Supporting Distributed Applications in the Amadeus Environment

Chris Horn and Vinny Cahill *
Distributed Systems Group,
Department of Computer Science,
Trinity College Dublin,
IRL - Dublin 2.

Abstract

Distributed programming is becoming commonplace, typically based on remote procedure call (RPC) and lightweight threads packages, possibly with an underlying distributed file service. In this article we argue that there are significant merits in providing an integrated distributed application environment, rather than merely augmenting one or more programming languages individually with an RPC package, threads support and remote file access.

Our Amadeus environment is a proof of concept implementation, currently extending C++ for distributed and persistent programming above Unix[1].

1 Introduction

Distributed programming is becoming commonplace, typically based on remote procedure call (RPC) and lightweight threads packages, possibly with an underlying distributed file service[18]. In this article we argue that there are significant merits in providing an integrated distributed application environment, rather than merely augmenting one or more programming languages with an RPC package, threads support and remote file access.

Applications designed for distributed environments are given a number of opportunities, but must also face up to a number of realities, which are intrinsic in distribution, and which are precluded or largely ignored in classical centralised systems. Some of the opportunities, and challenges, are as follows:

Concurrency A classical centralised system is a single processor, above which pseudo-concurrency is added by task switching. A distributed environment, like a multiprocessor system, provides inherent concurrency. Managed correctly, this provides an opportunity for gains in responsiveness and throughput.

Dynamic reconfiguration and migration Human organisations are distributed, and can benefit from locally available computing support. Information must circulate through the

---

[*] e-mail: horn@cs.tcd.ie

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

system, and migration of data and services to those who currently must work with them is attractive. Likewise, continuously operating applications can benefit from online upgrades and reconfiguration. Finally the overall environment can be incrementally augmented as dictated by economic, social and technological developments.

Increase in failure modes Consequent from the remote use of services is the disadvantage of increased probability of failure. In one sense failures in a distributed system are more unpleasent than on a single host, since the former are typically partial: one part of the computation stops whilst the remainder continues. Whilst this gives an opportunity to circumvent the failure, in general it is also more complex to design and program applications which are resilient to failure. On a single host, environment failures are often treated as total, inhibiting all further computation.

Security The use of communication lines increases the risk for exposure of sensitive information. Do the benefits of distributed computing frankly compensate for the risk for an organisation, and how minimal can that risk be made ?

Storage Many applications, even compute bound applications like some scientific programming, require access to stored data. In a distributed environment, the opportunity for information sharing and exchange increases, and it appears even more natural to seek a seamless approach to acquisition and management of stored information.

Latency Interacting with a remote service costs in time. Even should the technology of communication media bandwidth continue to improve, it is reasonable to assume that the cost of a remote interaction, including software overheads, will always be far larger than a local one.

Heterogeneity Single processors, and usually even multiprocessor systems, consist of homogeneous CPUs and operating system. Distributed systems are invariably heterogeneous in the hosts; operating systems; programming languages; communication protocol stacks and media; and storage subsystems.

These opportunities and problems are well recognised in distributed programming, and most have well understood technical solutions. Our contribution is to try and provide support for these mechanisms for various programming languages, in a way which improves the ease of programming distributed applications. We discuss our strategy with respect to the issues outlined above in the next section.

2 Our approach

Amadeus is the public name of an internal development project undertaken at Trinity College Dublin since 1989[2]. Our project has benefitted from participation in several projects, and in particular the Esprit Comandos project[1].

Our general approach, like those of several other projects [2][1][20] is to partition distributed applications into a set of co-operating "objects".

In particular we have been keen to support distributed programming in a number of programming languages. Our philosophy is, where feasible, to make a minimal number of changes to each programming language.

---

[2] Dublin being the current European city of culture, and it also being Mozart's 200th anniversary!

2.1 Object models

We believe that object oriented programming languages, which support inheritance and subclassing, are particularly interesting candidates for extensions for distributed programming. Clearly an object based language encourages application programmers to partition their programs as sets of co-operating objects. However subclassing is also important for distributed applications, since it allows a called entity – loosely, a "server" – which provides more functionality than actually required for a particular interaction to nevertheless be used. The more traditional binding services of RPC packages on the contrary only allow an exact match of the desired and professed interfaces[4]. For example, a "client" may require a service to send an electronic mail message to a particular recipient. The fact that a particular chosen service can not only send electronic mail messages, but also optionally file copies of sent messages, should clearly not preclude use of that service for merely sending messages. Hence the notion of marrying of desired and actual interfaces is important: and subclass based languages incorporate this notion. In practice, the notion of subclassing frequently differs in detail from one language to another, a point to which we return later below in section 2.6.

