“The Problem with Grand Unified Frameworks” by Douglas Thain
Technical Report 1492, Computer Sciences Department, University of Wisconsin

I prepared this text as introductory remarks I gave at a panel discussion titled “Is the Web Services-based OGSA an Architecture for all Grids?” at the IEEE Symposium on High Performance Distributed Computing (HPDC) on Monday, June 23rd, 2003.

This text was edited after the fact for grammar and clarity.
What I actually said that day might have been slightly different.

Hello, my name is Douglas Thain; I hail from the University of Wisconsin. I’d like to thank Brian Tierney for inviting me just a few days ago. I’m a little concerned that he didn’t ask me what position I would take! One thing I must get out of the way is that I am not “against” OGSA in the sense that it is an opponent to be eradicated! I’m sure that it will be put to some good use. It’s an ambitious task, and I wish the authors good luck.

That said, I believe that the Open Grid Services Architecture (OGSA) is yet another in a long line of Grand Unified Frameworks. The authors of OGSA need to convince us that they will avoid the mistakes of their predecessors, and not merely recreate them using tools starting with popular letters of the alphabet such as X and G. To avoid such mistakes, we need a little perspective on Grand Unified Frameworks.

Let me explain what I mean by a Grand Unified Framework. When an area of computing gains a certain level of diversity, we begin to get frustrated at dealing with all of the fiddly differences between similar systems. So, it’s common to imagine a shiny new system that will encompass all of our ugly differences and will permit us to ignore solved problems and move on to more interesting things. That’s what I mean by a Grand Unified Framework.

There have been many visions of Grand Unified Frameworks in different fields. A few decades ago, the idea of UNCOL promised to unify all compilers; in the 1980s, Sun RPC promised to unify all distributed computing; in the 1990s, the Java Virtual Machine promised to unify all microarchitectures. A favorite project of enthusiastic undergraduates is a universal framework for 3D shoot-em-up games. Some of these ideas have been envisioned but not built; some have enjoyed success; most fall somewhere in between. But none are ever universal, because each has peculiarities that are no better and frequently worse than the systems that they unify.

The problem with building Grand Unified Frameworks is that it is hard to distinguish between the fiddly little details and the fundamental structural issues. You can call the two syntax and semantics. Syntax is just the representation of data, whether it is in

English, ASCII, or XML. Semantics are the meaning behind the information. Most Grand Unified Frameworks have beautiful syntax but have lurking semantic landmines.

Let's step back 20 years to Sun Remote Procedure Call (RPC). It is quite a powerful system for remote execution, with several levels of name resolution, expiration and rebinding, a structured data language, a stub generator, and several robust applications. It's described by several RFCs, one of them an Internet standard. Consider this question: why wasn't this Grand Unified Framework used beyond Sun's own tools? (NFS and YP) Here we are all today doing distributed computing, and I am positive nobody here is building new systems with Sun RPC.

Sun RPC failed to become universal because a fiddly little detail had enormous semantic consequences. Recall that RPC has the flexibility to run over either UDP or TCP. (This is a false kind of "flexibility" that is common in Grand Unified Frameworks.) The stateless nature (with respect to clients) of an RPC service requires that any change to persistent data be done atomically and idempotently. That is a big deal! The semantics of all potential RPC applications are tied to the semantics of the most hostile transport method. Because of this, you can't do large data streaming nor can you do efficient transaction processing, nor can you discover and react to disconnected clients. That's fine for NFS and YP, but for other purposes, you must write a new protocol; this task is well within the capabilities of your average graduate student.

Even applications that can accept these constraints declined to use Sun RPC. For example, the Andrew File System (AFS) needed an RPC layer; they simply wrote their own. If you are a standards advocate, you might stomp and wail "proprietary solution." But what's the harm? There is no need for AFS to be syntactically compatible with NFS, because it is semantically incompatible. The only savings might be in code re-use, but to paraphrase Mark Twain, code re-use is a virtue that everyone praises but nobody practices. If anyone here has figured out the secret to code re-use, I'd like to know it.

Now, what does this have to do with Web Services? I frequently hear things like "SOAP can be carried over any sort of transport, even e-mail or FTP." Let me make something clear: This is nuts. E-mail is just like UDP: messages can be lost, re-ordered, or repeated. Thus, any "flexible" SOAP implementation is constrained by the properties of E-Mail. This is exactly the problem of Sun RPC: higher layers are imprisoned by the least common denominator in a "flexible" infrastructure.

I don't have time today for an exhaustive critique of Web Services. But let me drive the main point home: flexibility is not always a virtue. Different services will require different RPC-like layers for communicating control and data. One framework will not satisfy all implementations, and trouble results when we try too hard for generality.

Let me conclude with an example of syntax and semantics in a realm outside of computing. Consider an electrical wall outlet and D-cell battery. Both are similar; they provide electrical power. Of course, they have different semantics. The wall outlet provides 120V AC, while the battery provides 1.5V DC. They also have very different failure semantics. The wall outlet is fail-stop; it has a circuit breaker. The battery is fail-stutter; it loses voltage as load increases.

But, they also have a different syntax. The wall outlet takes a plug, while a battery takes a funny little cradle. This difference in syntax discourages unknowing users from mating the two systems. Now, suppose we got tired of the expense and frustration of using all these different types of plugs and cradles. Let's just give wall outlets and batteries identical interfaces, let's say an RJ-11 phone jack. You can imagine what sort of chaos would result.

Syntactic differences are not such a bad thing; they prevent incompatible systems from injuring each other. Have you ever noticed that the nozzles of diesel fuel pumps are a different size from the nozzles of gasoline pumps? This prevents you from ruining your passenger car with fuel meant for a truck. Syntactic differences are almost always a reflection of semantic differences.

The question is, should we spend our time talking about syntax or semantics? Once a semantic design is chosen, choosing syntax is easy. It's a choice between curly braces and angle brackets. (Frankly, I prefer curly braces.) But, building a complete working system that stays up for years is a much harder problem. Quite a few people here know this. (Larry's keynote, the Globus GRAM designers, The Condor-G group.) At HPDC, we should spend less time talking about Grand Unified Frameworks, and more time building and understanding working systems.

Thanks for listening.