Rethinking the Java SOAP Stack

Steve Loughran
HP Laboratories
Bristol, UK
Email: steve_loughran@hpl.hp.com

Edmund Smith
School of Informatics
University of Edinburgh, UK
Email: esmith4@inf.ed.ac.uk

***Abstract*— This paper examines the current SOAP APIs in Java, and in particular the *Java API for XML-based RPC*, commonly known as JAX-RPC, which is effectively the standard API for SOAP on the Java platform. We claim that JAX-RPC, and indeed any SOAP API that relies upon a perfect two-way mapping between XML data and native language objects is fundamentally flawed. Furthermore, we claim that the attempt JAX-RPC makes to extend the remote method invocation metaphor to SOAP services is counterproductive.**

We base our argument both upon experience with JAX-RPC and SOAP, and upon experience of previous distributed computing technologies. We argue that JAX-RPC is not capable of delivering on the SOAP design goals, but conclude by suggesting an alternate system, *Alpine*, which is free from many known flaws of existing systems, and should prove better able to deliver upon the promise of SOAP.

I. Introduction

In beginning any discussion of SOAP-based technologies, it is valuable to review the core features which made adopting SOAP attractive initially. We briefly characterise these as follows:

1) Simplicity: It is intended to be easy to work with.
2) Interoperability: It is more interoperable than binary predecessors.
3) Extensibility: The envelope/header/body structure allows extra data to be attached to a request, potentially without breaking existing systems.
4) Self-describing: Messages can contain type definitions alongside data, and provide human readable names.
5) Flexibility: Participants can handle variable amounts of incoming data.
6) Long-haul: It is designed to work through firewalls, building upon HTTP and other established protocols.
7) Loosely-coupled: Participants are not expected to share implementation code.
8) XML-centric: Built on XML and intended to integrate with XML-based technologies.

We will refer to these criteria throughout our discussion, as the desiderata against which any SOAP technology should be judged.

A. SOAP in Java

Communication with SOAP can be viewed either as XML-based remote procedure calls, or as a way of submitting XML documents to remote endpoints (optionally eliciting responses in the form of XML documents.) These two different perspectives represent the RPC-centric and message-centric viewpoints. In Java, the RPC-centric model has become the primary model for SOAP APIs.

The Java APIs representing the two different underlying perspectives are JAXM[1] [1] and JAX-RPC[2] [2]. We review each of these in turn.

1) JAXM: JAXM was written to support both basic SOAP, and more complex scenarios like asynchronous ebXML message exchange over SOAP. This flexibility introduces significant extra complexity into the design. Over time, the ebXML focus of JAXM has declined, while the API itself has been renamed SAAJ[3] [3].

In JAXM/SAAJ, the developer works with the SOAP message through Java interfaces derived from DOM[4] [4]. These are bound to a class that represents the body of the message, which provides various operations to manipulate the pieces. These include accessors and manipulators for the envelope, headers, body and any binary attachments.

JAXM does not provide significant transport support: the primary method of receiving JAXM messages is to implement and deploy a HTTP servlet. The sole method of sending a message is to ask a `SOAPConnectionFactory` for a `SOAPConnection` instance, and then make a blocking call of the endpoint.

JAXM has become an orphan specification. Had ebXML been more successful, it is conceivable that JAXM might have proven more popular, and made message-centric SOAP development in Java commonplace. As it is, JAX-RPC is touted as the recommended way to work with SOAP in Java.

2) JAX-RPC: In JAX-RPC[5] the XML representation of a message's encoding is hidden and developers work with Java objects created automatically from the XML data using a semi-standardised mapping process. Java classes can be automatically turned into SOAP endpoints, with each public method in the class exported as an operation with both a request message and a response message. The message structure is described in a WSDL descriptor and an associated XML Schema, both of which can be hand-written, or automatically extracted from the Java classes through introspection.

[1] Java API for XML Messaging
[2] Java API for XML-based RPC
[3] SOAP with Attachments API for Java
[4] Document Object Model
[5] The current edition of the JAX-RPC specification is version 1.1, and this is the version we discuss here. JAX-RPC 2.0 is discussed in section III-A.

