, it is expected that it rises the same amount of problems than the Sun RPC, particularly handling error situations.

5.4 Name service and Storage subSystem

DEC Name Service - X500

The actual generation and lookup of object names should be easy to implement using the DEC Name Service. The model implemented is inspired in the VMS name generation/lookup and on the facilities offered by the Unix shell in the usage of directory search lists.

Kerberos Authentication Service

The actual implementation does not implement any authentication of users. This area needs further study to try to incorporate as much functionality of Kerberos as possible without sacrificing performance or making extensive redesign of the system.

AFS + NFS

The porting of the actual implementation to a system with a distributed file system based on AFS + NFS does not create any problem since the current implementation already uses NFS. An eventual absence of the Yellow Pages service is not a restriction since the actual implementation can also work without it.

Garbage Collection

If the threads used were simply co-routines, than there should not be any problem since the algorithm just uses common Unix primitives. However, if preemptive threads are used, the present garbage collector has to be carefully considered in order to synchronize its operations. Otherwise, the coherence of objects addresses could be destroyed because the garbage collector uses a copying policy.

5.5 Jobs and Activities

The Activity Manager itself should be easily replaced by preemptive threads since the offered functionality is certainly the same. However, its impact on the RTS must be carefully considered since critical operations must be synchronized.

5.6 G++ Development chain

The use of the G++ utilities, specially the grammar description, is subjected to FSF copyrights agreement. The implementation described above is not affected by this copyright problem, since only the pre-processed code needs to be compiled by a C++ compiler being it ATT, GNU, or any other. No source code modification or even IPRs of any C++ compiler is involved.

We should verify the Bull plans for supporting G++ and study the availability of the G++ utilities (debugger, grammar description, etc...)

6 Future Work

These items correspond to the points where we fell a evolution is need and the ones raised in the meeting of July between Bull and Inesc.

6.1 Full C++ Support

The restrictions are unimportant when designing classes from scratch. Nevertheless incorporating already classes already written in standard C++ and applying the restrictions involves some effort. Processes are being devised to alleviate the restrictions though still maintaining C++ syntax and semantics.

6.2 Improving the Kernel

In a distributed environment object invocation may fail when the invoked object is not accessible, either for physical or protection reasons. An exception mechanism should be associated to the object invocation, allowing applications to handle such errors.

Dynamic linkage should be integrated with standard debugger. A way of doing it, is updating the executable file when a new class is dynamically loaded.

Cross Context Invocation should be implemented based on IPC messages, not on top of a standard RPC, mainly due to error handling and dead-lock prevention during rebounding calls.

Concerning Gargage Collection, there should be a management component which periodically reclaims un-referenced objects in the storage subsystem avoiding the existence of large quantities of garbage in secondary storage. Another aspect for future work should be the extension of the actual algorithm in order to reclaim the memory space explicitly allocated by the programmer with the *malloc()* Unix call. This feature should be mainly used in applications written in a language which is not purely object-oriented as C++ or Objective-C.

6.3 New Services

Configuration and Management So far IK does not provide any management tools. It is necessary to provide tools to configure of the system and to archive persistent objects in backup storage.

Interfaces with standard software In spite of our present prototype allowing to interface standard software, the interface itself cannot benefit of the platform features, such as persistence, dynamic linking and so forth. Namely, interfaces with SGDB and window systems such as X and Motif must be better integrated with the platform.

Architecture Extensibility Tools must be provided to allow communication and object mobility between different domains. Access to services external to the platform should also be possible through the transparent usage of proxies.

Programming Environment To fulfill its goals, the platform programming environment must include a browser, a distributed and fully integrated debugger, and a set of systems objects encapsulating services like signals, sockets etc.

References

- [1] P. Guedes and J. A. Marques. Operating System Support for an Object-Oriented Environment. In *Proceedings of the Second IEEE Workshop on Workstation Operating Systems*, Asilomar, CA., 27-29th September 1989.
- [2] J. A. Marques and P. Guedes. Extending the Operating System to Support an Object-Oriented Environment. In *Proceedings of OOPSLA 89*, New Orleans, 2-6th October 1989.
- [3] P. Sousa, P. Ferreira, J. Monge, A. Zúquete, P. Guedes, and J. A. Marques. IK Implementation Report. Technical report, INESC, Setember 1990.
- [4] D. Ungar. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices*, 19(5):157-167, May 1984.

Contents

1 Objectives	1
2 Programming Model	1
2.1 Object Model	1
2.2 Programming Language - Extended C++	3
3 Architecture	4
4 Actual Implementation	5
4.1 Code Production	5
4.2 Run-time System	5
4.3 Object Invocation	5
4.4 Dynamic Linking	5
4.5 Cross context invocation	5
4.6 Name service and Storage subSystem	6
4.7 Garbage Collection	7
4.8 Jobs and Activities	7
4.8.1 Synchronization	8
4.9 Sharing Objects	8
5 PORTING TO DCE	8
5.1 Object Invocation	8
5.2 Dynamic Linking	8
5.3 Cross context invocation	9
5.4 Name service and Storage subSystem	9
5.5 Jobs and Activities	9
5.6 G++ Development chain	9
6 Future Work	9
6.1 Full C++ Support	10
6.2 Improving the Kernel	10
6.3 New Services	10