Another Note on Distributed Computing

July 17th, 2008

A Note on Distributing Computing is among the most widely quoted papers on distributed computing. While I agree with much of what Jim Waldo *et al.* wrote, there is quite a bit I find myself disagreeing with, so here is "Another Note on Distributed Computing", to iron out a few misconceptions.

What is right

Here is a quote from the paper:

> *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.*

I could not agree more: I've been preaching for years that, if you go and design the APIs for a distributed application the same way as for a non-distributed one, you are likely to fall flat on your face. Waldo *et al.* cite a number of reasons for this, among them:

Latency issues cannot be ignored.

Yes, that is absolutely correct. Little surprise when one considers that a remote invocation is around four orders of magnitude (that's 10,000 times) slower than a local invocation.

It is impossible to provide uniform memory access for both local and remote objects.

Correct. Even if we were to provide a programming model that allows completely transparent access to local and remote memory at the programming language level, the different error semantics of local and distributed access would create non-uniform semantics.

Distributed invocations are subject to partial failure.

Correct, they are. If a server goes down unexpectedly, and the invocations executing in the server at the time of the failure are not completely stateless, the system as a whole will be in an indeterminate state, which makes it harder to recover from a failure on re-start.

Concurrency adds additional failure modes due to indeterminism.

Correct. For example, depending on how operations are invoked by the client and how they are dispatched in the server, it is possible for sequential invocations made by a single client to be processed out of order in the server.

Distributed invocations provide a fundamentally different quality of service.

Correct. It is harder to create robust distributed applications than non-distributed ones. That should not come as a surprise, seeing that there are many more ways for the former to fail.

What is not so right

So far, I have not actually disagreed with anything, so it's time to look a bit deeper…

Waldo *et al.* write that early distributed systems, such as CORBA, Arjuna, Emerald, and Clouds strived to provide a seamless view of distributed objects, such that *"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. 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 authors do not make it clear what they mean by "share an address space", and do not further explain what they mean by "local" and "remote". To talk meaningfully about this vision of unified objects, we need to be clear about what kinds of objects there actually are:

- **Remotable objects.** These are objects that can (but need not) reside in a different address space. If they do reside in a different address space, they can be reached only via inter-process communication, such as by sharing memory or sending messages over the backplane (for same-machine communication) or over a network (for communication with objects on other machines). If a remotable object is in the same address space, it offers the same interface as if it were remote. It just so happens that it can be reached via a more efficient communication mechanism.
- **Local, language-native objects.** These are the objects that come built into the programming language, such as C++ or Java objects. These objects have nothing to do with distribution.

For remotable objects, the implementation of operations is hidden behind their interface and, as far as the caller of an operation is concerned, the same API is used to invoke the operation, regardless of the actual location of the object (local or remote). However, that API is *not* necessarily the same as the API for a language-native object.

The authors go on to say that *"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."* This suggests that the authors, when they talk about local objects, actually mean language-native objects. However, they also say that local objects have an interface declared in an interface definition language, which suggests collocated remotable objects.

Now, while it is true that systems such as CORBA and Ice indeed strive to make distributed computing as frictionless as possible and to make a remote operation as easy to call as a local or language-native one, they do *not* try to paper over the difference between language-native objects and remotable objects. It just so happens that, *if* an object can be called remotely, it can be called the same way whether the object happens to be collocated in the same address space or not.

This does *not* mean that a language-native object can be called the same way as a remotable one, or vice versa. In particular, in systems such as CORBA and Ice, remotable objects have a type that differs from the type of any language-native object, and pointers (or references) to remotable objects *cannot* be used interchangeably with language-native ones. In particular, invocations on remotable objects are made via proxy types (such as Ice's `Prx` types), and these proxy types are *not* type compatible with a language-native pointer or reference.

Similarly, CORBA never suggested that all objects within an application should have an IDL interface. In fact, CORBA makes a very clear distinction between objects that have an IDL interface (and, therefore, can be made remotely accessible), and objects that do not (and, therefore, are part of the implementation, not interface, of an application).

In fact, neither CORBA nor Ice ever attempted to provide a unified vision of objects. Instead, they make it easy to call and implement remotable objects regardless of whether client and server are collocated or not. This is a far cry from saying that local and remote objects are the same, or that they can be treated as if they were the same.

The authors then assert:

> *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.*

What? Where on earth do they take this from? I cannot recall a single instance where anyone with even the least shred of credibility claimed such a thing, even in the dim-distant days of DCOM and CORBA. There is a big difference between making it easy to call a remote operation, and claiming that, because remote operations are easy to call, we can ignore object location when we design an application. If this is how people start out writing their applications, they are guaranteed to fail, and that fact has been well known and well documented for at least 15 years.

> *The second phase is to tune performance by "concretizing" object locations and communication methods.*

No, most definitely not. If I design the interfaces to my application as they say in phase 1, it is highly unlikely that any amount of performance tuning will save the day. True, performance tuning is necessary for distributed applications, just as it is for local ones. But the preceding two phases are tantamount to saying "Write your application any which way you like, with complete disregard of distribution, and you can fix things in phase 2."

