

Web Services/SOAP and CORBA

April 27, 2002

Irmen de Jong (and others)

[corba \(at\) razorvine.net](mailto:corba(at)razorvine.net)

Contents

| | | |
|-----------|--|-----------|
| 1 | Introduction | 2 |
| 2 | CORBA compared to SOAP/Web Services..... | 2 |
| 3 | Hype is interesting | 3 |
| 4 | Disadvantages of SOAP/Web Services..... | 3 |
| 4.1 | Elaborate coding required at a low level of abstraction | 3 |
| 4.2 | Lack of interoperability, i.e. many competing SOAP implementations..... | 3 |
| 4.3 | Saturation of firewall port 80 (HTTP) | 4 |
| 4.4 | Lack of standard language mappings..... | 4 |
| 4.5 | Lack of standard services, e.g. events | 4 |
| 4.6 | Lack of performance due to the requirement to parse and transport XML..... | 4 |
| 4.7 | Lack of compile time type checking; harder debugging..... | 5 |
| 4.8 | Elaborate, difficult-to-read interface descriptions in XML..... | 5 |
| 4.9 | Why compile-time type checking (static typing) will not save you from everything | 6 |
| 5 | Business- and economic issues..... | 6 |
| 6 | Is CORBA really so heavy, and SOAP not? | 6 |
| 6.1 | Complexity | 6 |
| 6.2 | Fix what's wrong rather than reinvent the wheel | 7 |
| 7 | Code complexity when using SOAP | 8 |
| 7.1 | Example code..... | 8 |
| 7.2 | Why C#'s [WebMethod] tag doesn't really help | 9 |
| 7.3 | Other SOAP libraries | 9 |
| 8 | Loose coupling of web services, hmm... .. | 9 |
| 9 | Possible true advantages of SOAP | 10 |
| 10 | Concluding remarks..... | 10 |

1 Introduction

I've been using CORBA a few years back (not anymore recently) but the recent hype about so-called 'web-services' has triggered my memory a bit. I was wondering what the big deal is. I convinced myself that CORBA and Web Services (SOAP etc.) are very similar, but strikingly different at other aspects, and I wanted to know more about this matter. A message posted to the UseNet newsgroup `comp.object.corba` led to an interesting discussion, which I tried to reproduce in the form of this document.

Major contributors (in no particular order): Michi Henning, Mark Woyna, Rosimildo daSilva, Don Box, Jim Melton, Jacob Jose Cherackal, Jesper Ladegaard, Rob Ratcliff, Gerald Brose, Paul Campbell, Jason Voegele. Thank you! (if any of you read this and feels uncomfortable about this document containing your comments without being quoted, sometimes a bit out of context, sometimes with a few phrases removed or changed, or just torn apart completely, please let me know. It was my intention to gather all interesting information from the discussion and cluster it together in a single document. I hope you find this ok.).

Newsgroup thread is available at Google:

http://groups.google.com/groups?hl=en&lr=lang_nl|lang_en&ie=UTF-8&threadm=a12ju0%243bq%241%40news1.xs4all.nl&rnum=2&prev=/groups%3Fnum%3D20%26hl%3Den%26lr%3Dlang_nl%257Clang_en%26scoring%3Dd%26q%3DComparison%2BCORBA%2BSOAP%2BIrmen

(Or search for "Comparison CORBA SOAP Web Services" in the newsgroup `comp.object.corba`, the thread started at 2002-01-03)

2 CORBA compared to SOAP/Web Services

First and foremost, you *cannot* compare CORBA to SOAP. SOAP is the protocol part of the RPC mechanism. The corresponding part of CORBA is IIOP, so you can only compare SOAP and IIOP directly. CORBA is much bigger than IIOP alone.