It should also be noted that languages which support inheritance can support multiple interfaces – even interfaces of the same "name". Consider a simple example, such as a service to store or retrieve an electronic mail message from a filing cabinet. The filing cabinet maintains the messages in a list. A simple store/retrieve interface for the filing cabinet could be implemented in two ways: to store and retrieve a message to or from the head of the list; or to do so to or from the end of the list. Thus in general, an interface can be implemented in several ways – perhaps simultaneously even by the same object. By describing the interface as an abstract class (in C++[8] terminology, or "abstract superclass" in Smalltalk[9] terminology, or "deferred class" in Eiffel[16] terminology), having a subclass for each implementation of the interface, and then merging these into a common subclass (a sub-sub class of the original abstract class), the same interface can be offered in different ways by a single object.

2.2 Transparency and Binding

We believe that the fundamental functionality that should be provided in a distributed, object oriented environment is transparent access via an object reference. By an object reference, we mean for example a pointer in C++. By transparent access we mean that the access to an object via an object reference when that object is remote from the reference is syntatically equivalent to when it is local, and does not require code to be rewritten, recompiled or even relinked if the proximity changes. Hence, (re-useable class library) code can be written which can manipulate objects independently of their current relative position. This has the further advantage that the location and physical configuration of objects can be changed dynamically, without impacting compiled code. As a further consequence, application designers are encouraged to identify in their design interfaces potentially remote objects – at run-time, such an object may actually be local to or remote from the entity which holds a reference for it. In principle, if it becomes established that an object will always be local, then compile or link time optimisations are possible to exploit that fact[12].

Obtaining an object reference to a (possibly remote) object is equivalent to binding a client to a server in a traditional RPC environment. Object references may be communicated from one object to another, so effecting further bindings. As noted in section 2.3, references may also be stored and later retrieved and re-used – this facility is not normally available in RPC packages.

Using traditional RPC facilities, it can be relatively involved to create and install a new service. However, in an object based environment, the creation of a new object creates a new object reference. Thus new services, represented by new objects, can be simply created.

Object references can also be obtained by consulting some directory system. In our terms, a directory system is one which can map and remember translations between names (and/or other attributes) and object references. A directory service may well offer search and associative retrieval services, and in fact be a database system. The directory system can be built above our environment (as an application using the environment itself), or can be external to it. Both approaches have been used, as is reported later in section 3.2.

In summary, binding in our environment can be either implicit or explicit. Implicit binding occurs when object references are manufactured, retrieved from storage, or otherwise communicated. Explicit binding occurs when a directory service is explicitly consulted for some object.

The object reference mechanism is naturally symmetrical[13]. In a classical RPC environment, a server must first export a reference to itself, which its potential clients can subsequently attempt to import. With this approach, a client which wishes the server to contact it back again might have to export a reference to itself for the server to import. A client may wish the server to contact it for a variety of reasons, including a "call-back" when particular events occur within the server. The binding mechanism is asymmetrical. It is in principle possible to overcome this limitation if the server is able to discern from a client request which client is calling it: further, the server must somehow know, or be able to find out, the call-back operations which the client supports. In an object environment, such as ours, a client can explicitly pass an object reference to itself to a server, allowing call-backs to be easily implemented.

A further point is that re-binding is unnecessary if an object moves. That is, our environment makes a best-effort attempt to locate an object given an object reference to it. If an object moves its location, object references for it will remain valid, and the environment will transparently find the new location. In a traditional RPC system, a server which moves whilst there are bindings from clients to it would cause those bindings to be lost; the clients would have to explicitly rebind to the relocated server.

An admitted limitation of our current implementation of migration is that an object cannot be moved whilst there are outstanding invocations upon it, unlike for example [12].

2.3 Storage

We strongly believe that any serious distributed (and indeed, non-distributed) environment should provide support for persistent and long-lived data. In programming environments, it is often necessary to manipulate many persistent objects, such as application data, source code, interface specifications, and compiled code. The advantage of providing persistence for programmers is that complex data and object structures do not have to be rebuilt every time the data are retrieved from storage.

