A Note on Distributed Computing

Jim Waldo
Geoff Wyant
Ann Wollrath
Sam Kendall

SMLI TR-94-29 November 1994

Abstract:

We argue that objects that interact in a distributed system need to be dealt with in ways that are intrinsically different from objects that interact in a single address space. These differences are required because distributed systems require that the programmer be aware of latency, have a different model of memory access, and take into account issues of concurrency and partial failure.

We look at a number of distributed systems that have attempted to paper over the distinction between local and remote objects, and show that such systems fail to support basic requirements of robustness and reliability. These failures have been masked in the past by the small size of the distributed systems that have been built. In the enterprise-wide distributed systems foreseen in the near future, however, such a masking will be impossible.

We conclude by discussing what is required of both systems-level and application-level programmers and designers if one is to take distribution seriously.

Sun Microsystems Laboratories, Inc.
A Sun Microsystems, Inc. Business
M/S 29-01
2550 Garcia Avenue
Mountain View, CA 94043

email addresses:
jim.waldo@east.sun.com
geoff.wyant@east.sun.com
ann.wollrath@east.sun.com
sam.kendall@east.sun.com

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].

engineering, since in the past such concerns have always been answered by speed increases in the underlying hardware.

The second prong of the reply is to admit to the need for tools that will allow one to see what the pattern of communication is between the objects that make up an application. Once such tools are available, it will be a matter of tuning to bring objects that are in constant contact to the same address space, while moving those that are in relatively infrequent contact to wherever is most convenient. Since the vision allows all objects to communicate using the same underlying mechanism, such tuning will be possible by simply altering the implementation details (such as object location) of the relevant objects. However, it is important to get the application correct first, and after that one can worry about efficiency.

Whether or not it will ever become possible to mask the efficiency difference between a local object invocation and a distributed object invocation is not answerable *a priori*. Fully masking the distinction would require not only advances in the technology underlying remote object invocation, but would also require changes to the general programming model used by developers.

If the only difference between local and distributed object invocations was the difference in the amount of time it took to make the call, one could strive for a future in which the two kinds of calls would be conceptually indistinguishable. Whether the technology of distributed computing has moved far enough along to allow one to plan products based on such technology would be a matter of judgement, and rational people could disagree as to the wisdom of such an approach.

However, the difference in latency between the two kinds of calls is only the most obvious difference. Indeed, this difference is not really the fundamental difference between the two kinds of calls, and that even if it were possible to develop the technology of distributed calls to an extent that the difference in latency between the two sorts of calls was minimal, it would be unwise to construct a programming paradigm that treated the two calls as essentially similar. In fact, the difference in latency between local and remote calls, because it is so obvious, has been the only difference most see between the two, and has tended to mask the more irreconcilable differences.

4.2 Memory access

A more fundamental (but still obvious) difference between local and remote computing concerns the access to memory in the two cases—specifically in the use of pointers. Simply put, pointers in a local address space are not valid in another (remote) address space. The system can paper over this difference, but for such an approach to be successful, the transparency must be complete. Two choices exist: either all memory access must be controlled by the underlying system, or the programmer must be aware of the different types of access—local and remote. There is no inbetween.

If the desire is to completely unify the programming model—to make remote accesses behave as if they were in fact local—the underlying mechanism must totally control all memory access. Providing distributed shared memory is one way of completely relieving the programmer from worrying about remote memory access (or the difference between local and remote). Using the object-oriented paradigm to the fullest, and requiring the programmer to build an application with "objects all the way down," (that is, only object references or values are passed as method arguments) is another way to eliminate the boundary between local and remote computing. The layer underneath can exploit this approach by marshalling and unmarshalling method arguments and return values for intra-address space transmission.