A little table to show you what in my opinion the corresponding parts of Web Services and CORBA are. The table is not meant to give a one-to-one comparison, which component of CORBA corresponds precisely to what component of Web Services. There's no need for that. For instance, UDDI is a nice example; it falls somewhere in between Naming Service and Trading Service in terms of the functionality it provides. It is not exactly one or the other, and it would be a waste of time to try too hard to *force* a correspondence that just isn't supported by the facts.

| Item | Web Services | CORBA |
|----------------------|------------------------|--|
| Protocol | SOAP, HTTP, XML Schema | IIOP, GIOP |
| Location identifiers | URLs | IORs, URLs |
| Interface spec | WSDL | IDL |
| Naming, directory | UDDI | Naming Service, Interface Repository, Trader service |

The general opinion of the people on the UseNet newsgroup `comp.object.corba` appears to be similar to my own: namely that the currently so over-hyped "*web services*" are just a new (but not a better) way of doing various things we have had since CORBA 2.0.

A little CORBA history:

October 1991: CORBA 1.0 (object model)

August 1996: CORBA 2.0 (full ORB interoperability and standardized C++ language mapping)

Summer 1998: CORBA 2.3 (standardized Java language mapping)

Late 1999: CORBA 3.0 (Quality Of Service, messaging)

So CORBA has been around since 1991, and since 1996 it has been a strong, useful, standard that produced fully interoperable ORB products from various vendors.

Interesting enough, SOAP only appeared in 2000. That's 9 years after CORBA 1.0.

Web Services are basically a reinvention of the CORBA wheel, with most of the important pieces, i.e. security, notification, *missing* or under-specified. The amount of articles referring to Web Services/SOAP as "being built on internet standards" is disturbing, because CORBA (or at least, IIOP) already *was* an Internet Standard. IIOP actually means Internet Inter-Orb Protocol. In the next chapters these statements are clarified.

3 Hype is interesting

Much more interest is currently going towards SOAP and Web Services than other, proven technologies such as CORBA. It appears that as soon as a technology becomes more or less accepted, proven, and generally used, it quickly loses attention. This may have several reasons:

- In our industry, unfortunately it's considered far more useful and intellectually challenging to discuss new, non-existing technologies than proven and existing ones ("boring!")
- We can use it as a possible scapegoat. We choose the "future-proof", cutting-edge (but non-existent) technology in a project. If our project then fails, we can blame it on the dysfunctional technology rather than our own mistakes. Some people may consider this handy to ensure continued employment.
- When technologies mature, they become common or even ubiquitous. With the right tools, they might even become invisible (but not less present!). And nobody talks about invisible things or things they don't even know about, but do exist in the system they're creating or using.
- The thing is hype, so everybody already talks about it, and we cannot stay behind. This looks like an endless loop, but someday somebody will come and burst the bubble. The hype curve then quickly diminishes.
- Publishers have to sell magazines and books. Stable technologies aren't very interesting to read or write about.

4 Disadvantages of SOAP/Web Services

There are quite a few issues with Web Services and SOAP. Below is a collection of the most important ones.

4.1 Elaborate coding required at a low level of abstraction

Performing a SOAP request is quite elaborate; the SOAP programmer has to construct a message, put the arguments in the message, and then send the message. After that, wait for reply, parse the reply XML, and fish out the data that's the result of your invocation. The web service is not available to the programmer as a well-known language construct such as the regular method call on an object. OK, no system-level network programming has to be done, but hey, we didn't have to do that with good old DCE RPC either.

CORBA is much more functional and closer to the programming style of the host language (i.e. regular method calls on objects). It doesn't require the programmer to fiddle with Call objects, set method names, provide serializers and deserializers, individually set parameters, extract results, check for faults, etc.

SOAP and CORBA are the same only if you compare them at the protocol level: both can be used to do RPC. So, again, one should not compare SOAP and CORBA directly, but rather SOAP and IIOP. When we do that, there still are some issues with the SOAP protocol: where is polymorphism? In fact, where are the objects!? It looks suspiciously like straight procedural RPC, instead of calling methods on remote objects. Of course object-orientation doesn't have to be a requirement, but it's definitely handy to have built-in support for down-casting and static type checking for call dispatch. These are possible only if polymorphism is built into the platform as a first-class concept. It is possible to implement polymorphism over SOAP (just as I can implement it in a non-OO language such as C), but I shouldn't have to as an application developer.