Client side proxy classes can be generated from the WSDL files, proxy classes which again provide a method for every operation the service supports. In communications between systems running on the Java platform, the result is that methods called on the proxy class result in the passing of the method parameters to remote methods on an instance of the implementation class, a behaviour that superficially resembles Java RMI [5]. We will return to this in section II-C.

One effective architectural feature of JAX-RPC is the *handler chain*, which consists of an ordered sequence of classes which are configured to manage requests and responses. Using the handler chain, it is possible to add support for new SOAP headers to existing services, or to apply extra diagnostics, in a relatively transparent fashion. The dispatch of operations to Java methods, Enterprise Java Bean methods or other destinations is generally implemented by specific handlers, making the handler chain the foundation for the rest of the system.

JAX-RPC is widely implemented, both by open source projects (for example Apache Axis [6]), and by commercial vendors like Sun, IBM and BEA. Although these SOAP toolkits do all implement the appropriate version of JAXM/SAAJ to complement the RPC model, this feature is neither broadly promoted nor used. All the *evangelisation* of SOAP focuses on JAX-RPC, as do most of the examples in the vendors' documentation.

The bias is such that, for Java development, it is widely seen that JAX-RPC *is* SOAP.

B. The Hard Lessons of Service Implementation

The authors have recently been involved in developing independent implementations of a SOAP API for deployment [7]. This API, specified in a set of XML Schema (XSD) [8] and Web Services Description Language (WSDL) files [9] defines a service endpoint providing seven operations, which permit suitably authenticated callers to deploy distributed applications onto a grid fabric.

The development of this service was performed in a "pure way", by creating the XSD and WSDL files first. This approach is believed to aid in creating a platform-independent system, and represents current best practise. However, the XSD file for the service messages is approximately 2000 lines, including all the comments and annotations needed to make it comprehensible. That it takes so many lines to describe a relatively simple service is clearly one reason why this approach is so unpopular.

Many problems were encountered turning this service specification into functional clients and servers, problems that we attribute largely to JAX-RPC. In section II we discuss a number of the problems we believe this work highlighted. Section II-D outlines the particular problems we believe typical JAX-RPC oriented approaches to WSDL generation create.

II. The Fundamental Flaws of JAX-RPC

A. The Object/XML Impedance Mismatch

JAX-RPC attempts to turn an XML document into Java classes, using service specific mapping information. This is distinct from the kind of mapping performed by DOM implementations, in that the classes are "serialised" from the XML tree, not merely created to represent it (it is a semantic rather than syntactic mapping). This serialisation/deserialisation is an essential part of JAX-RPC, allowing method calls to be translated into SOAP requests, and responses translated back into Java objects.

We believe that the term *serialisation* downplays the nature of the problem, likening it to the more tractable problem of creating a non-portable persistence format for a class. Instead, we prefer to use the term *O/X mapping* to emphasise the similarities it has with the heavily studied *O/R mapping problem*[6]. Over a decade has been spent trying create robust mappings between records in relational databases and language-level objects, and there is still no sign of an ideal solution. There is significantly less experience in mapping between XML and objects, and rather than drawing on the experiences of the many failed attempts at O/R mapping, O/X mapping technologies appear destined to share a similar evolution.

At first glance, the O/X mapping problem facing JAX-RPC appears simple: create a Java object for each XML element, building a directed, acyclic graph when serialising to RPC/encoded SOAP or a tree when using document/literal messages. Read or write between attributes and class fields, bind to children and the conversion is complete... If only it were so straightforward. Undermining it all is a fundamental difference between the type systems of XML (especially that of XML Schema) and that of Java, making any mapping both complex and brittle.

1) Binding XML Elements to Java Classes: The language of XML Schema is much richer than the object model of Java. In Java, inheritance can extend a type, and change some existing semantics, but derivation by restriction is not explicitly supported. Java, in common with many object oriented programming environments, allows derived types to expand upon the capabilities of their parents. XML schema allows one to extend a type by restricting it, constraining attribute and element values. Java has no intrinsic model for this type of constraint.