But adding a layer that allows the replacement of all pointers to objects with object references only *permits* the developer to adopt a unified model of object interaction. Such a unified model cannot be *enforced* unless one also removes the ability to get address-space-relative pointers from the language used by the developer. Such an approach erects a barrier to programmers who want to start writing distributed applications, in that it requires that those programmers learn a new style of programming which does not use address-space-relative pointers. In requiring that programmers learn such a language, moreover, one gives up the complete transparency between local and distributed computing.

Even if one were to provide a language that did not allow obtaining address-space-relative pointers to objects (or returned an object reference whenever such a pointer was requested), one would need to provide an equivalent way of making cross-address space reference to entities other than objects. Most programmers use pointers as references for many different kinds of entities. These pointers must either be replaced with something that can be used in cross-address space calls or the programmer will need to be aware of the difference between such calls (which will either not allow pointers to such entities, or do something special with those pointers) and local calls. Again, while this could be done, it does violate the doctrine of complete unity between local and remote calls. Because of memory access constraints, the two *have* to differ.

The danger lies in promoting the myth that "remote access and local access are exactly the same" and not enforcing the myth. An underlying mechanism that does not unify all memory accesses while still promoting this myth is both misleading and prone to error. Programmers buying into the myth may believe that they do not have to change the way they think about programming. The programmer is therefore quite likely to make the mistake of using a pointer in the wrong context, producing incorrect results. "Remote is just like local," such programmers think, "so we have just one unified programming model." Seemingly, programmers need not change their style of programming. In an incomplete implementation of the underlying mechanism, or one that allows an implementation language that in turn allows direct access to local memory, the system does not take care of all memory accesses, and errors are bound to occur. These errors occur because the programmer is not aware of the difference between local and remote access and what is actually happening "under the covers."

The alternative is to explain the difference between local and remote access, making the programmer aware that remote address space access is very different from local access. Even if some of the pain is taken away by using an interface definition language like that specified in [1] and having it generate an intelligent language mapping for operation invocation on distributed objects, the programmer aware of the difference will not make the mistake of using pointers for cross-address space access. The programmer will know it is incorrect. By not masking the difference, the programmer is able to learn when to use one method of access and when to use the other.

Just as with latency, it is logically possible that the difference between local and remote memory access could be completely papered over and a single model of both presented to the programmer. When we turn to the problems introduced to distributed computing by partial failure and concurrency, however, it is not clear that such a unification is even conceptually possible.

4.3 Partial failure and concurrency

While unlikely, it is at least logically possible that the differences in latency and memory access between local computing and distributed computing could be masked. It is not clear that such a masking could be done in such a way that the local computing paradigm could be used to produce distributed applications, but it might still be possible to allow some new programming technique to be used for both activities. Such a masking does not even seem to be logically possible, however, in the case of partial failure and concurrency. These aspects appear to be different in kind in the case of distributed and local computing.[2]

Partial failure is a central reality of distributed computing. Both the local and the distributed world contain components that are subject to periodic failure. In the case of local computing, such failures are either total, affecting all of the entities that are working together in an application, or detectable by some central resource allocator (such as the operating system on the local machine).

This is not the case in distributed computing, where one component (machine, network link) can fail while the others continue. Not only is the failure of the distributed components independent, but there is no common agent that is able to determine what component has failed and inform the other components of that failure, no global state that can be examined that allows determination of exactly what error has occurred. In a distributed system, the failure of a network link is indistinguishable from the failure of a processor on the other side of that link.

These sorts of failures are not the same as mere exception raising or the inability to complete a task, which can occur in the case of local computing. This type of failure is caused when a machine crashes during the execution of an object invocation or a network link goes down, occurrences that cause the target object to simply disappear rather than return control to the caller. A central problem in distributed computing is insuring that the state of the

2. In fact, authors such as Schroeder [12] and Hadzilacos and Toueg [13] take partial failure and concurrency to be the defining problems of distributed computing.

whole system is consistent after such a failure; this is a problem that simply does not occur in local computing.