To suggest that such an approach could actually work is disingenuous, to say the least and, again, I am not aware of anyone with any reputation whatsoever having made such a claim, not now, and not 15 years ago.

> *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.*

This, in all seriousness, suggests that I can succeed in dealing with partial failures *after* I have designed the application with complete disregard of distribution, and *after* I have carefully tuned its performance, only to then even *start* worrying about partial failure semantics and remedies to them. Clearly, this is utterly ridiculous—no-one in his right mind would do this.

Now, don't get me wrong—I don't for one moment believe that Jim Waldo and his co-authors actually believe these things. In fact, the second half of their paper makes it abundantly clear that they do not.

But why do they talk for the first nine of fourteen pages about something that no-one in his right mind ever believed in the first place, either now, or a long time ago? And why do they say that CORBA pretended that a distributed application could be written like a non-distributed one when, to the best of my knowledge, no-one even half-way competent ever made such a claim? To me, the answer is that they want to set the stage for the conclusion of the paper. In other words, the first part of the paper softens the ground for the second part.

Waldo *et al.* go on to cite NFS as an example of the consequences of ignoring the distinction between local and distributed computing at the interface level. They point out that, in a sense, NFS was doomed because it either provides non-transparent semantics to applications with soft mounts (which causes applications to fail in unexpected ways), or provides transparent semantics with hard mounts (which causes applications to hang in unexpected ways). Neither alternative is palatable because each leads to failures in the distributed case that simply do not happen in the local case.

They also correctly point out that the problem can be traced to the interface level: because NFS retained the original Unix system calls for file I/O, the catch-22 of NFS is inevitable. But, so what? All this shows is that it is a stupid idea to build a distributed application as if it were a non-distributed one.

The authors go on to say that

> *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.*

Yes! That is *exactly* how we should build distributed applications. We cannot forget—*ever*—when we are dealing with distribution and when we are not. That is true regardless of the technology we use for distribution, regardless of the specific APIs, and regardless of the underlying protocol. The differences are due to *distribution itself*, not due to any artifact of design or implementation.

What is wrong

Now we get to the part of the paper where Waldo et al. jump to seriously wrong conclusions. Let me quote a few key passages.

> *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.*

This is a trivial truism, and hardly worth mentioning. *Of course* engineers need to know when it is appropriate to use a remote object. In fact, engineers need to know not only when it is appropriate to use a *remote* object, they also need to know when it is appropriate to use *any* object. That is, engineers must know not only about the side-effects of remote invocations, they must know about the side effects of *all* invocations, whether they are remote or not.

Whenever I call *any* function or method, I must be aware of the potential consequences of doing so:

- I must know the performance characteristics of the function—$O(\log n)$, $O(n)$, $O(n^2)$, or whatever.
- I must know whether the function performs disk I/O, reads user input, or may attempt to acquire a lock.
- I must know what state the function may leave the system in if something goes wrong. Does the function provide the strong or weak exception guarantee? What is the state of its in- and inout-parameters when something goes wrong?

In fact, just about *everything* I need to think about for a remote invocation, I also need to think about for a local invocation:

- Even if everything works perfectly, many local function invocations take a lot longer than many remote invocations. And many local function invocations take just as long as a remote invocation for the same data. (Just think of sorting a large set of values—the performance difference of the local and remote case is negligible.)
- A local invocation can block or take a long time just as much as a remote invocation can. For example, any invocation that does disk I/O can block (potentially indefinitely, as Waldo *et al.* point out themselves). Similarly, any operation that gets input from a user or attempts to commit a transaction can block for extended amounts of time. And, of course, any operation that attempts to acquire locks can block and, depending on the exact operation, can block for an extended period.
- There are *many* local APIs that provide neither the strong nor the weak exception guarantee. For example, most APIs that perform I/O leave the system in an indeterminate state when something goes wrong during a write (unless we use transactions). Similarly, almost all libraries I have ever seen fail in weird ways in the face of memory starvation. If I am lucky, my process will crash and I'll at least know that something went badly wrong. But, quite often, the programmer who wrote the code wrote it with a mind-set of "Memory never runs out." What happens when memory *does* run out is anyone's guess—it is not unusual for a program to survive temporary memory starvation, but to leave partially updated data structures behind that may (or may not) cause the program to fail later.

The point is that Waldo and his colleagues discuss things such as performance and partial failure in great detail when, in fact, that discussion is *largely orthogonal* to distributed computing. That is because the same issues arise in the local case as well. Not as frequently maybe, but they *do* arise and, when they do, all the same issues come up as for a distributed system.

Note that even what looks like a non-distributed system may turn out to be distributed. In fact, any program that reads from a (local) disk and writes data back to that disk is distributed. Not distributed in space, but distributed in time: if a previous incarnation of the program crashed while it was writing to disk, the next incarnation of the program has to make sense of the mess that its predecessor left behind. This is little different from recovering a distributed system after a crash: either way, one side has to make sense of the mess that was left by "the other side".

