Towards Generic Support for Distributed Information Systems

Vinny Cahill, Chris Horn and Gradimir Starovic

Distributed Systems Group, Dept. of Computer Science, Trinity College, Dublin, Ireland.

Abstract

We are concerned with providing support for a range of object oriented programming langauges to be used in multi-user, multi-machine, heterogeneous environments requiring associative access, as well as concurrency and storage management. In order to operate in this environment the implementations of current object oriented languages must however be extended. Our goal is to provide a generic runtime support system open to a range of programming language implementations and requiring no (or only minor) alterations to each supported language.

1 Introduction

In considering strategies for building distributed applications, object oriented languages appear attractive. Design of distributed applications is concerned with clear identification of encapuslated resources. Further, naming of encapsulated resources, transmission of such names, and transparent access to remote resources based on their names, are all essential. Object oriented languages can, in principle, address all of these concerns.

Object oriented languages also appear attractive for many information system applications. The ability of object systems to model some complex real world data is clearly an advantage over more classical techniques based solely, for example, on the flat relational model. Object oriented languages can provide a basis for such applications, if they can be extended to operate in persistent environments and so provide modelling and query facilities, as well as concurrency and storage management.

Although apparently attractive for distributed and persistent applications, the implementations of current object oriented languages must nevertheless be extended if they are to be used in multi-user, multi-machine and heterogeneous environments. A fundamental challenge in adapting object oriented languages for distributed and persistent environments is how to do so without imposing entirely new programming language models and constructs on the base language.

In adapting a specific language, decisions must be made as whether it is necessary to restrict the use of certain language constructs, or to change the semantics of certain constructs, as well as perhaps adding new constructs to the language. The correct answer in each case should depend on the semantics, complexity and ethos of the language in question, and not necessarily on the underlying runtime support environment. Thus any such multi-language support environment must provide a range of mechanisms, suitable for different languages – in effect it must be open to different tradeoffs for key issues.

To support such an environment while maintaining our overall goal of no (or just minor) alterations to each supported language, we need to identify the functionality required of a support system generic to a range of programming languages. Thus our overall architecture consists of three fundamental levels: alternative language environments (i.e. compiled code and language runtime systems); an underlying "generic" support environment; and finally the host operating system. The generic support system requires a clear separation of concerns from the runtimes and compiled code for each language [11]. Nevertheless the generic support system must be able to interact with each language system – for example to dispatch incoming remote invocations. Thus the question of where to place the interface to the support layer arises. What functionality should be left to the language? What functionality should be available to the language designer from the support layer, and what functionality can be expected from the host operating system so as to support this? In this paper we explore the level of functionality that should be provided to the language designer.

2 Our experiences

Within our research group, our first prototype implementation of a distributed and persistent support environment was Oisin[4]. Its chief characteristic was that it supported a single programming language (a version of Modula-2 extended with object constructs). Object pointers were larger than virtual addresses and were interpreted on each dereference into a local memory address or a (transparent) request to the support system to either fault-in the target object, or perform a remote invocation[8]. We also implemented distributed parallel computations. We made limited attempts to retrofit other languages above the same support environment, including Budd's Little Smalltalk[3] but it was clear that our approach was insufficiently general. We also made considerable efforts to optimise I/O times, including use of clustering and a tailored I/O disk subsystem[16].

Given this experience, we have designed a new system - Amadeus - so as to meet the goals outlined in section 1 above in supporting multiple languages and object formats. The Amadeus generic runtime functions (e.g. object fault resolution, migration, garbage collection, dynamic linking, etc) are common for different languages and object models.

2.1 Units of distribution and storage

In principle the unit of distribution could range from a byte (or page) as in DSM systems such as Ivy [9], to an object as in Amber [2], to a cluster as in Oisin [4] or an entire address space [10]. In general we are uncomfortable with current DSM implementations, because they do not seem to scale well to a multi-user heterogeneous environment. Nevertheless we do acknowledge that DSM can provide full distribution transparency and not impose restrictions on programming languages. It seems clear that a generic system must support not the finest granularity possible, but instead provide a range of mechanisms suitable for different ganularities, appropriate to different environments.

In language based object systems, objects can vary widely in their size. Clearly this influences the mechanisms required for object storage. Since objects can be small, it is usually important to group together a set of objects which are in some way mutually related. Such groups can then form units of i/o to and from storage, and possibly also of distribution and migration.