The reality of partial failure has a profound effect on how one designs interfaces and on the semantics of the operations in an interface. Partial failure requires that programs deal with indeterminacy. When a local component fails, it is possible to know the state of the system that caused the failure and the state of the system after the failure. No such determination can be made in the case of a distributed system. Instead, the interfaces that are used for the communication must be designed in such a way that it is possible for the objects to react in a consistent way to possible partial failures.

Being robust in the face of partial failure requires some expression at the interface level. Merely improving the implementation of one component is not sufficient. The interfaces that connect the components must be able to state whenever possible the cause of failure, and there must be interfaces that allow reconstruction of a reasonable state when failure occurs and the cause cannot be determined.

If an object is coresident in an address space with its caller, partial failure is not possible. A function may not complete normally, but it always completes. There is no indeterminism about how much of the computation completed. Partial completion can occur only as a result of circumstances that will cause the other components to fail.

The addition of partial failure as a possibility in the case of distributed computing does not mean that a single object model cannot be used for both distributed computing and local computing. The question is not "can you make remote method invocation look like local method invocation?" but rather "what is the price of making remote method invocation identical to local method invocation?" One of two paths must be chosen if one is going to have a unified model.

The first path is to treat all objects as if they were local and design all interfaces as if the objects calling them, and being called by them, were local. The result of choosing this path is that the resulting model, when used to produce distributed systems, is essentially indeterministic in the face of partial failure and consequently fragile and non-robust. This path essentially requires ignoring the extra failure modes of distributed computing. Since one can't get rid of those failures, the price of adopting the model is to require that such failures are unhandled and catastrophic.

The other path is to design all interfaces as if they were remote. That is, the semantics and operations are all designed to be deterministic in the face of failure, both total and partial. However, this introduces unnecessary guarantees and semantics for objects that are never intended to be used remotely. Like the approach to memory access that attempts to require that all access is through system-defined references instead of pointers, this approach must also either rely on the discipline of the programmers using the system or change the implementation language so that all of the forms of distributed indeterminacy are forced to be dealt with on all object invocations.

This approach would also defeat the overall purpose of unifying the object models. The real reason for attempting such a unification is to make distributed computing more like local computing and thus make distributed computing easier. This second approach to unifying the models makes local computing as complex as distributed computing. Rather than encouraging the production of distributed applications, such a model will discourage its own adoption by making all object-based computing more difficult.

Similar arguments hold for concurrency. Distributed objects by their nature must handle concurrent method invocations. The same dichotomy applies if one insists on a unified programming model. Either all objects must bear the weight of concurrency semantics, or all objects must ignore the problem and hope for the best when distributed. Again, this is an interface issue and not solely an implementation issue, since dealing with concurrency can take place only by passing information from one object to another through the agency of the interface. So either the overall programming model must ignore significant modes of failure, resulting in a fragile system; or the overall programming model must assume a worst-case complexity model for all objects within a program, making the production of any program, distributed or not, more difficult.

One might argue that a multi-threaded application needs to deal with these same issues. However, there is a subtle difference. In a multi-threaded application, there is no real source of indeterminacy of invocations of operations. The application programmer has complete control over invocation order when desired. A distributed system by its nature introduces truly asynchronous operation invocations. Further, a non-distributed system, even when multi-threaded, is layered on top of a single operating system that can aid the communication between objects and can be used to determine and aid in synchronization and in the recovery of failure. A distributed system, on the other hand, has no single point of resource allocation, synchronization, or failure recovery, and thus is conceptually very different.

5 The Myth of "Quality of Service"

One could take the position that the way an object deals with latency, memory access, partial failure, and concurrency control is really an aspect of the implementation of that object, and is best described as part of the "quality of service" provided by that implementation. Different implementations of an interface may provide different levels of reliability, scalability, or performance. If one wants to build a more reliable system, one merely needs to choose more reliable implementations of the interfaces making up the system.

