Project Nr 2071

SUPPORTING OBJECT ORIENTED LANGUAGES ON THE COMANDOS PLATFORM

Vinny Cahill [1], Chris Horn, Gradimir Starovic
Distributed Systems Group, Dept. of Computer Science, Trinity College, Dublin, Ireland.

Rodger Lea
Chorus Systèmes, 6 av. Gustave Eiffel, 78182 St. Quentin-en-Yvelines, France

Pedro Sousa
INESC, Rua Alves Redol, 9, 1000, Lisboa, Portugal

Abstract

The Comandos project is designing and implementing a platform to support distributed persistent applications. In particular the platform supports the object oriented style of programming. An essential requirement of the Comandos platform is that it must support applications written in a variety of existing as well as new (object oriented) programming languages. Moreover, the platform must support interworking between different languages. Each language may naturally have its own object model and execution structures implemented by a *language specific* runtime system. Rather than forcing each language to adopt a common object model and execution structures in order to exploit the distribution and persistence support provided by the Comandos platform, Comandos provides a *generic runtime* system on top of which individual language's specific runtimes may be implemented. In this paper we show how a language specific runtime for an existing language such as C++ can be constructed above the Comandos generic runtime.

The growing use of distributed systems reflects both technological and organisational evolutions. Advances in computing and networking have led to the use of local-area distributed systems composed of heterogeneous workstations and servers. Human organisations are often, by their nature, distributed, but with strong requirements for interworking and overall integration within the enterprise. This evolution leads currently to the concept of "collective computing" [3], in which the overall application is constructed or assembled from many components, each performing its own specialised function.

The development and integration of application software is currently a labour and cost-intensive proposition, particularly for distributed applications which handle large volumes of structured data. Methodologies and tools are needed to master the complexity inherent in heterogeneous distributed environments and application requirements.

Comandos is primarily targeted at the development and support of integrated distributed applications within a *cell,* which constitutes the basic organisational and administrative component within an enterprise. A cell is composed of a set of cooperating workstations, servers and processor pools connected through a high-speed local area network. The goal is to present the distributed system as a coherent entity to its users despite the variety of its components, and to provide the enabling technology for reducing the cost of distributed application development and maintenance.

---

[1] email vjcahill@cs.tcd.ie

The overall objective of the Comandos project is to identify and construct an integrated multi-language application support environment for developing and administrating distributed applications which can manipulate persistent – long-lived – data. It is intended to provide such an environment in the framework of multi-vendor distributed systems. This platform will allow both the development of new applications, and the co-existence with old-style (Unix[2] oriented) applications.

The Comandos platform includes infrastructure for:

- distributed concurrent computations;
- storage and retrieval of persistent data;
- multiple programming languages;
- reuseable and extensible software modules;
- secure and protected data;
- on-line management, monitoring and control;
- access to pre-existing applications and information systems;
- interworking with non-Comandos environments.

The project is innovative in that it is integrating operating systems, programming languages and databases technologies. The unifying view is provided by a model and system architecture based on the object-oriented approach, coupled with persistent distributed storage.

Within the Comandos project, the object paradigm provides the basis for an integrated view of application construction, and system management and control.

The project started by defining a conceptual model for structuring distributed applications. This model defines the Comandos virtual machine, and provides an object-oriented view of the distributed environment.

Based on this model, the project is providing a multi-lingual environment which supports the concepts of the model through various languages (initially C++[5], Eiffel [4] and the Comandos object-oriented language, known as Guide [2]).

In this paper we describe the support provided by the Comandos platform for language implementers and show how an existing language can be adapted to use the facilities of the Comandos platform.

1 The Comandos Virtual Machine

The Comandos Virtual Machine is internally composed of six components:

- the `Execution Subsystem` (ES) provides support for object execution and the Comandos notion of a distributed process;

---

[2] Unix is a trademark of Unix Systems Laboratories, Inc.

- the `Virtual Object Memory` (VOM) hnadles all operations related to the manipulation of virtual address spaces;
- the `Storage Subsystem` (SS) provides long-term storage for persistent objects;
- the `Transaction Subsystem` (TS) supports the execution of atomic transactions;
- the `Communication Subsystem` (CS) is responsible for providing a generic RPC interface independent of the underlying stack of protocols;
- the `Protection Subsystem` (PS) ensures the specified level of protection during application execution.