4.2 Lack of interoperability, i.e. many competing SOAP implementations

Consider the following counter argument to the statement from the paragraph title: "If these implementations all conform to the SOAP standard, they should interoperate, even when they may not be portable." *This is not true*: the standard *itself* is

currently full of ambiguities that lead to interoperability problems. Not only will implementations have to improve in order to interoperate, but also the spec itself will have to mature.

The Web Services camp is quite fractured today. Web services have interoperability that resembles CORBA around 1994 (i.e. before the CORBA 2.x spec). CORBA has had 10 years to mature. It won't take that long for Web Services, but there is a long way to go.

4.3 Saturation of firewall port 80 (HTTP)

It's true that OMG waited until the last minute to standardize firewall traversal, but now there's a specification for this. (<http://www.omg.org/cgi-bin/doc?orbos/01-08-02.pdf>) There are also some IOP-over-HTTP tunnels but they are far from optimal. Now there is another party wanting to travel over port 80: the Web Services' HTTP transport. Simply assuming that life will be good when running everything over port 80 is a laugh. The first virus that's propagated through a SOAP call will result in a very tight lock down of that port.

4.4 Lack of standard language mappings

There is no portable, standard API for SOAP, so the developer is left in the dark about how to map SOAP data to implementation data and vice versa. If you use a web services product for, say, Java from one vendor and then want to switch to a web service product for Java from another vendor, you get to rewrite all your code, because the new vendor's toolkit and API are different.

There is *no single way* to map your program's objects and data model to SOAP and vice-versa. There is no such thing as a language and platform independent interface and object model specification language as CORBA's IDL (Interface Definition Language) together with the standardized programming language mappings.

Consider the following counter argument to the statement from the paragraph title: "Widespread support for the WSDL standard means you won't have to, just feed the WSDL to the new vendor's system." *This is not true:* WSDL doesn't describe *implementation*, does it? So, even with a WSDL compiler, the best I could hope for is standardized APIs for interacting with a particular web service. But I still won't get standardized APIs for interacting with other parts of the run time, do I? There is more to a language mapping than just translating interface definitions!

Furthermore, CORBA has had the fully portable POA (Portable Object Adapter) specification for years. Not only the client code you write, but also the *server* code (where you implement and serve the distributed objects) is portable across ORB vendors.

Because there is no standardized SOAP language binding, the client and server code written by the SOAP developers *is not portable*. If I, as a developer, commit to any particular SOAP library or framework, I've just locked myself into a proprietary solution, whereas the OMG has spent considerable time standardizing both the client and server side libraries of CORBA. As long as I stick to the CORBA standard, I can easily switch ORBs on both the client and server side. How long are we going to have to wait for standard SOAP language binding? How long are you prepared to wait? Don't hold your breath. From the time a standards body first looks at a language mapping to the time that mapping is commercially available from a number of vendors (and actually portable), several years elapse. In the mean time, it's vendor lock-in and non-portable code. OMG taught us this with CORBA in its earlier years.

There is no alternative to a *standardized language mapping* when we are interested in portable code. I think many developers argue that they don't want to go back to the days where they have to rely on one particular vendor and be at that vendor's mercy when it comes to upgrading, getting new feature support, or being shielded from that vendor's decision or obligation to pull out of this particular market...

4.5 Lack of standard services, e.g. events

There is only one standard service available for Web Services: the UDDI interface service. Various other important services are not specified. There is no notification service (event service), no transaction service, no security service,

4.6 Lack of performance due to the requirement to parse and transport XML

The use of XML places a heavy burden on your system. Despite the relative simplicity of XML, there are huge memory and CPU requirements compared to other solutions. Generating and parsing XML documents is a time consuming task that

also needs a lot of memory compared to the actual data that is in those documents. Complex schema's or DTDs slow down the processing of the structure and content of the document even more.