On the surface, this seems quite reasonable. If I want a more robust system, I go to my catalog of component vendors. I quiz them about their test methods. I see if they have ISO9000 certification, and I buy my components from the one I trust the most. The components all comply with the defined interfaces, so I can plug them right in; my system is robust and reliable, and I'm happy.

Let us imagine that I build an application that uses the (mythical) queue interface to enqueue work for some component. My application dutifully enqueues records that represent work to be done. Another application dutifully dequeues them and performs the work. After a while, I notice that my application crashes due to time-outs. I find this extremely annoying, but realize that it's my fault. My application just isn't robust enough. It gives up too easily on a time-out. So I change my application to retry the operation until it succeeds. Now I'm happy. I almost never see a time-out. Unfortunately, I now have another problem. Some of the requests seem to get processed two, three, four, or more times. How can this be? The component I bought which implements the queue has allegedly been rigorously tested. It shouldn't be doing this. I'm angry. I call the vendor and yell at him. After much finger-pointing and research, the culprit is found. The problem turns out to be the way I'm using the queue. Because of my handling of partial failures (which in my naiveté, I had thought to be total), I have been enqueuing work requests multiple times.

Well, I yell at the vendor that it is still their fault. Their queue should be detecting the duplicate entry and removing it. I'm not going to continue using this software unless this is fixed. But, since the entities being enqueued are just values, there is no way to do duplicate elimination. The only way to fix this is to change the protocol to add request IDs. But since this is a standardized interface, there is no way to do this.

The moral of this tale is that robustness is not simply a function of the implementations of the interfaces that make up the system. While robustness of the individual components has some effect on the robustness of the overall systems, it is not the sole factor determining system robustness. Many aspects of robustness can be reflected only at the protocol/interface level.

Similar situations can be found throughout the standard set of interfaces. Suppose I want to reliably remove a name from a context. I would be tempted to write code that looks like:

```
while (true) {
   try {
       context->remove(name);
       break;
   }
   catch (NotFoundInContext) {
       break;
    }
   catch (NetworkServerFaliure) {
       continue;
   }
}
```

That is, I keep trying the operation until it succeeds (or until I crash). The problem is that my connection to the name server may have gone down, but another client's may have stayed up. I may have, in fact, successfully removed the name but not discovered it because of a network disconnection. The other client then adds the same name, which I then remove. Unless the naming interface includes an operation to lock a naming context, there is no way that I can make this operation completely robust. Again, we see that robustness/reliability needs to be expressed at the interface level. In the design of any operation, the question has to be asked: what happens if the client chooses to repeat this operation with the exact same parameters as previously? What mechanisms are needed to ensure that they get the desired semantics? These are things that can be expressed only at the interface level. These are issues that can't be answered by supplying a "more robust implementation" because the lack of robustness is inherent in the interface and not something that can be changed by altering the implementation.

Similar arguments can be made about performance. Suppose an interface describes an object which maintains sets of other objects. A defining property of sets is that there are no duplicates. Thus, the implementation of this object needs to do duplicate elimination. If the interfaces in the system do not provide a way of testing equality of reference, the objects in the set must be queried to determine equality. Thus, duplicate elimination can be done only by interacting with the objects in the set. It doesn't matter how fast the objects in the set implement the equality operation. The overall performance of eliminating duplicates is going to be governed by the latency in communicating over the slowest communications link involved. There is no change in the set implementations that can overcome this. An interface design issue has put an upper bound on the performance of this operation.

6 Lessons from NFS

We do not need to look far to see the consequences of ignoring the distinction between local and distributed computing at the interface level. NFS®, Sun's distributed computing file system [14], [15] is an example of a non-distributed application programer interface (API) (open, read, write, close, etc.) re-implemented in a distributed way.

Before NFS and other network file systems, an error status returned from one of these calls indicated something rare: a full disk, or a catastrophe such as a disk crash. Most failures simply crashed the application along with the file system. Further, these errors generally reflected a situation that was either catastrophic for the program receiving the error or one that the user running the program could do something about.