The implementation of these components is split across three interfaces.

- the *Virtual Machine Interface* — The interface across which a supported language communicates with the Comandos platform;
- the *Kernel Interface* — Dividing those parts of the platform which are accessible directly by applications ("user" mode) and those which are accessible only in a privileged mode (i.e. the Comandos kernel);
- the *Environment Interface* - The interface from a Comandos implementation to its underlying hosting environment: Unix or micro-kernels like CHORUS[3].

The Virtual Machine Interface is the uniform view presented by the Comandos platform to each of the various supported languages. It is provided by the primitives of the *Generic Runtime* (GRT) layer that itself interfaces with the services of the underlying Comandos kernel.

The GRT implements the local object space directly manipulated by application programs i.e. it is mainly concerned with local object management. The GRT is implemented by code running in user mode within each address space. Services such as address space creation and deletion; remote invocation; sharing of objects between address spaces as necessary and object location within the distributed system may be implemented in kernel mode or by specific servers as well as in user mode within an address space, depending on the underlying environment.

As different languages have different calling semantics, a *language specific runtime* must adapt the GRT primitives to the language specific format. Moreover, as most of these primitives are based on manipulation of objects, whose format and model differ is each of the different languages, each specific runtime must also hide these language dependencies from the GRT support.

To provide this flexibility and to make a minimum number of impositions on any language, we propose a general model in which the language makes calls to the GRT but expects the GRT to understand little of the semantics of these calls. To deal with the problem of language specific information, the architecture makes heavy use of *up-calls*, where an up-call is a call from a lower level to a higher one, using an entry point previously supplied by a regular call (*down-call*).

---

[3] CHORUS is trademark of Chorus Systèmes

This two-way interface between a language-specific runtime and the GRT allows objects of heterogeneous languages to be handled commonly by the GRT. This scheme is sufficiently generic so as to allow flexible implementations of both the generic and language specific runtimes.

The next section gives an overview of the basic functionality of the GRT.

2 The Generic Runtime

The GRT provides fine grained passive data objects (which will be refered to, in this and subsequent sections, simply as *objects*) to which a language, through its language specific runtime, may bind class code. Language level object models can thus be built. An object may be uniquely identified by its so-called *global name* which is sufficient to locate the object even when it is mapped on a remote node or stored in the storage subsystem.

The GRT provides generic support for local management of these passive objects, including their creation and possible garbage collection. Objects, which are viewed by the GRT simply as contiguous blocks of memory, are used to contain and enclose language level objects (refered to as *language objects*) containing both direct data and references to other language objects.

Supported languages (i.e. their language specific runtimes) can build language specific forms of object references (refered to as *language names*), such as tagged pointers, which may be stored in language objects, and thus in objects while they are mapped into some address space.

In particular, the GRT supports direct addressing over the objects mapped into an address space. The identifiers for objects communicated between the GRT and a supported language take the form of direct addresses[4].

The protocol between the GRT and the language (specific runtime) - the upcall mechanism - allows the GRT to perform a number of operations on language objects, such as location of language names stored in local objects and conversion of these names to/from global names as objects are mapped and unmapped. This protocol requires that each object must have a minimum set of methods, one for each basic operation which the GRT may request.

Object Creation and Promotion
Objects are brought into existence by the `Create` call. Create allocates space, which is untyped, and will be used by the language specific run time as a repository for language objects.

Any object created in this way is known as a *volatile*. Such objects exist and are known only within the address space in which they were created. Objects may become known outside of their address space of creation (either because a reference to the object has been passed out of the address space, or because the object itself migrates out of the address space). If this happens, the object is said to have been “promoted” to being a globally known object.

Actually achieving this promotion is a two stage process. First, the object is registered with the GRT by the `Register` call which associates an upcall table to the object allowing

---

[4] Note that a language name itself is not necessarily a direct address.

it to be managed by the GRT.

To become globally known, the language specific runtime must call the `AssociateGName` function which returns a global name for the object.

The object cannot now be garbage collected using address space local information alone. The object may be unmapped on address space deletion or when no longer required by the language run-time.