In keeping with a minimal impact on programmers of our extended environment, we match our extensions for potentially remote objects with similar ones for potentially persistent objects. That is, given an object reference (such as a C++ pointer to an object), the designated object might actually not be currently mapped into (virtual) memory but instead be stored in (e.g.) a file store. An attempt to use that object will result in its being transparently retrieved from the store, and loaded into memory in a manner not unlike the classical "page fault" of a virtual memory system. Further, the object may be later unmapped from memory and returned to storage, without requiring the object reference to it to be modified by the programmer.

Thus in an our environment an object reference in a programming language (e.g. a C++ pointer) may refer to a remote or local object, in either case in virtual memory or on disk. In all cases, the current location of that object is transparent to the use of the reference so as to access the object.

2.4 Failures

As noted in the introduction, distributed applications are exposed to more failure modes than traditional single host ones. Like most RPC packages, we accept that having requested a particular operation on a remote entity, there may not only be a successful outcome, but possibly also erroneous or unusual outcomes[5]. However we firmly believe that the management of these conditions should wherever possible be reflected naturally in the exception model of each applications programming languages. For example, Eiffel has a reasonably sophisticated exception mechanism, and errors arising from our environment can be reflected to an Eiffel programmer as extensions to the kernel library exception class. In languages which do not have their own exception model, environment errors can be, for example, managed by predefined routines which may be tailored by the programmer (e.g. a table of pointers to functions in C).

2.5 Concurrency

The usual exploitation of concurrency in a distributed environment is based on lightweight threads, or asynchronously requested operations, or a combination of both. Since we advocate that use of a potentially remote entity be transparent by default – the entity may actually be local or remote, and indeed its proximity may change dynamically – we recognised some time ago[11] that an extension to the baisc thread model was needed. A synchronous request made by one thread to a remote object is served by a second thread in that object: however conceptually, particularly if remote requests are transparent, there is only a single logical thread transcending the calling and called object. We call such a logical thread an "activity".

An activity itself is an object in our model. It has a class interface offering operations to, for example, suspend, resume or kill it. Given a reference to an activity object, the object itself can be local or remote from that reference. References to activities can be stored and later retrieved and re-used. Further, the operations available upon it can be requested independently of the current execution point of the activity.

An activity is created by associating it with an operation request on an object. It terminates when that request is finished, and its persistent value is the outcome of that operation request. Note that an activity can of course be created by the current object on itself: in particular this can be done during the initialisation of an object (e.g. within a C++ constructor), so building an "active" object[17].

Since operation request and execution is synchronous by an activity, we model asynchronous request by creating new activities, one per asynchronous request. The outcome of an asynchronous request may be discarded, or it may be later retrieved (from the activity object). Forking an asynchronous request and later making a rendezvous with its outcome is termed a "deferred synchronous request", and is somewhat similar to the futures mechanism of [10] or promises of [15]. Asynchronous operations appear to be essential in a distributed environment, for which communication latencies are frequently high.

Our environment currently provides extremely low level synchronisation support. Some programming languages provide synchronisation support as a part of their language definition, and their runtimes [7]: we believe such runtimes can exploit our low level synchronisation support. In other languages, such as C++, various libraries are available whcih support concurrency and synchronisation: again we believe such libraries can be ported to our environment.

One aspect of concurrency which is challenging is to identify rules by which the concurrency constraints on a request interface can be met by those on a professed interface – i.e. identifying subclassing rules when concurrent execution of operations is constrained. We are actively experimenting in the area, based on predicate calculus, but as yet we have no implementation results to report.

2.6 Heterogeneity

The problems of converting between machine representations of data are well understood, and are solved in RPC packages. However there are further considerations in a distributed object environment such as our own. One is that interworking of objects from the same applications may require that different executable images of the same source language class are needed. At runtime, the correct binary image must be located and dynamically link loaded. A second issue is that if objects can migrate, their representation may need to be changed as they move from one machine to another. Both of these are receving our attention in our implementation.