NFS opened the door to partial failure within a file system. It has essentially two modes for dealing with an inaccessible file server: soft mounting and hard mounting. But since the designers of NFS were unwilling (for easily understandable reasons) to change the interface to the file system to reflect the new, distributed nature of file access, neither option is particularly robust.

Soft mounts expose network or server failure to the client program. Read and write operations return a failure status much more often than in the single-system case, and programs written with no allowance for these failures can easily corrupt the files used by the program. In the early days of NFS, system administrators tried to tune various parameters (time-out length, number of retries) to avoid these problems. These efforts failed. Today, soft mounts are seldom used, and when they are used, their use is generally restricted to read-only file systems or special applications.

Hard mounts mean that the application hangs until the server comes back up. This generally prevents a client program from seeing partial failure, but it leads to a malady familiar to users of workstation networks: one server crashes, and many workstations—even those apparently having nothing to do with that server—freeze. Figuring out the chain of causality is very difficult, and even when the cause of the failure can be determined, the individual user can rarely do anything about it but wait. This kind of brittleness can be reduced only with strong policies and network administration aimed at reducing interdependencies. Nonetheless, hard mounts are now almost universal.

Note that because the NFS protocol is stateless, it assumes clients contain no state of interest with respect to the protocol; in other words, the server doesn't care what happens to the client. NFS is also a "pure" client-server protocol, which means that failure can be limited to three parties: the client, the server, or the network. This combination of features means that failure modes are simpler than in the more general case of peer-to-peer distributed object-oriented applications where no such limitation on shared state can be made and where servers are themselves clients of other servers. Such peer-to-peer distributed applications can and will fail in far more intricate ways than are currently possible with NFS.

The limitations on the reliability and robustness of NFS have nothing to do with the implementation of the parts of that system. There is no "quality of service" that can be improved to eliminate the need for hard mounting NFS volumes. The problem can be traced to the interface upon which NFS is built, an interface that was designed for non-distributed computing where partial failure was not possible. The reliability of NFS cannot be changed without a change to that interface, a change that will reflect the distributed nature of the application.

This is not to say that NFS has not been successful. In fact, NFS is arguably the most successful distributed application that has been produced. But the limitations on the robustness have set a limitation on the scalability of NFS. Because of the intrinsic unreliability of the NFS protocol, use of NFS is limited to fairly small numbers of machines, geographically co-located and centrally administered. The way NFS has dealt with partial failure has been to informally require a centralized resource manager (a system administrator) who can detect system failure, initiate resource reclamation and insure system consistency. But by introducing this central resource manager, one could argue that NFS is no longer a genuinely distributed application.

7 Taking the Difference Seriously

Differences in latency, memory access, partial failure, and concurrency make merging of the computational models of local and distributed computing both unwise to attempt and unable to succeed. Merging the models by making local computing follow the model of distributed computing would require major changes in implementation languages (or in how those languages are used) and make local computing far more complex than is otherwise necessary. Merging the models by attempting to make distributed computing follow the model of local computing requires ignoring the different failure modes and basic indeterminacy inherent in distributed computing, leading to systems that are unreliable and incapable of scaling beyond small groups of machines that are geographically co-located and centrally administered.

A better approach is to accept that there are irreconcilable differences between local and distributed computing, and to be conscious of those differences at all stages of the design and implementation of distributed applications. Rather than trying to merge local and remote objects, engineers need to be constantly reminded of the differences between the two, and know when it is appropriate to use each kind of object.