Also don't forget that those documents have to travel across the network. In a high throughput system, the sheer volume of the XML data that has to travel over the wire may be too much (with XML the actual information is only a small portion of the total data; there is a high overhead). Various people report that SOAP performance is staggeringly bad and that the wasted memory and network bandwidth is enormous.

4.7 Lack of compile time type checking; harder debugging

While very simple interactions may be easier to build using Web Services, debugging everything is definitely harder. With SOAP, you won't get a type error until the XML message is validated on the *server at runtime*. Unless of course you use a SOAP framework on the client, which will basically duplicate the functionality of the CORBA ORB in this regard – and voids the argument that no extra software is necessary.

The CORBA IDL language bindings support type safety. Static stub classes provide compile-time type checking. This means that an attempt to send a parameter of the wrong type with an invocation will result in a compile-time error. But to be honest, compile-time type checking doesn't save you from all of your problems. See below.

4.8 Elaborate, difficult-to-read interface descriptions in XML

Let's compare some SOAP interface XML with the corresponding CORBA IDL definitions.

```
<esd:CType name="EchoData">
  <esd:item name="aLong" type="xsd:int" builtin="true" array="false" inout="false"/>
  <esd:item name="aBool" type="xsd:boolean" builtin="true" array="false" inout="false"/>
  <esd:item name="aString" type="xsd:string" builtin="true" array="false" inout="false"/>
</esd:CType>

<esd:Method name="getData">
  <esd:InParam name="GetDataRequest">
    <esd:item name="l" type="xsd:int" builtin="true" array="false" inout="false"/>
    <esd:item name="b" type="xsd:boolean" builtin="true" array="false" inout="false"/>
    <esd:item name="s" type="xsd:string" builtin="true" array="false" inout="false"/>
  </esd:InParam>
  <esd:OutParam name="GetDataResponse">
    <esd:item name="return" type="typens:EchoData" builtin="false" array="false"
inout="false"/>
  </esd:OutParam>
</esd:Method>
```

versus

```
struct EchoData {
    long    aLong;
    boolean aBool;
    string  aString;
};

EchoData getData( in long l, in boolean b, in string s );
```

XML is powerful but your average C/C++/Java programmer will have troubles reading it and deducing the meaning, compared to the rather straightforward IDL.

Of course there are or will be tools that generate the appropriate WSDL XML interfaces for you, straight from a Java class interface for instance. However, the generated code is ugly. There have been many attempts in the past for such automatic mappings: Java to IDL, the COM-CORBA mapping, the CORBA-DCE mapping, etc. They all suffer from the same problem: minor mismatches in the type system, scoping rules, and object model semantics lead to generated code that looks like a dog's breakfast and is so ugly that no-one wants to use it.

However, there is of course the possibility that if the code is generated the right way - adhering to industry standard patterns and idioms - it will actually be useful. But probably only the 80% common case is reasonable or even completely transparent; it's the 20% edge case that is hideous. Experience shows that the only case where code generation works is when application programmers don't directly call the generated code. Examples are GUI generators and parser generators, for which code generation has been highly successful. However, as soon as code generation is used for bridging (to adapt one API to another or, worse, to adapt one type system to another), experience shows that code generation is unsuccessful because it lacks the human insight that is necessary for crafting ergonomic and efficient APIs.

All that aside, it's still a bit strange that some people apparently prefer to write an implementation first and then, later, to find out what the interfaces look like :-). They write code (implementation!) in Java and C#, and then afterwards process these files to generate the WSDL interface description that describes the service. This is bad practice for various reasons. It generally is better and safer to *first* design the interface (contract) and *then* build the software (implementation and clients) that conforms to this contract. I think that tools that convert readily available services implementations to interface descriptions (being WSDL or CORBA IDL) should be treated very suspiciously, and that doing it the other way round is to be preferred.

4.9 Why compile-time type checking (static typing) will not save you from everything

This is also known as "What a Compiler Can and Cannot Do". In theory we need static typing, because then we will get easy compile-time type checking. This prevents us from nasty typing errors at run time. However, it is *not a sufficient contract*. OK, you got the argument count correct, and yes, you sent an Integer when it was expected. You *still* have to write all the same unit tests to demonstrate the *behavior* is what you expected.