One trivial strategy is not to group objects at all. An alternative is to let the runtime system apply some reasonable criterion for grouping objects[15]. While this relieves the programmer of the task, it would seem reasonable to allow semantic information available within a compiler, or provided by the programmer, to be used to improve the effectiveness of the clustering [1].

Thus distribution in our system is done at the object level but it is the responsibility of the language level to identify suitable objects for distribution. Such object are known as *global* objects. For increased efficiency of i/o, global objects can be grouped into clusters[1]. Whenever an object is mapped into memory, or unmapped from memory, the whole cluster to which the object belongs is mapped/unmapped. Clusters need not be visible at the language level. Moreover, objects can be independently migrated between nodes as a result of load balancing.

2.2 Object naming

One possible approach to object naming is to insist that the language adopt *the* system wide object naming mechanism [5, 4]. This approach clearly has disadvantages, of which the most important is likely to be the performance penalty incurred in mapping a global name to the objects address, in the common case of colocated objects. Moreover, this approach will undoubtly require extensive compiler modifications [14], or unnatural use of library pointers (e.g. the "permPtrs" of [6][2]). The alternative is to allow the language to continue to use its original naming scheme between colocated objects and to provide support for translation to the system wide format at appropriate boundaries as necessary[1]. This approach incurs extra cost on object mapping and unmapping and on remote access and requires that the language also provide the necessary information to the system to locate and translate names to the language specific format [13]. This in fact has been the basis of our approach to supporting a range of naming formats above a generic layer. There is also work in progress on avoiding the mandatory pointer swizzling during mapping/unmapping of objects in order to accelerate mapping and unmapping for database type applications [14].

In the current system, it is also possible to bind a Unix file name to a global object name. This makes cooperation between different applications more convenient. However, global objects' names saved in files must be taken into account during garbage collection[3].

2.3 Object faults and resolution

In distributed and persistent systems, detecting and resolving faults due to attempted accesses to remote objects, or to objects which are currently not mapped from storage, are critical. Since these mechanisms relate to extensions to the usual non-distributed, single address space model adopted by most programming languages, at first sight it might appear that the mechanisms should belong to the support environment. Paradoxically, the object fault detection must actually be tied into the language system. Thus the granularity of remote and persistent objects must be guided at the language level, and not soley by the support system.

The choice of which mechanism for fault detection and resolution to use depends on (at least) the expected granularity of objects, the level of encapsulation afforded by the language, the language's naming scheme and support afforded by the hosting kernel and/or hardware. One approach is not to make any language extensions, and to rely on explicit programmer code to test for, load and save objects. In a language such as C++, this also requires the programmer to identify the representation of each persistent object[7]. Inserting runtime locality tests into the dereferencing code, as in [2], is another possibility. It has the obvious overhead when accessing local objects, as well as the requirement for compiler support. Yet a further possibility is the use of local proxies for absent objects which can intercept remote invocation requests and forward them as necessary. With this approach there is no loss of local performance – however compiler support is required to generate proxies. Moreover this approach is not appropriate for direct access to data. Finally, even if DSM is not used, object faults can be detected as accesses to invalid memory [9]. This approach incurs no runtime overhead in accessing local objects, but is heavily dependent on the support provided by the underlying host system (e.g. external pagers). It is also difficult to make the mecha-

---

[1] In the current implementation a cluster is mapped to a Unix file.

[2] We recognise that the "smart" pointers of C++ can address the issue, but unfortunately this is a language specific solution.

[3] The current implementation uses a conservative mark and sweep algorithm.

nism independent of any particular programming language or compiler.

Currently in Amadeus a proxy mechanism is used for object fault detection. Language pre-processors support automatic generation of proxy objects. At the moment, proxies are basically RPC stubs, and direct access to global object's data is not supported. We are investigating other ways of using the proxy mechanism however making proxies visible at the language level results in loss of distribution transparency. Fault resolution may be performed either by remote function call or by fetching the object.

2.4 Associative access

Associative access is important in a general persistent storage scheme for a distributed multi-user environment, but creates a number of challenges.