Accepting the fundamental difference between local and remote objects does not mean that either sort of object will require its interface to be defined differently. An interface definition language such as IDL can still be used to specify the set of interfaces that define objects. However, an additional part of the definition of a class of objects will be the specification of whether those objects are meant to be used locally or remotely. This decision will need to consider what the anticipated message frequency is for the object, and whether clients of the object can accept the indeterminacy implied by remote access. The decision will be reflected in the interface to the object indirectly, in that the interface for objects that are meant to be accessed remotely will contain operations that allow reliability in the face of partial failure.

It is entirely possible that a given object will often need to be accessed by some objects in ways that cannot allow indeterminacy, and by other objects relatively rarely and in a way that does allow indeterminacy. Such cases should be split into two objects (which might share an implementation) with one having an interface that is best for local access and the other having an interface that is best for remote access.

A compiler for the interface definition language used to specify classes of objects will need to alter its output based on whether the class definition being compiled is for a class to be used locally or a class being used remotely. For interfaces meant for distributed objects, the code produced might be very much like that generated by RPC stub compilers today. Code for a local interface, however, could be much simpler, probably requiring little more than a class definition in the target language.

While writing code, engineers will have to know whether they are sending messages to local or remote objects, and access those objects differently. While this might seem to add to the programming difficulty, it will in fact aid the programmer by providing a framework under which he or she can learn what to expect from the different kinds of calls. To program completely in the local environment, according to this model, will not require any changes from the programmer's point of view. The discipline of defining classes of objects using an interface definition language will insure the desired separation of interface from implementation, but the actual process of implementing an interface will be no different than what is done today in an object-oriented language.

Programming a distributed application will require the use of different techniques than those used for non-distributed applications. Programming a distributed application will require thinking about the problem in a different way than before it was thought about when the solution was a non-distributed application. But that is only to be expected. Distributed objects are different from local objects, and keeping that difference visible will keep the programmer from forgetting the difference and making mistakes. Knowing that an object is outside of the local address space, and perhaps on a different machine, will remind the programmer that he or she needs to program in a way that reflects the kinds of failures, indeterminacy, and concurrency constraints inherent in the use of such objects. Making the difference visible will aid in making the difference part of the design of the system.

Accepting that local and distributed computing are different in an irreconcilable way will also allow an organization to allocate its research and engineering resources more wisely. Rather than using those resources in attempts to paper over the differences between the two kinds of computing, resources can be directed at improving the performance and reliability of each.

One consequence of the view espoused here is that it is a mistake to attempt to construct a system that is "objects all the way down" if one understands the goal as a distributed system constructed of the *same kind* of objects all the way down. There will be a line where the object model changes; on one side of the line will be distributed objects, and on the other side of the line there will (perhaps) be local objects. On either side of the line, entities on the other side of the line will be opaque; thus one distributed object will not know (or care) if the implementation of another distributed object with which it communicates is made up of objects or is implemented in some other way. Objects on different sides of the line will differ in kind and not just in degree; in particular, the objects will differ in the kinds of failure modes with which they must deal.

8 A Middle Ground

As noted in Section 2, the distinction between local and distributed objects as we are using the terms is not exhaustive. In particular, there is a third category of objects made up of those that are in different address spaces but are guaranteed to be on the same machine. These are the sorts of objects, for example, that appear to be the basis of systems such as Spring [16] or Clouds [4]. These objects have some of the characteristics of distributed objects, such as increased latency in comparison to local objects and the need for a different model of memory access. However, these objects also share characteristics of local objects, including sharing underlying resource management and failure modes that are more nearly deterministic.

It is possible to make the programming model for such "local-remote" objects more similar to the programming model for local objects than can be done for the general case of distributed objects. Even though the objects are in different address spaces, they are managed by a single resource manager. Because of this, partial failure and the indeterminacy that it brings can be avoided. The programming model for such objects will still differ from that used for objects in the same address space with respect to latency, but the added latency can be reduced to generally acceptable levels. The programming models will still necessarily differ on methods of memory access and concurrency, but these do not have as great an effect on the construction of interfaces as additional failure modes.