If you are putting dynamic code into production that fails because of argument counts or incorrect data types then, frankly, you did not test your software for the *behavior* that was expected either! You probably don't write enough tests in your static language too. The compiler maybe gave you a warm feeling by compiling your code, but I bet there are still run-time bugs in there.

5 Business- and economic issues

There are not only technical issues to consider when choosing for SOAP. The business issues (security, Quality of Service, licensing, etc.) surrounding the use of Web Services still have a lot of questions to be answered. Architecting a system to utilize a third-party web service will be a difficult business decision. Would you bet you business on an untrusted web service? What happens if the company shuts down? Has a bug in their implementation? Will the service scale? Will web service hosting companies allow *anyone* to invoke their services? How will they get paid? Who do I turn to when the service provider goes belly-up?

What about the many XML schemas being developed, resembling the EDI standards of the past. It is unlikely that there will ever be a single schema defined for most vertical industries that is accepted by all parties. What happens when your company chooses the *wrong* definition of a purchase order? What if your potential business partner has selected the *other* definition? Much effort can be spent trying to unify disjoint schemas being pushed by business groups with lots to lose if their schema doesn't win.

A big plus for SOAP and Web Services is the momentum that it has because many big players actively support it, such as IBM and Microsoft. Also it reuses the big infrastructure already built for the Web and the HTTP protocol.

6 Is CORBA really so heavy, and SOAP not?

6.1 Complexity

An article from Cape Clear Software, published on www.webservices.org, tells us that "Another issue faced by organizations is a mismatch between the skills of Web developers and authors, and the skills required to build even a simple CORBA client." and they continue that a "typical B2B developer" is a mainstream developer using Visual Basic, and while able to build B2B web services, is "often not qualified to build CORBA clients". I find that comment rather silly. Not only does it take only 6 lines of code to get an object reference from the naming server, and call a method, but while

distributed programming is not the hardest thing on earth, it does require above-average skills regardless of the underlying technology!

(The article is available at [http://capescience.capeclear.com/articles/content/CORBA and Web Services.pdf](http://capescience.capeclear.com/articles/content/CORBA%20and%20Web%20Services.pdf)).

Another often heard argument against using CORBA is the requirement of an ORB being present on all machines that participate. Usually there is no (suitable) ORB out-of-the-box, especially not on client machines.

The Web Services community has an advantage here; they can happily construct some XML, pipe it to port 80, and listen for an answer, tying things together using Perl or Python or whatever. Now look closer. In fact, you don't *really* need an ORB on the client to do simple CORBA request. If you're a *real* programmer, you can use IIOP directly! Read the IIOP specification and construct IIOP requests directly from your application. It all boils down to doing requests and waiting for replies. So, what's the SOAP client programmer going to do? Are they really going to use raw sockets and craft XML by hand, or are they going to use a SOAP toolkit/framework that provides all the heavy lifting? If so, it sounds like the same problem! The CORBA ORB is there for you to do the heavy stuff!

The interesting thing is that, as soon as you beef up the protocol to the level where it meets real-life functional and performance demands, you end up with something complex, like it or not. The reason that RPC protocols are complex is not because the people who designed them were idiots or technocrats who like complexity, but because the functionality you need to build forces the complexity on you. (Plus some complexity added through design by committee, but that's a different story...)

Below is an example showing the amount of coding required by a CORBA client vs. a client using a SOAP framework. Not only does the CORBA client have significantly fewer lines of code, the level of abstraction is much higher. The client simply invokes an operation on the stub/proxy just like it was a local object. The SOAP programmer, however, has to construct a message, put the arguments in the message, and then send the message.

6.2 Fix what's wrong rather than reinvent the wheel

Rather than reinventing the wheel or throwing out the baby with the bath water, why not fix what's "wrong" with CORBA? Here are some of the arguments found in the preceding chapters:

- **CORBA is too complex**

If CORBA is too complex, take a subset or create a simpler higher-level API. There is no sense in throwing years of development out the window due to some perceived superficial "flaws". Also, people that find CORBA too complex may need to obtain better knowledge about the subject. Distributed systems programming *is inherently complex*.