> *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.*

In other words, the APIs for local and remote objects should be different and local APIs "could be much simpler, probably requiring little more than a class definition in the target language".

This statement is factually incorrect. For one, with a modern platform such as Ice, if a remotable object is collocated, call dispatch to it is essentially as efficient as for a language-native call. (If you know that an object will be called only locally, you can tell the compiler to get rid of unnecessary marshaling and dispatch code.) Second, with Ice, native APIs cannot be much simpler than remote ones. That's because remote invocations are already as simple as native ones, and because implementing an interface already requires little more than a class definition in the target language. (In defense of the authors, at the time they wrote their paper, things were not as elegant as they are with Ice today.)

But here is where Waldo et al. really go off the deep end:

> *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.*

Translation:

If we give fundamentally different APIs to local and remote objects, that will help programmers write better distributed applications.

I am stunned how the authors can possibly arrive at this conclusion, especially in light of what they so lucidly explain in the first part of their paper. The premises in no way support the conclusion; there simply is no logical link between them. The whole argument reads like:

> *All Greeks have beards. Socrates was a Greek. Therefore, income tax increases will stimulate the economy.*

In fact, the authors themselves explain that much of the difficulty of writing distributed systems stems from problems that have *nothing to do with any specific API*. And yet, somehow, an API for remote calls that differs from the API for local calls is going to "aid the programmer" and solve all our distributed computing problems? Hardly.

What gets me is how patronizing (if not insulting) this conclusion is to programmers. Do Waldo *et al.* really believe that programmers who write distributed systems are so naive that they need a different syntax that "constantly reminds" them when they are making a remote call? That is giving distributed programmers a lot less credibility than they deserve. (Not to mention that, as we have seen, programmers must be aware of the consequences of making *any* call—whether local or remote—anyway.)

But there is something else that Waldo *et al.* apparently did not consider. Let us suppose for a moment that I have what they ask for, and that any remote invocation has to be enclosed in a `remote_call` function or macro. (Let's not quibble about the exact syntax—the point is that there is some syntactic reminder that a call is remote.) Now I write something like this:

```
public class Person
{
    public void
    updateAddress(Address a)
    {
        _person.remote_call("updateAddress", a);
    }

    private RemotePerson _person;
    // ...
}
```

The caller of this person object can now write:

```
Person p = new Person();
Address a = new Address();
// ...
p.updateAddress(a);
```

As far as I can see, the authors' suggestion dies right there: as soon as I make any remote invocation inside another function or method, that function or method itself now must be called like a remote function, otherwise the syntactic marker that is supposed to "remind the programmer" is lost. In other words, the "remoteness" of invocations is transitive and very quickly permeates a program at almost all levels. But in turn, that greatly diminishes the already dubious value of a syntactic marker. Instead, it creates constant overexposure to an inconvenient syntax without any benefit.

Conclusion

In the introduction to their paper, the authors say that a *"unified view of objects is mistaken"*, and then proceed to arrive at the recommendation that "engineers need to be constantly reminded of the differences" between local and distributed computing. I do believe that is indeed good to remind engineers of the difference.

And platforms such as Ice do exactly that, but in a way that does not get in the way of programming. For example, if I want to pass a proxy to a remotable person object to a function, I have to declare the function as follows:

```
void doSomethingWithPerson(PersonPrx person);
```

Because the remotable version of a person has a type `PersonPrx`, and that type differs from and is not compatible with any language-native type `Person`, the act of passing a remotable object is made explicit, and there can be no doubt as to what kind of object we are dealing with. That is all the reminder the programmer needs.

As far as the unified view is concerned, it is not mistaken, at least not for remotable objects. Whether a remotable object is collocated or not should not matter at the point of call, and should not matter in the implementation of the object. Keeping the two the same does not provide a unified view, but location transparency. And location transparency is important. For example, moving out-of-process objects in-process is possible only with this transparency. (Anyone who has ever turned a stand-alone Ice server into an Icebox service will know how easy this is, and that it intrinsically relies on location transparency.) Another advantage of location transparency is that programmers do *not* constantly have to deal with different syntax and can put their attention where it is needed, namely on the application semantics.

A unified view *is* mistaken if it attempts to paper over the difference between language-native and remotable objects, or tries to pretend that programmers can treat remote objects the same way as local ones (whether remotable or not). But neither CORBA nor Ice ever tried to do this, and neither system is unified in that sense.

Distributed computing is hard enough as is, and Ice does its best to not make it harder still. But, ultimately, API style has little to do with the real reasons for why distributed computing is hard. What we need to accept, first and foremost, is that—regardless of APIs, technologies, whether interactions are synchronous or asynchronous, and whether we use objects or "services" (whatever those might be)—distributed computing is hard because it is *distributed* computing.

As far as A Note on Distributed Computing is concerned, it argues from false premises and arrives at conclusions that are not supported by these premises. In fact, the paper is largely irrelevant to modern middleware such as Ice.

Now, does it matter whether I use CORBA, or Ice, or REST, or something else? You bet it does! But that I will make the topic of other posts…

Cheers,

Michi.