Simple associative collections are relatively simple to build given a persistent language system. However efficient retrieval and scanning of large collections of objects is difficult, even given a persistent store. An example issue is how to manage updates to objects for which keyed indexes exist. Keyed indexes can, in general, be created (and deleted) dynamically and are not usually fixed at the time the collection is itself created. Further, changing the state of an object within a collection may require a change to one or more of the collection indexes. Individual objects (of the same class) may or may not persist. Of those which persist, they need not necessarily all end up in the same persistent storage subsystem. Thus, given a pointer to an object, it may not be apparent that that object is also registered in a collection which has keyed indexes.

One proven strategy is to require the programmer to explicitly indicate that an object has been changed, and that therefore the system should update any related indexes. This in fact is the usual mode of operation of a classical DBMS. However we suggest that this is overly restrictive, given the kind of persistent environment outlined above. One solution to this problem is to dynamically alter the behaviour of an object by binding it to a different version of its class code. The class code can then issue notification changes to the keyed indexes as required. An alternative might be to "wrap" the object, so that use of it is moderated by an intermediary which can issue notifications when necessary. We are actively exploring both of these approaches.

2.5 Transactions

A final issue of importance in the assumed environment is transaction management. The mechanisms must be available to a range of languages however they must also be flexible allowing application specific knowledge and requirements to be exploited. The system must support mechanisms to detect accesses to atomic objects. Such a mechanism is similar to that for detecting objects faults, however it is constrained by the fact that the object may be present in the address space and may be concurrenty accessed by other activities. This latter fact effecively rules out techniques based on virtual memory trapping, so that only explicitly coded checks and methods based on intercepting operations are viable. In our assumed envirorment the fact that the same code may be used to access both atomic and non-atomic objects means that we favour intercepting attempted access to atomic objects in order to enforce concurrency control and recovery policies.

3 Status

Our current implementation supports application programming in C++; a specific aim is that existing (unaltered) C++ code continues to run in our environment. Persistence and distribution are provided by a small number of extensions to the language, which are interpreted by suitable preprocessing. Currently we e are actively working on support for Eiffel.

Our current implementation is above Unix. We are also implemening above the Mach 3.0 and CHORUS microkernels. We are prototyping interfaces to both an OODBMS and a relational DBMS and are exploring how associative techniques on large object collections can be applied. The design of the internal interfaces for integrating the Relax transactions layer has been done[12]; work on porting it to our prototype system is starting. We are naturally keen to gain further experience with the environment, and in that regard have made it available to a number of sites for evaluation.

References

[1] E. Jul et al. Fine grained mobility in the emerald system. Technical Report 87-02-03, Department of Computer Science, University of Washington, February 1987.

[2] J.S. Chase et al. The amber system: Parallel programming on a network of multiprocessors. Technical Report 89-04-01, Department of Computer Science, University of Washington, April 1989.

[3] T. Budd. *A Little Smalltalk*. Addison-Wesley, 1987.

[4] V. Cahill. OISIN, the design of a distributed object-oriented kernel for COMANDOS. Master's thesis, Department of Computer Science, Trinity College Dublin, 1988.

[5] D. Decouchant et al. Guide: An Implementation of the COMANDOS Object Oriented Architecture. In *Proceedings of the EUUG Autumn Conference*, October 1988.

[6] SOR group. Programmers manual for sos prototype - version 4. Technical Report 103, INRIA, December 1988.

[7] K.E. Gorlen et al. *Data Abstraction and Object-Oriented Programming in C++*. John Wiley and Sons.

[8] A. Kramer. The design and implementation of the Oisin Runtime support. Master's thesis, Department of Computer Science, Trinity College Dublin, September 1989.

[9] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[10] B. Liskov and R. Scheifler. Guardians and actions: linguistic support for robust distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[11] M. Weiser et al. The portable common runtime approach to interoperability. In 12^{th} *Symposium on Operating Systems Principles*, pages 114–122. ACM, December 1989.

[12] M. Mock and R. Kroeger. Implementing atomic objects with the RelaX transaction layer. In *Submitted to International Workshop on Object-Orientation in Operating Systems*, 1991.

[13] M.P. Atkinson et al. An approach to persistent programming. *Computer Journal*, 26(4):360–365, 1983.

[14] J.E. Richardson and M.J. Carey. Persistence in the e language: Issues and implementation. *Software Practice and Experience*, 19(12), December 1989.

[15] J.W. Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Transactions on Computer Systems*, 2(2):155–180, May 1984.

[16] G. Starovic. The design and implementation of an object-oriented i/o and storage system for a distributed kernel. Master's thesis, Department of Computer Science, Trinity College Dublin, September 1989.