Object Global Naming

Objects are assigned global names (only) when they are promoted. Thus a location independent form of naming is used to transmit object references outside of the address space in which the object was created. Global names may be passed in remote/cross-address-space invocations and stored in the storage subsystem along with the object which contains the name.

The GRT interacts with the storage subsystem in allocating global names since a global name consists of a secondary storage container identifier and a generation number which is unique within that container. The container identifier names the secondary storage container initially specified for the object. If the object migrates between containers, then the initial container must track the object's current storage location. Volatile forwarding information may be used to accelerate a search, but may be out of date or incomplete due to node crashes.

Object Invocation

Object invocation is the basic primitive of the model reflecting all the features related with transparent handling of persistence, distribution and sharing.

Invocations on objects mapped locally are performed at language specific run time level. However, attempts to access objects which are not mapped in the current address space - *object faults* - are trapped by the specific runtime which calls the GRT. It in turn, either arranges to map the object into the current address space, or arranges to carry-out a cross-address-space or remote invocation as required. The GRT may interact with the protection subsystem, storage subsystem and global location service in determining how to handle the object fault. The following paragraphs introduce the basic mechanisms within the GRT to handle object faults.

Object Mapping

When mapping an object, the GRT must first allocate memory for the object. The GRT translates global names contained in the mapped object to language names, by interacting with the mapped object's specific runtime. This translation may be delayed until the object is actually used (e.g. being invoked) if the object was pre-fetched. If the referenced object does not currently reside in the local address space, then the GRT may be used to build a local data object which represents the referenced absent object.

The translation from global name to language name is maintained in the GRT's Context Object Table (COT) within the address space where the object is mapped.

Groups of objects, i.e. clusters, form the actual unit of transfer between the GRT and the storage subsystem. Since objects are retrieved in clusters, the GRT must handle demand loading of one object and pre-fetching of the others. Pre-fetched objects are simply registered in the faulting address space's COT.

Remote and Cross-Address-Space Invocation
The GRT co-operates with the requesting language in performing remote and cross-address-space invocations. The GRT supports the building of parameter frames for transmission between heterogeneous machines by supplying - to language dependent RPC stubs (which may be generated by standard stub compilers) - generic routines to encode parameter data in network format. The language specific runtime can then perform marshalling into buffers provided by the GRT using the appropriate GRT routine to encode the data.

The GRT interacts with the communications subsystem and the protection subsystem to supply the mapping between the target objects global name and the authenticated remote invocation transport path.

Local Garbage Collection
The creation of new objects may lead to exhaustion of memory. To recycle unused memory, the GRT incorporates a local (per address space) garbage collector, taking externally known data objects (which have been assigned a global name) as its root, and which collects unreachable volatile objects.

As the GRT imposes no parameter frame formats, the garbage collector must be conservative when scanning stacks[5]. The local collector must cooperate with on-line global garbage collection, if it is present. Local garbage collection uses the upcall mechanism to locate object references held by typed language level objects. The garbage collection should be compacting only if dynamic object movement is supported by the language and its specific runtime.

3 Supporting an Object Oriented Language above the GRT

In this section we examine what the major technical problems are when mapping a language such as C++ onto the Comandos Virtual Machine.

Our goal is to transparently provide the C++ programmer with a distributed persistent programming environment. However, we are concerned to minimise the side-effects of using our platform for programmers. This generates two, conflicting, constraints: that we maintain, as far as is possible the current model and syntax of the C++ language, yet, we do not penalise C++ programmers who make no use of the enhanced functionality provided.

Both distribution, and persistence impose a common requirement on the C++ programming model for which current compilers do not cater. All references to objects may actually point to objects that are not mapped in the current address space i.e. objects which are either mapped on a machine elsewhere in the system, or are currently inactive and so are on secondary storage.

Since current implementations of C++ use virtual memory pointers to refer to objects we are faced with a dilemma: either we adapt current compilers to generate globally unique, long lived object pointers, or we add support for the GRT to map the compiler generated local object identifiers to global ones i.e. those generated and managed by the Comandos GRT.

---

[5] Stack allocated objects are never visible to the GRT.

Another problem is the need to deal with pointers to context dependent data, for example file descriptors. If we pass a file descriptor out of an address space, without some means of conversion, it may be impossible to use that descriptor in any other address space.

