A Note on Distributed Computing

Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall

Sun Microsystems Laboratories
2550 Garcia Avenue
Mountain View, CA 94043

1 Introduction

Much of the current work in distributed, object-oriented systems is based on the assumption that objects form a single ontological class. This class consists of all entities that can be fully described by the specification of the set of interfaces supported by the object and the semantics of the operations in those interfaces. The class includes objects that share a single address space, objects that are in separate address spaces on the same machine, and objects that are in separate address spaces on different machines (with, perhaps, different architectures). On the view that all objects are essentially the same kind of entity, these differences in relative location are merely an aspect of the implementation of the object. Indeed, the location of an object may change over time, as an object migrates from one machine to another or the implementation of the object changes.

It is the thesis of this note that this unified view of objects is mistaken. There are fundamental differences between the interactions of distributed objects and the interactions of non-distributed objects. Further, work in distributed object-oriented systems that is based on a model that ignores or denies these differences is doomed to failure, and could easily lead to an industry-wide rejection of the notion of distributed object-based systems.

1.1 Terminology

In what follows, we will talk about local and distributed computing. By *local computing* (local object invocation, etc.), we mean programs that are confined to a single address space. In contrast, we will use the term *distributed computing* (remote object invocation, etc.) to refer to programs that make calls to other address spaces, possibly on another machine. In the case of distributed computing, nothing is known about the recipient of the call (other than that it supports a particular interface). For example, the client of such a distributed object does not know the hardware architecture on which the recipient of the call is running, or the language in which the recipient was implemented.

Given the above characterizations of “local” and “distributed” computing, the categories are not exhaustive. There is a middle ground, in which calls are made from one address space to another but in which some characteristics of the called object are known. An important class of this sort consists of calls from one address space to another on the same machine; we will discuss these later in the paper.

2 The Vision of Unified Objects

There is an overall vision of distributed object-oriented computing in which, from the programmer's point of view, there is no essential distinction between objects that share an address space and objects that are on two machines with different architectures located on different continents. While this view can most recently be seen in such works as the Object Management Group's Common Object Request Broker Architecture (CORBA) [1], it has a history that includes such research systems as Arjuna [2], Emerald [3], and Clouds [4].

In such systems, an object, whether local or remote, is defined in terms of a set of interfaces declared in an interface definition language. The implementation of the object is independent of the interface and hidden from other objects. While the underlying mechanisms used to make a method call may differ depending on the location of the object, those mechanisms are hidden from the programmer who writes exactly the same code for either type of call, and the system takes care of delivery.

This vision can be seen as an extension of the goal of remote procedure call (RPC) systems to the object-oriented paradigm. RPC systems attempt to make cross-address space function calls look (to the client programmer) like local function calls. Extending this to the object-oriented programming paradigm allows papering over not just the marshalling of parameters and the unmarshalling of results (as is done in RPC systems) but also the locating and connecting to the target objects. Given the isolation of an object's implementation from clients of the object, the use of objects for distributed computing seems natural. Whether a given object invocation is local or remote is a function of the implementation of the objects being used, and could possibly change from one method invocation to another on any given object.

Implicit in this vision is that the system will be "objects all the way down"; that is, that all current invocations or calls for system services will be eventually converted into calls that might be to an object residing on some other machine. There is a single paradigm of object use and communication used no matter what the location of the object might be.

In actual practice, of course, a local member function call and a cross-continent object invocation are not the same thing. The vision is that developers write their applications so that the objects within the application are joined using the same programmatic glue as objects between applications, but it does not require that the two kinds of glue be implemented the same way. What is needed is a variety of implementation techniques, ranging from same-address-space implementations like Microsoft's Object Linking and Embedding [5] to typical network RPC; different needs for speed, security, reliability, and object co-location can be met by using the right "glue" implementation.

Writing a distributed application in this model proceeds in three phases. The first phase is to write the application without worrying about where objects are located and how their communication is implemented. The developer will simply strive for the natural and correct interface between objects. The system will choose reasonable defaults for object location, and depending on how performance-critical the application is, it may be possible to alpha test it with no further work. Such an approach will enforce a desirable separation between the abstract architecture of the application and any needed performance tuning.

The second phase is to tune performance by "concretizing" object locations and communication methods. At this stage, it may be necessary to use as yet unavailable tools to allow analysis of the communication patterns between objects, but it is certainly conceivable that such tools could be produced. Also during the second phase, the right set of interfaces to export to various clients—such as other applications—can be chosen. There is obviously tremendous flexibility here for the application developer. This seems to be the sort of development scenario that is being advocated in systems like Fresco [6], which claim that the decision to make an object local or remote can be put off until after initial system implementation.

The final phase is to test with "real bullets" (e.g., networks being partitioned, machines going down). Interfaces between carefully selected objects can be beefed up as necessary to deal with these sorts of partial failures introduced by distribution by adding replication, transactions, or whatever else is needed. The exact set of these services can be determined only by experience that will be gained during the development of the system and the first applications that will work on the system.

A central part of the vision is that if an application is built using objects all the way down, in a proper object-oriented fashion, the right “fault points” at which to insert process or machine boundaries will emerge naturally. But if you initially make the wrong choices, they are very easy to change.

One conceptual justification for this vision is that whether a call is local or remote has no impact on the correctness of a program. If an object supports a particular interface, and the support of that interface is semantically correct, it makes no difference to the correctness of the program whether the operation is carried out within the same address space, on some other machine, or off-line by some other piece of equipment. Indeed, seeing location as a part of the implementation of an object and therefore as part of the state that an object hides from the outside world appears to be a natural extension of the object-oriented paradigm.