This is a fundamental difference which means that one cannot accurately model an XSD type hierarchy in a Java class hierarchy. All one can do is inaccurately model it. Here, for example, a postcode is modelled by restricting a string:

```
<simpleType name="UKPostcode">
  <restriction base="xsd:string">
    <pattern value="[A-Z]{2}\d \d[A-Z]{2}"/>
  </restriction>
</simpleType>
```

[6] *"Object-relational mapping is the Vietnam of Computer Science"* - Ted Neward, 2004.

The actual result is going to be a simple class of type `String`: all restriction information will be lost in the transformation from XML Schema to Java. This is a fundamental difference, and one which would appear to remain intractable except in special cases.

Note that XML Schema offers other type extension mechanisms, such as substitution and derivation. These mechanisms have similar issues with mapping to the inheritance and type model of Java.

2) Mapping XML Names to Java Identifiers: Not all XML names can be turned into Java identifiers. XML names may begin with a letter in one of many Unicode languages, an ideograph or an underscore ("_") . They can be followed by any of the same characters, and also a hyphen "-" or a full stop ".". Some examples are: `schrödinger`, `_unknown.type-set`, and `String`.

Java identifiers almost comprise a proper subset of XML names[7]. Because of the much greater range of allowable XML identifiers, the system will often need to perform a non-trivial mapping from the XML names to valid class and package names. Package names are typically derived from namespace URLs if not overridden, as discussed in section II-A.6.

The translation is inordinately brittle: whenever a new version of Java is released, the logic must be updated to avoid new reserved words (like `assert` and `enum`), or the generated code will no longer compile in the enhanced language. Needless to say, such an upgrade will break any existing code that linked to old classes which made use of these names.

3) Enumerations: One example that deserves special mention is the way in which `xsd:enumeration` declarations are mapped into generated Java code. Before Java 1.5, there was no explicit `enum` clause in the language, and the JAX-RPC approach to SOAP enumerations is based upon a workaround. However, the problem of mapping enumerations from XML to Java is unchanged, regardless of the language version used: generate a set of identifiers, one for each value in the enumeration.

This appears a straightforward example of how O/X mapping should work. But what if the value of the one of the enumeration types is a reserved word? Our API (as described in section I-B contains a lifecycle state machine like this:

```
<xsd:simpleType name="lifecycleStateEnum">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="initialized"/>
    <xsd:enumeration value="running"/>
    <xsd:enumeration value="failed"/>
    <xsd:enumeration value="terminated"/>
    <xsd:enumeration value="null"/>
  </xsd:restriction>
</xsd:simpleType>
```

One element in this enumeration is reserved: `null`. However, the JAX-RPC specification states that an implementation must now enumerate all states as `value1`, `value2`, and so on, for the entire list. The enumeration names in the Java source no longer contain any informative value at all, other than a position number in the set. Any change to the enumeration could reorder the values, without this change being detected by code that used the enumeration. The defect would only show up in interoperability testing.

4) Unportable types: Some Java types are by nature explicitly unportable. One would not expect to be able to have a SOAP runtime serialise a database connection instance and have it reconstituted in working order at the far end, for example. One might hope that a `java.util.Hashtable` could be translated into some XML structure that could be turned into a platform-specific equivalent at the far end. But surely a `java.util.Calendar` object could be sent over the wire, with its obvious relationship to the `xsd:dateTime` type in XML Schema?

We can certainly attempt to send such times. They are readable on the wire, and are mapped into whatever the remote endpoint uses to represent time. Unfortunately, in this case, differences in expectations between Java and .NET date/time classes prevent the same time being received at the far end. If both client and server are in the UTC time zone all works well, but if either of them are in a different location, hours appear to get added or removed. Clearly a different expectation regarding time processing is at work.

This is an insidious defect as it is not apparent on any testing which takes place in the same time zone, or between Java implementations. It is only apparent when remote callers, using different platforms, attempt to use the service.

5) Serialising a Graph of Objects: XML is a hierarchical data structure, and can only describe trees or lists of data. Java classes almost invariably refer to other objects, often creating cyclic graphs of references. If such a cyclic graph is to be mapped into XML, the mapping infrastructure must recognise the cycle (a naive implementation would enter a non-terminating loop). Once the cycle is recognised, it must be addressed. The options appear to be:

1) Signal an error.
2) Insert cross references into the XML message, for processing by the mapper at the destination.
3) Break the graph by duplicating content in the XML.