In summary, the language designer has three main problems to solve:

- Dealing with context dependent data.
- Managing pointers to stored objects.
- Managing pointers to remote objects.

3.1 Context Dependent Data

We adopt an approach to context dependent data that requires programmer intervention. Our motivation is that such data is outside the scope of the object oriented world, and, more importantly, the programmer making no use of distribution or persistence should not be concerned with the problem.

The approach is to encapsulate such data within a type that is capable of dealing with distribution or persistence.

Consider for example a file pointer; if we create a function that the system can call whenever a file pointer is passed into or out of an address space, to open or close the correct file, then we can correctly support this pointer.

3.2 Pointers to Stored Objects

Since an object may contain references to objects that are currently in the persistent store, we must ensure that any attempt to access such an object causes an 'object fault'. We are currently experimenting with two mechanisms for this purpose: The first mechanism involves the use of a proxy object to represent the stored object such that when accessed, the proxy is responsible for locating the correct object and overlaying itself. The alternative mechansim is to have references to absent objects point to invalid memory so that attempts to access such objects cause memory faults. Thus we can trap the 'object fault' without using a proxy and can load the required object in the fault handler.

3.3 Pointers to Remote Objects

The solution to the last major problem, that of trapping invocations to remote objects and forwarding the request uses a model similar to that described above. However, in the remote case we are always constrained to use a proxy so that accesses to the data of the object can be caught. Using a proxy object in place of the remote object, we catch invocations, locate the remote object, marshall the arguments and send the remote request using the support provided by the GRT.

4 Using the Comandos GRT

To illustrate how the Comandos GRT supports the C++ language, we detail the creation of a C++ object on the Comandos platform and the subsequent invocation of another C++ object.

4.1 Object Creation

In traditional languages such as C++, language objects can be instantiated in several possible scopes. An object can be declared in a global scope (outside of any function or class), and thus reside in the data space of the application; can be dynamically allocated in the heap of the application, by means of mechanisms similar to `malloc` in Unix; can reside in the stack if declared within the scope of a function definition; and can exist within another object (wherever it may be).

However, this scenario of object creation is independent of the GRT, which does not require language objects to be located in a specific region. Although the GRT provides the `Create` call for allocating new space,[6] we are never constrained to use only this space. The only requirement is that all *promoted* objects reside in memory known to the GRT. Thus not all language objects need be known to the GRT.

Objects to which references are passed outside of the address space in which they were created (as described in section 2) need to become known to the GRT. This promotion is an operation carried out on demand. In fact, whenever references are sent outside of the address space, a test needs to be made to determine whether or not they refer to globally known objects. If they refer to volatile objects, then these objects must be promoted by calling the `Register` and `AssociateGName` primitives as described previously. The test is carried out by language specific code, i.e. as part of the upcall to the appropriate object.

When the `Register` operation is called, the language specific run-time must provide the binding to the correct upcall functions to allow the GRT to manage the newly promoted object with respect to persistence, distribution and context dependant data.

This code, that encapsulates context dependant data and the upcall functions that support persistence,[7] may be generated automaticaly using either pre-processor, or compiler adaptations, or may be hand generated by the programmer.

The reader is directed to appendix A for a full description of the upcall functions that must exist for each GRT managed object.

4.2 Object Invocation

When an invocation is attempted using a virtual memory pointer, a number of events may be triggered depending on the state of the corresponding object:

- The object is known locally and the invocation proceeds without use of the GRT.
- The object is unknown locally, causing an 'object fault', passing into the GRT and asking the GRT to locate the object and map it locally.

---

[6] as described in section 2

[7] as well as the proxy code to be used for remote objects

- As above, however, the object is already mapped elsewhere or must be mapped elsewhere[8], requiring a remote invocation to be made.

Here we are only concerned with the last two cases, as the first will be confined to the language specific runtime and makes no use of the GRT.

4.2.1 Map the Object Locally

When the use of a reference causes an object fault, control is passed to the GRT via the `Invoke` primitive passing as parameters the global name of the object and `ActivationData` - a structure holding the method identifier and parameters for the invocation. It should be noted here that the code handling the object fault must make a mapping from the local C++ virtual memory pointer to a valid Comandos global name.