Such a system would enjoy many advantages. It would allow the task of software maintenance to be changed in a fundamental way. The granularity of change, and therefore of upgrade, could be changed from the level of the entire system (the current model) to the level of the individual object. As long as the interfaces between objects remain constant, the implementations of those objects can be altered at will. Remote services can be moved into an address space, and objects that share an address space can be split and moved to different machines, as local requirements and needs dictate. An object can be repaired and the repair installed without worry that the change will impact the other objects that make up the system. Indeed, this model appears to be the best way to get away from the “Big Wad of Software” model that currently is causing so much trouble.

This vision is centered around the following principles that may, at first, appear plausible:

- there is a single natural object-oriented design for a given application, regardless of the context in which that application will be deployed;
- failure and performance issues are tied to the implementation of the components of an application, and consideration of these issues should be left out of an initial design; and
- the interface of an object is independent of the context in which that object is used.

Unfortunately, all of these principles are false. In what follows, we will show why these principles are mistaken, and why it is important to recognize the fundamental differences between distributed computing and local computing.

3 Déjà Vu All Over Again

For those of us either old enough to have experienced it or interested enough in the history of computing to have learned about it, the vision of unified objects is quite familiar. The desire to merge the programming and computational models of local and remote computing is not new.

Communications protocol development has tended to follow two paths. One path has emphasized integration with the current language model. The other path has emphasized solving the problems inherent in distributed computing. Both are necessary, and successful advances in distributed computing synthesize elements from both camps.

Historically, the language approach has been the less influential of the two camps. Every ten years (approximately), members of the language camp notice that the number of distributed applications is relatively small. They look at the programming interfaces and decide that the problem is that the programming model is not close enough to whatever programming model is currently in vogue (messages in the 1970s [7], [8], procedure calls in the 1980s [9], [10], [11], and objects in the 1990s [1], [2]). A furious bout of language and protocol design takes place and a new distributed computing paradigm is announced that is compliant with the latest programming model. After several years, the percentage of distributed applications is discovered not to have increased significantly, and the cycle begins anew.

A possible explanation for this cycle is that each round is an evolutionary stage for both the local and the distributed programming paradigm. The repetition of the pattern is a result of neither model being sufficient to encompass both activities at any previous stage. However, (this explanation continues) each iteration has brought us closer to a unification of the local and distributed computing models. The current iteration, based on the object-oriented approach to both local and distributed programming, will be the one that produces a single computational model that will suffice for both.

A less optimistic explanation of the failure of each attempt at unification holds that any such attempt will fail for the simple reason that programming distributed applications is not the same as programming non-distributed applications. Just making the communications paradigm the same as the language paradigm is insufficient to make programming distributed programs easier, because communicating between the parts of a distributed application is not the difficult part of that application.

The hard problems in distributed computing are not the problems of how to get things on and off the wire. The hard problems in distributed computing concern dealing with partial failure and the lack of a central resource manager. The hard problems in distributed computing concern insuring adequate performance and dealing with problems of concurrency. The hard problems have to do with differences in memory access paradigms between local and distributed entities. People attempting to write distributed applications quickly discover that they are spending all of their efforts in these areas and not on the communications protocol programming interface.

This is not to argue against pleasant programming interfaces. However, the law of diminishing returns comes into play rather quickly. Even with a perfect programming model of complete transparency between “fine-grained” language-level objects and “larger-grained” distributed objects, the number of distributed applications would not be noticeably larger if these other problems have not been addressed.

All of this suggests that there is interesting and profitable work to be done in distributed computing, but it needs to be done at a much higher-level than that of “fine-grained” object integration. Providing developers with tools that help manage the complexity of handling the problems of distributed application development as opposed to the generic application development is an area that has been poorly addressed.

4 Local and Distributed Computing

The major differences between local and distributed computing concern the areas of latency, memory access, partial failure, and concurrency.[1] The difference in latency is the most obvious, but in many ways is the least fundamental. The often overlooked differences concerning memory access, partial failure, and concurrency are far more difficult to explain away, and the differences concerning partial failure and concurrency make unifying the local and remote computing models impossible without making unacceptable compromises.

4.1 Latency

The most obvious difference between a local object invocation and the invocation of an operation on a remote (or possibly remote) object has to do with the latency of the two calls. The difference between the two is currently between four and five orders of magnitude, and given the relative rates at which processor speed and network latency speeds are changing, the difference in the future promises to be at best no better, and will likely be worse. It is this disparity in efficiency that is often seen as the essential difference between local and distributed computing.

Ignoring the difference between the performance of local and remote invocations can lead to designs whose implementations are virtually assured of having performance problems because the design requires a large amount of communication between components that are in different address spaces and on different machines. Ignoring the difference in the time it takes to make a remote object invocation and the time it takes to make a local object invocation is to ignore one of the major design areas of an application. A properly designed application will require determining, by understanding the application being designed, what objects can be made remote and what objects must be clustered together.

The vision outlined earlier, however, has an answer to this objection. The answer is two-pronged. The first prong is to rely on the steadily increasing speed of the underlying hardware to make the difference in latency irrelevant. This, it is often argued, is what has happened to efficiency concerns having to do with everything from high level languages to virtual memory. Designing at the cutting edge has always required that the hardware catch up before the design is efficient enough for the real world. Arguments from efficiency seem to have gone out of style in software

1. We are not the first to notice these differences; indeed, they are clearly stated in [12].