Another issue results from interworking of different languages and type systems. It is clearly possible to insist on a canonical definition, such as an RPC Interface Definition Language, and rely on preprocessors to generate appropriate stubs in each supported language[4]. One problem we perceive with this approach is that of matching the types and values of a particular programming language with that of the IDL definition and stub usage: the IDL type system may not exactly match that of the language. Similar problems can occur with embedded query languages, such as embedded SQL[6]. Although we have not solved the problems, we believe it interesting to take the reverse approach: to take a language definition of some interface (e.g. a class) rather than an IDL one, and to automatically generate shadow interfaces for it in other languages. One of the interesting challenges is to do so in conformance with each languages type system: we are investigating builidng canonical descriptions of the type systems of various languages, as a basis for translation of interfaces.

2.7 Security

Secure interworking appears of fundamental importance in most distributed applications. Authentication and encryption services can be added to RPC packages [18]. In an object oriented environment, it is natural to consider whether access control can be based on the operations of an interface. In a potentially fine grained object environment, such as our won, it is clearly undesireable to impose protection overheads on each object invocation. It is also clear that protection mechanisms cannot be purely based on user mode software solutions. For example we reject the idea that protection can be implemented purely by a specific compiler and language run time mechanism, since our environment should not preclude the use of any programming language.

Our apporach is to allow a combination of objects to be protected together. We call such a combination an "extent". Exported references to objects within the extent can be individually protected, to the level of individual operations in each interface. At runtime, a given extent of objects may be partially mapped (using the object faulting mechanisms of section 2.3) into one or more process address spaces. Object usage within an extent is not mediated by the access control mechanism; cross-extent usage however is. We beleive that the extent mechanism, when coupled with appropriate authentication and encryption mechanisms, should provide a workable solution to protection in our environment.

3 Implementation

The basic strategy we have adopted towards supporting multiple programming languages, each with extensions but in its own philosophy and religion, has been to identify a common layer above which language specific runtime features are built [22]. The common layer is generally called the "generic runtime", although internally it is itself partitioned into a runtime layer and core services. The core services include support for the concurrency model and communication.

Each supported language above the generic runtime requires interfacing of its own specific runtime to our generic one. For languages which do not have a runtime as such, additional code may need to be produced during the compilation process to interface to the generic runtime.

The generic runtime provides facilities to create persistent objects, to manage linkages when objects are loaded dynamically from storage, to initiate remote invocations, and to dispatch incoming remote requests. To achieve these, it may need to interact with a particular language specific runtime.

When an object is in memory, any object references it contains are normally formatted in a language specific way. For example, if the object is from a C++ class, its object references are (e.g. 32bit) memory pointers. When an object is stored, its object references are much larger since each must be able to locate the designated object within the distributed and persistent system. A limitation of, for example, 32 bits is overly restrictive for that purpose. Rather than reformatting the entire layout of an object when fetched from the store or stored back again, the language specific object reference is replaced by an offset value. This offset is to an adjunct to the object which gives the full "system-wide" object reference. Location of, and conversion between the formats of, object references is performed by the generic runtime in conjunction with each language specific one.

3.1 Trapping object faults

Some researchers are currrently investigating the use of distributed shared memory to support distributed programming[14][23][3]. On the other hand, there is a certain scepticism by some who believe that such approaches can never scale well, and are finding it difficult to address heterogeneity. Instead more classical RPC approachs appear more fruitful. Our own position on this debate is that in our environment such as our own, either approach can be engineered, and that conceivably an application programmer need not be aware of which is adopted.

One issue that our implementation faces is how to represent a remote object in the local address space (e.g. of a Unix process) in a transparent way. Usage of the remote object should appear similar to a local one; indeed the remote object may subsequently move and actually become local. Attempts to use the object whilst it is remote must be trapped, and converted into remote requests or deferred until the object can be moved into the local space. Similar issues arise for detecting attempted accesses to an object which is currently absent and on storage.

There are a few approaches to this representation and detection problem. One approach, which we have implemented, is to represent the remote object by a "shadow", or "proxy"[20], whose executable class code is different from that of the true object. Using the dynamic binding of an object oriented language allows the transparent replacement of the proxy executable code by the code of the true object, should the true object be brought into the address space.

A second approach is to rely on the support for memory management handlers external from the operating system kernel, as is becoming available in some kernels such as OSF/1[3] and CHORUS[4]. In this case, again a shadow representation of a remote object is present in the address space, but on a protected memory page. Accesses to it cause a protection fault, which a "user-mode" handler can catch and translate into a remote access, fetch from store, or migration to the local space as in the scheme above. Although we have yet to gain experience with this approach it is clearly feasible.