`Invoke` causes the GRT to locate the object. We will assume here that the object will be mapped into the calling object's address space.

The GRT uses an internal service to locate the object. It then maps the object into the current address space (c.f. section 2). Once mapped, the GRT upcalls the object's `GetArrayRef()` function. This function returns the address of an array of references contained within the object. This array, in conjunction with the `GetVMemAdd()` upcall allows the GRT to obtain a list of pointers within the object to other objects.

For each reference within the object the GRT has a choice, either it can go ahead and load the referenced object, updating the pointer[9] or it can arrange that the use of the reference causes an object fault[10].

Once loaded, an upcall is made to the function `AfterMap()` which is responsible for ensuring that any context dependent data, such as file descriptors as described above are dealt with (3.1).

Once this has been carried out, the GRT can then assume that the object is correctly mapped, and initialised, and can upcall the function `Dispatch()`, passing it the `ActivationData` structure obtained in the `Invoke` downcall which allows the language level to call the correct method, after it has carried out any necessary work connected with stack management.

4.2.2 Map the Object Remotely

The scenario for dealing with an object that is mapped remotely is similar to that of a local invocation in terms of dealing with the mapping, converting global references etc. However, the difference is in the propagation of the invocation frame.

When the object is mapped remotely, the GRT transfer processing from the current address space to the remote address space[11]. In this case the `ActivationData` must be transferred to the the remote address space.

---

[8] for example, because of heterogeneity

[9] and recursively load other referenced objects.

[10] Again via virtual memory techniques or using proxy objects.

[11] A lightweight invocation may occur if the object is in another address space but on the same site.

The main complication arises when we have to deal with heterogeneous architectures. In this case both the object itself (if it is being mapped from the storage subsystem) and the invocation data may need to be converted to the appropriate format.

As far as the actual object is concerned, in addition to the mapping and series of associated upcalls described in section (4.2.1), we may need to upcall `ConvertFormat(from, to)` which converts to or from a standard architecture independent format. In the case of mapping the object, it will be converted from a standard representation on disk to the required representation for the node.

Dealing with architectural differences at the level of the invocation data is carried out in a similar manner. The GRT, when it recognises that it is sending to a different architecture ensures that the outgoing invocation frame is converted to an architecture independent one using the `ConvertActivationDate(activationData, from, to)` upcall. On receipt of an incoming invocation, the standard representation is converted into one suitable for the receiving machine. Invocation then proceeds as before using the `Dispatch` function.

5 Conclusions

In this paper we presented the mechanisms provided by the Comandos platform to support a variety of languages in the Comandos distributed environment.

We are currently in the process of implementing the Comandos kernel and GRT on Unix, CHORUS and Mach 3.0. Language specific runtimes for C++, Eiffel and Guide are also being implemented. A prototype implementation, known as Amadeus, which supports distributed persistent C++ above Unix is currently available to interested parties for evaluation and experimentation.

6 Acknowledgements

The authors acknowledge the fruitful interactions with all of the participating institutions in the Comandos project and in particular with Bull and IMAG in Grenoble; GMD in Bonn; and the University of Glasgow.

The authors particularly acknowledge the contribution made by Andre Kramer while working at Trinity College.

References

[1] “Functional Specification of Release-2”, Esprit Project 2071 - Deliverable D1-T2.3, March 1991.

[2] “Comandos Language Reference Manual”, Esprit Project 2071 - Deliverable D4-T2.3, March 1991.

[3] I. Goldstein, *OSF RI Notes*, No 3, June 1990.

[4] B. Meyer “Object Oriented Software Construction”, *Prentice Hall*, 1988.

[5] B. Stroustrup “The C++ Programming Language” *Addison-Wesley*, 1987.