The only one of these solutions which seamlessly marshals cyclic graphs of objects is the second: inserting cross references into the XML. The SOAP solution for this is described in section 5 of the specification [10]. This linking mechanism is only supported in RPC/encoded messages; the document/literal message format does not allow it.

JAX-RPC was originally based on RPC/encoded messages, but the alternate representation, document/literal, is now broadly agreed to be more flexible and generally superior. There is no way to marshal a cyclic graph into a document/literal message without custom code[8]. Any technology

[7] XML names beginning in "xml" (any case) are reserved, whereas in Java they are valid identifier names.

[8] This problem is covered in detail in "Effective Enterprise Java" [11], where it is termed *the object-hierarchical impedance mismatch*.

that attempts to map XML to a cyclic object graph will suffer from the same problem.

6) XML Metadata and Namespaces: As discussed in the previous sections, XML Schema provides a type system that is much richer than that of Java. One aspect not mentioned so far is the relationship between XML metadata, notably namespaces, and Java classes.

The problem is essentially as follows: each node in an XML message can have attached to it a namespace. There is no related construct in Java which can model this accurately. The choice that is normally made is to model it inaccurately by package names (mapping namespaces to Java packages provides many of the problems discussed in section II-A.2, since these are more examples of identifiers).

The problems that typically arise are of two kinds:

1) Mapping an incoming message to a web service object requires that either the namespace of either the operation itself or its parameters be guessed. This guessing can be wildly inaccurate when the web service's Java interface was generated from WSDL using package renaming.
2) When dynamic invocation is desired (service invocation without the use of pre-built stub classes) it can be very difficult to determine the correct namespaces for service invocations (the WSDL typically leaves this unspecified, meaning that for JAX-RPC services the WSDL is not a complete description of the service interface).

If more metadata were recorded with generated types, this problem would not arise. We therefore believe that future versions of JAX-RPC will address this problem by way of the annotation facility recently added to the Java language.

7) Message validation: When a message is received, the serialised form is generated and passed to the handlers for processing. In typical SOAP stacks, no validation of the incoming XML against the message schema is performed, and in particular any restrictions on the number of times an item is required are not checked. This forces the implementation code to follow one of two paths:

1) It could ignore the problem. If the client code and functional tests do not generate invalid messages (as is likely if they are also all written in JAX-RPC) then the problem will not be noticed, only only surfacing when a third party attempts to use the service.
2) The developers could write procedural logic to verify that the Java classes representing a deserialised message have a structure that matches their expectations given the schema. This requires an understanding of the schema, knowledge of the serialisation mapping and its potential trouble spots, and the willingness to write the tests to validate this extra logic.

We suspect that most services err on the side of ignorance, and do not validate their incoming messages adequately. This brings their ability to operate in a heterogeneous environment into serious question.

8) Inadequate Mixing of XML and Serialised Data: JAX-RPC and JAXM are two different views of the world. When JAX-RPC encounters a piece of XML which cannot be deserialised within a message, it creates a SAAJ `Node` to describe that part of the document tree. From that point on, the tree below the node is permanently isolated from JAX-RPC processing (in some sense the developer has sailed off the edge of the JAX-RPC world, and fallen into the universe of XML.) Any O/X mappings which may exist for data within this piece of the message are now inaccessible: all that is left is the low-level JAXM API.

This behaviour implies that incorporating arbitrary XML within a SOAP message is not an approved action, yet the ability to easily incorporate such XML is a key aspect of SOAP's flexibility and a major part of enabling it to be more extensible and less brittle than its predecessors.

9) Fault processing: JAX-RPC's fault handling encompasses two distinct and opposing viewpoints. On one hand, the `SOAPFaultException` type provides direct access to the XML elements of a generic SOAP fault. Users of this mechanism can construct SOAP faults with arbitrary contents[9] and process faults from any remote endpoint, regardless of its implementation type or version.