As a consequence of the ability to translate attempted accesses to an absent object into message requests sent to a remote machine, and to alternatively migrate the absent object and map it locally, we suggest that the approach subsumes the RPC v. DSM debate: both are possible and have their relevance, and both can be used to support a distributed object environment.

---

[3] OSF is a trademark of the Open Software Foundation.

[4] CHORUS is a trademark of Chorus Systèmes.

3.2 Storage and Naming

Our current implementation uses Unix files to store persistent objects. Many objects can be stored into the same file, as a “cluster”. Mapping an object from such a cluster causes the entire cluster to be mapped into virtual memory.

For a distributed store, we currently rely on a distributed file system, such as NFS[19]. However we have also implemented a distributed store of our own, and not relied on having all sites interconnected by a distributed file system [21].

For directory naming and binding of objects, we again currently use the Unix file store. That is, any object reference to a particular object can be stored into a Unix file, and thus give a (Unix file path-) name. Our colleagues at Bull in the Esprit Comandos project have taken a different approach, and built an X.500 prototype Directory Service above their Guide environment.

4 Status

Our current implementation is above Unix. Each application is linked with the Amadeus runtime library, and further there currently needs to be an Amadeus server daemon running on each host. The daemon is chiefly responsible for communication binding services, maintaining tables of well-known network addresses as other daemons on remote hosts are brought up or down.

Our first implementation was above Digital’s Ultrix, and includes its own RPC and threads packages (since currently Ultrix does not provide these). We are investigating replacement of these particular modules by other industry standard packages.

Our first supported language is C++, based on the Free Software Foundation’s g++ compilation system. We are interested in supporting other C++ compilers (notably that from AT&T), and other languages including in particular Eiffel.

The current software package has been distributed at the time of writing to a number of sites within Europe for evaluation and feedback. Previous versions have been demonstrated at, for example, the annual Esprit Conference weeks in Brussels, and further demonstrations are planned.

Some small example C++ programs for our environment are included in the appendix.

5 Acknowledgements

Within TCD, we would greatly acknowledge in particular the work and input of Sean Baker, Ann Barry, Dominic Herity, Andre Kramer, John Moreau, Annrai O’Toole, Mark Sheppard, Gradimir Starovic, Brendan Tangney, and Bridget Walsh, as well as the various Ph.D. and M.Sc. students who from time to time have worked on the project. Finally all the TCD team owe our special thanks to the work of Neville Harris as TCD project director.

We are also conscious of our indebtedness to our many colleages outside of TCD with whom we have collaborated in Esprit work. We would in particular express our appreciation to Jose Alves Marques, Roland Balter, Michel Gien, Paulo Guedes, David Harper, George Jelatis, Sacha Krakowiak, Reinhold Kroger, Roger Lea, Bernard Martin, Helmut Meitner, Micheal Mock, Xavier

Rousset de Pina, Pedro Sousa, Hans Spappens, Kurt Stadler, Gerard Vandome and John Winterbotham.

A Some programming examples

C** is an extension of the C++ programming language suitable for writing programs which run in the Amadeus environment.

A.1 Trivia

A.1.1 Persistence

Consider the following C++ program:

```
#include <stream.h>

class integer {
        int value ;
public:
        integer (int Value=0) { value = Value ; }
        int operator ()() { return value ; }
        void inc() { value++; }
} ;


main()
{ integer *i;

  i = new integer ();
  cout << "i is " << (*i)() << "\n";
  i->inc();
}
```

Clearly, if we compiled this program as “foo” and ran it twice, we would obtain:

```
% foo
i is 0
% foo
i is 0
%
```

We can compile and run this same program using C** too, and (naturally) obtain the same result. However, we can alter the behaviour of class `integer` so that its instances are *persistent*.

If we make class `integer` persistent (in the manner explained later below), then when we run the (new) foo, we will obtain:

```
% foo -reset
i is 0
% foo
```

```
i is 1
% foo
i is 2
%
```

Note that there is an optional switch `-reset`. If given, it indicates that the application (here `foo`) should do a full "initialisation".

A.1.2 Persistence via a `permclass`

In order to make our class `integer` persistent, we can either derive a persistent class from it, or modify the class attributes. The latter is explored below.