The other reason for treating this class of objects separately from either local objects or generally distributed objects is that a compiler for an interface definition language can be significantly optimized for such cases. Parameter and result passing can be done via shared memory if it is known that the objects communicating are on the same machine. At the very least, marshalling of parameters and the unmarshalling of results can be avoided.

The class of locally distributed objects also forms a group that can lead to significant gains in software modularity. Applications made up of collections of such objects would have the advantage of forced and guaranteed separation between the interface to an object and the implementation of that object, and would allow the replacement of one implementation with another without affecting other parts of the system. Because of this, it might be advantageous to investigate the uses of such a system. However, this activity should not be confused with the unification of local objects with the kinds of distributed objects we have been discussing.

9 References

[1] The Object Management Group. "Common Object Request Broker: Architecture and Specification." *OMG Document Number 91.12.1* (1991).

[2] [Parrington, Graham D. "Reliable Distributed Programming in C++: The Arjuna Approach." *USENIX 1990 C++ Conference Proceedings* (1991).

[3] Black, A., N. Hutchinson, E. Jul, H. Levy, and L. Carter. "Distribution and Abstract Types in Emerald." *IEEE Transactions on Software Engineering* SE-13, no. 1, (January 1987).

[4] Dasgupta, P., R. J. Leblanc, and E. Spafford. "The Clouds Project: Designing and Implementing a Fault Tolerant Distributed Operating System." *Georgia Institute of Technology Technical Report GIT-ICS-85/29*. *(*1985).

[5] Microsoft Corporation. *Object Linking and Embedding Programmers Reference*. version 1. Microsoft Press, 1992.

[6] Linton, Mark. "A Taste of Fresco." Tutorial given at the *8th Annual X Technical Conference (*January 1994).

[7] Jaayeri, M., C. Ghezzi, D. Hoffman, D. Middleton, and M. Smotherman. "CSP/80: A Language for Communicating Sequential Processes." *Proceedings: Distributed Computing CompCon* (Fall 1980).

[8] Cook, Robert. "MOD- A Language for Distributed Processing." *Proceedings of the 1st International Conference on Distributed Computing Systems (*October 1979).

[9] Birrell, A. D. and B. J. Nelson. "Implementing Remote Procedure Calls." *ACM Transactions on Computer Systems* 2 (1978).

[10] Hutchinson, N. C., L. L. Peterson, M. B. Abott, and S. O'Malley. "RPC in the x-Kernel: Evaluating New Design Techniques." *Proceedings of the Twelfth Symposium on Operating Systems Principles* 23, no. 5 (1989).

[11] Zahn, L., T. Dineen, P. Leach, E. Martin, N. Mishkin, J. Pato, and G. Wyant. *Network Computing Architecture*. Prentice Hall, 1990.

[12] Schroeder, Michael D. "A State-of-the-Art Distributed System: Computing with BOB." In *Distributed Systems*, 2nd ed., S. Mullender, ed., ACM Press, 1993.

[13] Hadzilacos, Vassos and Sam Toueg. "Fault-Tolerant Broadcasts and Related Problems." In *Distributed Systems*, 2nd ed., S. Mullendar, ed., ACM Press, 1993.

[14] Walsh, D., B. Lyon, G. Sager, J. M. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, and P. Weiss. "Overview of the SUN Network File System." *Proceedings of the Winter Usenix Conference* (1985).

[15] Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. "Design and Implementation of the SUN Network File System." *Proceedings of the Summer Usenix Conference (*1985).

[16] Khalidi, Yousef A. and Michael N. Nelson. "An Implementation of UNIX on an Object-Oriented Operating System." *Proceedings of the Winter USENIX Conference* (1993). Also *Sun Microsystems Laboratories, Inc. Technical Report SMLI TR-92-3* (December 1992).

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@eng.sun.com>.
For distribution issues, contact Amy Tashbook, Assistant Editor <amy.tashbook@eng.sun.com>.