In contrast, the other fault processing mechanism offered by JAX-RPC aims to provide for loss-less marshalling of Java exceptions across the SOAP infrastructure. The implementation and platform dependence inherent in this represents the complete antithesis of the generic SOAP fault mechanism.

While this may seem a simpler approach for the application developer, mapping faults into Java exceptions with a specific static structure means that extra data included in the original SOAP fault is likely to be lost. For example, should a remote runtime include a hostname or stack trace, this information will be stripped out on the receiving side before the client can be made aware of it. The process is also inordinately brittle: if, for example, an updated endpoint were to throw a new exception, any callers built against earlier WSDL would have to revert to handling a generic `SOAPFaultException`.

While the goal appears to have been to provide near-seamless marshalling of Java exceptions whilst permitting the exceptions to be immutable, the technical details of the marshalling undermine the success of the approach even without the interoperability problems. For example, there is a requirement that special constructors exist to configure the created exception to preserve its immutability, and WSDL generation is unreliable unless sufficient debug information is included in the bytecode for the names of all parameters to the constructor to be visible.

We believe that the attempt JAX-RPC makes to seamlessly marshal faults is consistent with its general approach, but like so much of that approach, it undermines the interoperability and robustness of services created with it. By propagating Java's requirement that all posssible exceptions be declared into remote interfaces, it exposes platform implementation details. Interoperability is a major strength of SOAP, yet here

[9]In Apache Axis, this technique is used to add non-standard diagnostic elements such as hostname, HTTP error codes and stack trace to messages.

JAX-RPC tries too hard to mimic Java RMI, and abandons many of the gains SOAP would otherwise have over RMI. In our opinion, the only sensible approach for working with faults in JAX-RPC is to have services throw pure SOAP faults and leave it to the recipient to process them.

B. SOAP is not just RPC

SOAP's parentage includes XML-RPC [12] and (indirectly) COM/DCOM [13] and CORBA [14]. It was clearly designed at its outset to be a form of remote procedure call in XML, over HTTP. Over time, the world-view that lead to that choice has changed. Though it is often presented as a form of RPC, we would argue that it is coming to be seen as more powerful when viewed as a system in which arbitrary XML documents are exchanged between parties, perhaps asynchronously, and potentially via intermediaries.

In this world, the programming paradigms that seemed appropriate for an RPC infrastructure look out of place. On a fast network, RPC invocation is often a good choice, as other models of communication are harder to code, and their benefits are not readily apparent. A complex communication can be modelled in a few lines of code, rather than a state machine, and the synchronous nature of the communication makes it easier to to build a model of the state of the remote system.

When we begin to work over long-haul connections, however, or with large content (e.g. several megabyte attachments), the limitations of RPC become clear. The greatest of these is that RPC is synchronous. Although asynchronous behaviours can, with some difficulty, be introduced, this is not the natural way for RPC to behave. As content becomes larger and the network latency increases, the problems posed by synchronous calls become more and more acute.

Currently, our only option is to split network communication into a separate thread from the rest of the program. While this works, it provides the programmer no way to give the user effective feedback or control over the communications. There is no way to receive progress notifications or cancel an active call, despite the face that the underlying transport code invariably permits such features. This can cause problems when working with file transfers, foe example: one of the authors wrote a GUI front end to a service that could accept 15-30MB CAD files, and whilst multithreading could keep the UI responsive, there was no way to present an upload progress indicator or offer a cancel button. These are both features one expects in an application of this kind.

Again, following our principle that SOAP technologies should uphold the same desiderata as SOAP itself, we note that one reason SOAP was adopted was to simplify the task of working over long haul connections. By making it both difficult and complicated to work over a long connection, JAX-RPC fails to meet this criterion for a SOAP technology.

C. SOAP is not RMI

JAX-RPC suffers from a greater flaw than those classically associated with RPC invocation: it tries to make the communications look like Java RMI. Java's RMI system is a simple and effective mechanism for connecting Java classes running on different machines. It is an IDL-free system that relies upon introspection to create proxy classes and to marshal data. It works because the systems at both ends are running on the Java platform, typically different pieces of a single larger application. Even then, its implementation-first design means that it is brittle to change, and most effective when both ends are using the same versions of all classes.