In C**, a persistent class is any `permclass` or any class which inherits (directly or indirectly) from any other `permclass`. `permclass` is a new keyword introduced in C**.

```
#include <amadeus.h>
#include <stream.h>

permclass perm_integer {
        int value ;
public:
        perm_integer (int Value=0) { value = Value ; }
        int operator ()() { return value ; }
        void inc() { value++; }
} ;


main ()
{ perm_integer *i;

  i = new perm_integer ();
  cout << "i is " << (*i)() << "\n";
  i->inc();
}
```

Note that, as in this example, persistent objects should always be allocated on the heap using `new`.

A.1.3 Binding to the Persistent Store

In Amadeus every object whose class is a `permclass` (base or derived) is said to be persistent. Specifically, it persists if it itself is registered in the Amadeus Name Store, or if it is (transitively) referenced by a pointer from an object which itself persists. If the object is not registered or referenced in this way, it will (eventually) be discarded by the Amadeus garbage collector. Note that for an object to persist it does not necessarily need to be registered itself in the Name Store – it suffices for it to be referenced by any other persistent object.

In the version of Amadeus above Unix, the Amadeus Name Store is implemented above the Unix file system. Each object registered in the Name Store can be given a Unix file name. Thus, the example of a persistent integer (c.f. section A.1.2) can be completed:

```
main()
```

```
{ perm_integer *i;

  if (amadeus.reset())                        // whether -reset given
    { i = new perm_integer();
      amadeus.record ("myinteger.ns", i); }    // bind to Unix filename

  // normal case
  if (! (i = amadeus.lookup ("myinteger.ns"))) // lookup Unix filename
   { cout << "Object lost from name store!\n"; exit(-1); }

  cout << " i is " << (*i)() << "\n";
  i->inc();
}
```

Note that the programmer does not need to explicitly save or fetch a persistent object; when called, `record` and `lookup` do *not* themselves cause an object to be moved to or from the persistent store.

A.2 Remote use

As well as extending C++ with support for persistence, C** allows specified member functions of a class to be used remotely. The keyword `global` is used to mark such member functions.

Returning to our integer example, we could mark member functions for remote use:

```
class system_wide_int {
        int value ;
public:
        system_wide_int (int Value=0) { value = Value ; };
        global int operator ()() { return value; }
        global void inc() { value++; }
} ;
```

To create an integer of this class for remote use:

```
system_wide_integer *i = new system_wide_integer();
```

The pointer to the new object can subsequently be communicated to other nodes in the distributed system, and used to access the object from any point in the distributed environment:

```
cout << "i is " << (*i)() << "\n";
```

Thus the use of a pointer to a remote object in C** is syntatically identical to the usual use of a pointer in C++: the current location of a (potentially remote) object is not apparent from a pointer to it. Indeed, the location of the object may change at runtime, transparently to other objects holding pointers to it.

B A moderate example

Banking is a typical and everyday application which can be implemented using distributed persistent objects. Account holders make enquiries, deposits, and withdrawals using automatic teller machines. The account holder needs to have access to the account balance at different locations; the balance must be persistent. Accounts are protected by "Personal Identity Numbers", or PINs:

```
//
// class Account declaration
//
enum result {ok, BadPIN, InsufficientFunds} ;

typedef struct rep {result r; int sum;} report ;

permclass Account {
        int pin ;                   // PIN for transaction authorisation
        int bal ;                   // Account balance
    protected :
        virtual int Status () {return bal;};     // Derived classes may find
        virtual void Adjust(int v) {bal += v;};  // and alter balance
    public :
        Account(int PIN) ;
                // Create account with given PIN

        global result Transfer(int amount, int PIN) ;
                // credit (amount>0) or debit account.
                // PIN must be correct

        global report Balance(int PIN) ;
                // Enquire account balance.
} ;
```

The `PIN` is supplied in the constructor and may not be changed or revealed; it is used to authenticate withdrawals and enquiries.

The balance `bal` represents the account's balance in the smallest currency unit (penny, cent, centime, or appropriate unit).

The account has three publicly accessible methods: a constructor, a `Transfer` method encompassing deposits and withdrawals, and a `Balance` method. Given a pointer to an `Account` object anywhere in the distributed system, whether that `Account` object is currently resident in memory or on disk, the member functions `Transfer` and `Balance` can be used to access it – assuming naturally that the correct `PIN` is provided.