A Class object

This class describes the upcalls which must be provided by a Comandos object. Each language level object must have the following set of methods available when it wishes to be managed by the system:

```
// Structure used to pass parameter frames between language specific runtimes
// and the generic runtime

struct ActivationData {
    char *operationName;    // String containing the name of the operation
    int   parametersSize;   // Size of the parameters
    char *parameterBlock;   // raw data containing the parameters
    GName targetObject;     // Global name of the invoked object, defined
                            // later.
};

class Object {

public:

                      ~Object(Error *error);
  virtual void         AfterMap();
  virtual void         BeforeUnMap();
  virtual void         Trap(int trapName);
  virtual void         Dispatch(ActivationData *actData, Error *error);
  virtual GName        GetClass();
  virtual VMemAdd      GetVMemAdd();
  virtual RefLocation *GetArrayRef();
  virtual VMemAdd      ConvertFormat(Architecture *from,
                                Architecture *to, Error *error);
  virtual VMemAdd      ConvertActivationData(ActivationData *actData,
                                Architecture *from,
                                Architecture *to, Error *error);

  virtual String       ExplainError(Error *error)
  virtual void         DisplaySelf()
};
```

`Object` defines the set of methods that all Comandos objects must support to be correctly managed by the system. Typically, these methods are generated automatically or are inherited by each object in the system.

AfterMap is up-called immediately after the mapping of the object into an address space. This allows the programmer to update any address space dependent data within the object.

BeforeUnMap is up-called immediately before unmapping the object from an address space. It is used for the same purposes as the `AfterMap` method.

Trap is up-called when a hardware or system exception is raised, and allows the language run-time to handle exceptions caught by the virtual machine.

Dispatch is up-called by the Generic Run-Time when an incoming cross-address-space invocation for the object is received, and when the object is mapped locally.

GetClass returns the global name of the class object for this object.

GetVMemAdd retrieves the address of the object instance data.

GetArrayRef is up-called when the generic run-time needs to know the location of references within an object. The structure returned is an array containing header information and a list of offsets, containing the location of references within the object. The final address of the references is obtained by adding these offsets to the object location, returned by the `GetVMemAdd` up-call.

ConvertFormat is up-called when an object's representation need to be converted to one suitable for another architecture. Conversion between heterogeneous architectures is performed in two steps: from current format to a standard one (within the calling address space), and then from the standard to the desired format (within the called address space). Thus, one of the Architecture arguments is guaranteed to be a standard one.

ConvertActivationData is up-called to convert the parameters contained within an `ActivationData` structure. The semantics of this primitive is similar to the previous one.

B Class context

An instance of the `Context` class is created and properly initialised by the GRT when an address space is created. The operations available on this class describe the downcalls available to language specific runtimes.

```
class Context {

public:

  VMemAdd    Create(int size, Error *error);
  void       Register(VMemAdd vMemAdd, GName gName, Object *object,
                                                      Error *error);
  GName      AssociateGName(VMemAdd vMemAdd, Error *error);
  void       Invoke(GName gName, ActivationData *actData, Error *error);
  Bool       IsLocal(GName gName, Error *error);
  void       LoadObject(GName gName, Error *error);
  void       ReleaseObject(GName gName, Error *error);
  void       SetCurrentObject(Object *object);
  void       Group(GName *gName, U32 nbGname, Error *error);
```

```
  virtual String       ExplainError(Error *error)
  virtual void         DisplaySelf()
};
```

An instance of the **Context** class is created and initialised by the GRT when needed.

Create is called to reserve a memory region for holding objects. Typically, languages use this primitive to allocate space for their objects. The `Create` primitive returns the address of the allocated memory.

Register is called to make the specified object (represented by the `VMemAdd` parameter) known to the generic run-time. The `GName` parameter indicates the global name of the class for this instance. The `Object` parameter contains the set of up-calls required for in the registration procedure, and must be previously created and initialised by the corresponding language specific run time.

AssociateGName() associates a global name to the object located at `VMemAdd` address. The global name is returned.

Invoke causes invocation to be transferred to the object specified by `GName` with the prameters specified by `ActivationData`.

IsLocal returns true if the object specified by `GName` is mapped in the current address space.

LoadObject maps the designated object in virtual memory so that it can be scanned without being invoked. The object stays locked in memory until it is explicitly released with the `ReleaseObject` primitive.

ReleaseObject informs the system the designated object is no longer needed in memory, and therefore can be stored back to disk.

SetCurrentSignalObject informs the system that the designated object should become the default handler for hardware traps occuring within the current address space.

Group is called to inform the GRT that the list of objects should be stored together. This call is advisory.