- **CORBA is too expensive**

Unlike a few years ago, when the argument was true, there are now many fully compliant free ORBs available for various platforms and various programming languages, C++, Java, Python and others. When you need the extra power you can upgrade to commercially supported ORBs. (Note that Sun's Java SDK readily contains a CORBA ORB and tools).

- **IIOP doesn't work through firewalls or proxy web servers**

Commercially available IIOP firewalls do exist. One could also standardize the use of HTTP tunneling protocols for CORBA.

- **I can't read IIOP like I can XML**

Again, use some type of human readable protocol like SOAP when performance isn't an issue to make it easier for tool developers. Personally I only want to see the underlying communication protocol when developing a higher-level communication system. When developing distributed applications I don't want to see what's going over the wire, I'd rather concentrate on the important parts of the system! However, there is a counter example; HTTP became dominant on the Internet due to its simplicity and human readability even though the performance was poor. There may be a lesson here. WSDL interfaces have the possibility to describe the interface in humanly readable text. The description is part of the interface. CORBA's IDL doesn't have this, but why couldn't this be added to the IDL spec? It shouldn't be that hard to not only store the IDL interface in the interface repository, but also the associated textual description. Also there are at least two free ways to trace CORBA calls and show them in a human-readable format: Ethereal (<http://www.ethereal.com>) is an interactive network analyzer. Corbatrace (<http://corbatrace.tuxfamily.org/>) is Java code to intercept CORBA calls made by a Java application, and dump them to XML.

- **CORBA isn't being hyped by IBM and MS**

Should we really be making decisions driven by marketing rather than striving for incremental progress of existing technologies? Will we benefit from such decisions in the long run? Personally, I think life is much too short to take too many steps backwards.

7 Code complexity when using SOAP

Here are the basic steps for creating a client that interacts with a SOAP RPC-based service as described in the Apache SOAP v2.2 documentation:

1. Obtain the interface description of the SOAP service.
2. Make sure that there are serializers registered for all parameters that you will be sending, and deserializers for all information that you will be receiving back.
3. Create the `org.apache.soap.rpc.RPCMessage.Call` object.
4. Set the target URI into the Call object using the `setTargetObjectURI(...)` method.
5. Set the method name that you wish to invoke into the Call object using `setMethodName(...)`.
6. Create any Parameter objects necessary for the RPC call and set them into the Call object using the `setParams(...)` method.
7. Execute the Call object's `invoke(...)` method and capture the Response object which is returned from `invoke(...)`.
8. Check the Response object to see if a fault was generated using the `generatedFault()` method.
9. If a fault was returned, retrieve it using the `getFault(...)` method, otherwise extract any result or returned parameters using the `getReturnValue()` and `getParams()` methods respectively.
10. For every subsequent method call, repeat steps 3 through 9.

For CORBA, essentially the following steps have to be performed:

1. Initialize the ORB.
2. Obtain a reference to the root naming context by requesting the orb to `resolve_initial_references`.
3. Narrow the reference down from a CORBA object reference to the actual NamingContext class.
4. Obtain a reference to the required CORBA object by querying the naming context for the object name.
5. Narrow the reference down from a CORBA object reference to the actual class.
6. Invoke the method on the object reference as if it were the object itself.
7. For every subsequent method call, repeat step 6.

Note that in the CORBA case, every step is a single statement.