We could implement the `Account` class as follows:

```
//
// permclass Account implementation
//

Account::Account(int PIN)
  { pin = PIN ; bal = 0 ; }

result Account::Transfer(int amount, int PIN) {
    if (PIN != pin)                           // require correct pin
        return BadPIN;
    if (amount < 0 &&                         // If transfer is debit,
         Status() + amount < 0)               //   and sufficient funds
           return InsufficientFunds;
```

```
    Adjust (amount) ;                               //   do transaction
    return ok;
}

report Account::Balance(int PIN) {
    report r;
    if (PIN != pin)
      {r.r = BadPIN; r.sum = 0;}
    else
      {r.r = ok; r.sum = Status();}
    return r;
}
```

References

[1] J. Alves Marques, R. Balter, V. Cahill, P. Guedes, N. Harris, C. Horn, S. Krakowiak, A. Kramer, J. Slattery, and G. Vandome, "Implementing the Comandos Architecture", *Esprit'88: Putting the Technology to Use*, pp1140-1157, 1988 North-Holland.

[2] "ANSA Reference Manual", Available from *Architecture Projects Management Limited*, Cambridge, UK.

[3] J. Bennett, J. Carter and W. Zwaenepoel "Munin: Distributed Shared Memory based on Type-Specific Memory Coherence", *ACM SIGPLAN Notices*, Vol 25, No 3, pp168-176, March 1990.

[4] A. Birrell and B. Nelson, "Implementing Remote Procedure Calls", *ACM Transactions on Computing Systems*, Vol 2, pp39-59, Feb. 1984.

[5] D. Cheriton "Making Exceptions Simplify the Rule (and Jusitify their Handling)", *Proceedings of IFIP'86*, North-Holland.

[6] C. J. Date "A critique of the SQL Database Language" *ACM SIGMOD*, Vol 14, No 3.

[7] D. Decouchant et al., "Implementation of an Object-Oriented Distributed System Architecture on Unix", *proceedings of Autumn EUUG Conference*, 1988, Cascais Portugal, October 1988.

[8] M. Ellis and B. Stroustrup, "The Annotated C++ Reference Manual", Addison-Wesley, 1990.

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

[10] R. Halstead "Multilisp: a language for concurrent symbolic computation", *ACM Transactions on Programming Languages and Systems*, Vol 7, No 4, Oct 1985.

[11] C. Horn and S. Krakowiak "Object Oriented Architecture for Distributed Office Systems", *Proceedings of 1987 Esprit Conference*, North Holland, September 1987.

[12] E. Jul, H. Levy, N. Hutchinson and A. Black "Fine-Grained Mobility in the Emerald System", Technical report 87-02-03, University of washington, Feb. 1987.

[13] H. Levy and E. Tempero "Modules, Objects and Distributed Programming: Issues in RPC and Remote Object Invocation", *Software Practice and Experience*, Vol 21, No 1, Jan. 1991, pp77-90.

[14] K. Li and P. Hudak "Memory Coherence in Shared Virtual Memory Systems," *ACM Transactions on Computer Systems*, Vol 7, No 4, pp321-359, Nov. 1989.

[15] B. Liskov and L. Shrira "Promises: Linguistic Support for Efficient Asynchronous Procedure Calling" *Proceedings of the SIGPLAN-88 Conference on Programming Language Design and Implementation*, June 1988, pp260-267.

[16] B. Meyer "Object Oriented Software Construction", *Prentice Hall*, 1988.

[17] O. Nierstrasz "Hybrid – A Language for Programming with Active Objects" *Objects and Things*, University of Geneva, 1987.

[18] "OSF/DCE Release Notes", *Open Software Foundation*, October 1990.

[19] R. Sandberg "The Sun Network File System: Implementation and Experience", *Proceedings of EUUG Spring 1986 meeting*, Florence.

[20] M. Shapiro et al "SOS: An Object Oriented Operating System: Assessment and Perspectives", *Computing Systems*, Usenix, Vol 2, No 4, Fall 1989.

[21] G. Starovic "The Storage Subsystem for the Oisin Kernel", M.Sc. thesis, Trinity College Dublin, 1989.

[22] M. Weiser, A. Demers and C. Hauser "The Portable Common Runtime Approach to Interoperability", *Xerox PARC*, March 1989.

[23] S. Zhou, M. Stumm and T. McInerney "Extending Distributed Shared Memory to Heterogeneous Environments", *Proceedings of the 10th Conference on Distributed Computing Systems*, pp30-37, June 1990.