7.1 Example code

This is an example SOAP invocation in java (using the Apache SOAP library):

```
URL url = new URL("http://localhost:8080/apache-soap/servlet/rpcrouter");
Call call = new Call();
call.setTargetObjectURI("urn:Hello");
call.setMethodName("sayHelloTo");
call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
Vector params = new Vector();
params.addElement(new Parameter("name", String.class, "Mark", null));
call.setParams(params);
Response resp = null;
try {
    resp = call.invoke(url, "");
    if ( !resp.generatedFault() ) {
        Parameter ret = resp.getReturnValue();
        Object value = ret.getValue();
        System.out.println(value);
    }
    else {
        Fault fault = resp.getFault();
        System.err.println("Generated fault!");
    }
}
catch (Exception e) {
```

```

    e.printStackTrace();
}

```

Compare this to a CORBA invocation in Java:

```

try {
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
    org.omg.CORBA.Object rootObj = orb.resolve_initial_references("NameService");
    NamingContextExt root = NamingContextExtHelper.narrow(rootObj);
    org.omg.CORBA.Object object = root.resolve(root.to_name("AcmeMyService"));
    MyService myService = MyServiceHelper.narrow(object);

    int ret = myService.sayHelloTo("Mark");
} catch (MyServiceException e) {
    System.err.println("Generated fault!");
} catch (Exception e) {
    e.printStackTrace();
}

```

Remember that this is a single method call. When you want to call more methods, the `setParams` part and the `call-getvalue` logic of the SOAP example has to be repeated. In the CORBA example, we just add another `myService.methodName("argument")`.

To be fair, using other SOAP libraries can reduce the required code dramatically to about the same as the CORBA case, with very similar steps. But there is a problem here, see the paragraph below about other SOAP libraries.

7.2 Why C#'s [WebMethod] tag doesn't really help

I've heard that in C#, when I want to expose a method as a web service, I write: `[WebMethod]` in front of the call. When I want to call a web service, I generate a proxy around the web service and then call it like any other function. That sounds very convenient.

But to believe that this will solve the distributed computing problem for anything but a toy application is naive in the extreme. Have a look at CORBA's POA [Portable Object Adapter] and the large number of implementation choices it supports. This is no accident: in order to allow developers to build scalable and high-performance implementations, all this variety is necessary. If it were possible to build anything like a realistic application by simply writing `[CCORBAMethod]` into a C++ class, believe me, CORBA would have done it years ago.

7.3 Other SOAP libraries

The Apache SOAP library v2.x is not user-friendly. It makes doing SOAP a complex task. Some say that this library is not a good example, and indeed, there are others that are much easier to use (eSOAP for instance, <http://www.embedding.net/eSOAP>). The eSOAP toolkit does appear to be much easier than the Apache toolkit, but this is only because they've basically re-implemented the same stub-based framework as specified by the CORBA spec, only in a *proprietary* manner. And exactly, this is one of the major disadvantages of SOAP mentioned in the chapter above.

8 Loose coupling of web services, hmm...

The lack of a strict type model or strict interface specifications is beneficiary for the loose coupling that would be possible with web services. But the whole "loosely coupled" argument around web services is just a lot of hot air. Many proponents of web services have touted the "loose coupling" as a big advantage. In particular, the use of XML as an encoding is often cited as providing this "loose coupling" because, after all, the receiver can simply ignore those tags it doesn't understand. This thought is deceptive, because when you think about it, the only way you can have reliable communication between two parties at the semantic level is if they agree on a common and complete type system. (But, of course, as soon as you have such a type system, the whole notion of "loosely coupled" goes out the window because changes in the conversation (surprise, surprise) require a change in the type system (and, even for loose typing, changes to the source code). In fact, what people think of when they say "loosely coupled" requires brains: human insight is what we use to make up for the

lack of a strict type system in human interactions. Unfortunately, computers are nowhere near getting such capabilities. It follows that "loosely coupled" as far as type systems are concerned will remain a dream for many more years.

Even in an untyped scripting language you have to know what the interface looks like and what its behavior is (the *semantics* of the interface) to be able to use it. Also CORBA's Dynamic Invocation Interface is of limited use; essentially it only saves you the trouble of having to generate static stubs and skeletons. Fundamentally you are still writing code to the specific interface. For instance, the knowledge that some "calculator" interface has methods called "add" and "multiply" and what the intent of those methods are, is embodied in your client code. The mechanism isn't dynamic in the sense of being *adaptive*. A true dynamic (adaptive) client would be able to discover and use *any* calculator interface. Dynamic Invocation Interface or Interface Repositories take us no closer to this level of adaptiveness. True "dynamic discovery and use" of business interfaces is essentially impossible to automate, however, this claim could sure be made by SOAP proponents/marketers (just as it could be used for CORBA). Implying that you can build systems without rigorous interface specification is always a powerful selling technique to the clueless.

9 Possible true advantages of SOAP

As already noted above, a big plus for SOAP and Web Services is the momentum that it has because many big players actively support it, such as IBM and Microsoft. It also reuses the big infrastructure already built for the Web and the HTTP protocol.

Furthermore, SOAP may have stronger support from the open source community (but CORBA is quite strong here too), and stronger support for recent programming languages that people will be using more now and in the near future (e.g., python, perl, C#, etc). But *standardized* support won't arrive in the near future.

Interestingly enough, SOAP's greatest advantage and disadvantage at the same time, is its lack of standard above it and below it. SOAP is as simple as the implementer of the middleware wants it to be. SOAP may be nasty and complex but flexible when using tool A, but very easy (though inflexible) when using tool B.

The SOAP community has the possibility to happily construct some SOAP XML and send it out on the net, then listening for an answer. Using simple (and free!) script tools this can be easily done, so the barrier to use SOAP appears to be very low. It has not been said that this is the preferred programming model, however!

Partly because of that, another advantage may be ubiquity. SOAP may become the de facto standard of doing RPC over the web. It has the potential to become a lingua franca that you can fall back on when all else fails. As it stands, CORBA's IIOP has not penetrated the Internet as HTTP has. If SOAP can run on HTTP or any other Internet Protocol, I would say that it is bound to enter the domain the IIOP has not. It remains to be seen whether SOAP will become ubiquitous, but it will be used where no better alternative is available (then it becomes much like comma-separated data for exchanging spreadsheet files between spreadsheet programs). I'm not sure if that is good or bad.

Some may argue that it is a good thing that the protocol is readable by humans. This may allow for easier developing and debugging, but is that true? What real use is human readability at the protocol level? In any inter-corporation data exchange scenario it's going to be encrypted anyway. And are developers really interested in what passes over the wire? I'd rather not think about that and concentrate on the important parts of the application.

10 Concluding remarks

It's amazing that many in the SOAP community have basically dismissed CORBA/IIOP as being too "difficult" to use, and not suitable for the Internet. They state that XML is so much easier to use, but yet they don't actually expose the XML to the programmer. Instead, they develop incompatible, proxy-based frameworks based on the same designs as the CORBA vendors. CORBA is heavy-duty because distributed applications appear to require that. Once the Web Services community discovers all the missing pieces, Web Services will be quite heavy duty also. Depending on the framework selected, SOAP programming is no easier, and possibly more work than the equivalent CORBA approach. And regardless of the framework selected, the resultant SOAP code *is not portable* between frameworks.

In the long run, the hype about Web Services will diminish (as all hypes do), and CORBA and Web Services will both have their place. Just as CORBA and COM, and for that matter RMI, coexist. When necessary, bridges will be used to tie everything together. It looks like we're going to use SOAP to talk to the outside world, and we'll be using CORBA in the inside. In the short term, it will take a number of years for SOAP to mature, and the *missing* pieces to be defined.

Some argue that a thorough understanding of Software Architecture would provide you with the true benefits and disadvantages of the platforms and technologies of the past, the present and the future. Those people will be few, I think, because this requires great skill and expertise. Also, we developers are so busy continuously reinventing the wheel that there rarely is time to actually evaluate the architectural aspects of a system and learn from them. Some may find that there isn't a single thing that could be called *innovation* in the entire SOAP and Web Services area. It's just repackaging of already solved problems and, unfortunately, with many mistakes of the past being repeated. Although being a strong statement, it certainly has some truth in it. Many "technologies" are driven more by marketing, hype and revenue rather than sound technical decisions that would encourage incremental progress.