In a system with shared code at both ends, objects can be trivially serialised and transmitted across a network connection. Exceptions are just another type of object, and so too can be sent over the wire. There is less need for an IDL, as Java interface declarations can perform much of the same role. And as the recipient is a remote object, state is automatic. One can even keep code synchronised by using a special class loader, one that fetches code from jointly accessible URLs.

JAX-RPC tries to reuse many of the programming patterns of RMI. For example, the runtime will attempt to serialise classes marked as `Serializable`, ignoring those fields marked as `transient`. It will even serialise complex compound objects where possible. The user appears to have a reference to something like an object, though one that represents the current conversation with an endpoint, not a direct endpoint proxy.

Recall that one of the perceived gains that drove the adoption of SOAP (section I) was that it enabled loose coupling between the components of a distributed system. SOAP strove to overcome many of the failings of precursor technologies like CORBA and DCOM, which worked over local area networks, and enabled rich bidirectional communications, but were not completely cross platform[10], and ended up being used to produce distributed object systems that were too tightly coupled.

While Java RMI provides convenience, the one thing it does not provide in any way is loose coupling. Interacting systems typically run from the same codebase, and each element of the distributed system contains many implicit assumptions about the rest of the system. By trying to turn SOAP into RMI, we imitate this architecture, and risk losing the very things we turned to SOAP for in the first place.

D. WSDL: an extra complication

The role of an interface definition language (IDL) has always been twofold:

1) Firstly, an IDL allows the creation of a definition of *the interface* of the remote system, independent of any particular implementation, programming language or environment. This is "interface" in the sense of *the implementation independent signature of the service*, and does not imply that an implementation language needs an explicit notion of interfaces. The interface is inherently implementation independent, and can be frozen or carefully managed with respect to versioning.

[10] Admittedly for arguably political rather than technical reasons

2) Secondly, the act of writing an IDL description forces the author to define the system in terms of the portable datatypes and operations available in the restricted language of the IDL. This can effectively guarantee portability, and is a significant improvement over interfaces defined in the implementation languages themselves, which invariably contain constructs which are not portable.

IDLs have many advantages for creating interoperable systems, yet the generally accepted practice for working with JAX-RPC discards all these notions. Instead of generating implementation classes from WSDL, the WSDL description is usually generated from the implementation classes using tools leveraging Java's Reflection API. We shall term this process *contract-last development*.

This has the following consequences.

- There is no way to ensure that a service's published interface remains constant over time. Every redeployment of the service or may change the classes, and hence the contract.
- Some aspects of the interface a service provides are not amenable to extraction simply from signatures of its constituent classes and methods. A given method might, for example, choose to extract attachments from messages programmatically, without declaring that attachment in the signature.
- No warnings about portability issues are available before deployment occurs. When defining a service using an IDL, the author typically knows when there are problems as the IDL will not compile. Yet with contract-last development, everything may well seem to work until the service goes live and a customer using a different language attempts to import the WSDL and invoke the service.

The alternative to contract-last development is clearly *contract-first development*. Although this is the better approach from the perspective of portability and interface stability, web service developers are not pushed in this direction.

One of the underlying causes of this has to be the sheer complexity of XML Schema and WSDL. The XSD type system bears minimal resemblance to that of current object oriented languages, and WSDL itself is over-verbose and under-readable. As evidence of this, consider the broad variety of products that aim to make authoring XSD and WSDL documents easier, and recall that such products were never necessary in the IDL-era of distributed systems programming.

Returning to the desiderata for SOAP, following a contract-last process sacrifices interoperability for ease of service development. Perhaps WSDL is not the appropriate language for describing SOAP services (we are certainly not enthused about it), yet the sole solution being advocated is not a major undertaking to fix WSDL's core flaws, it is to continue to encourage developers to hand over to their SOAP stacks the challenge of deriving a stable and portable service interface from the inherently unstable and unportable service implementation.

We are not proposing any changes to WSDL, merely mourning the fact that its over-complexity discourages contract-driven development more aggressively than any previous IDL ever did. We do observe that once the type declarations of a service have been moved into their own document, WSDL becomes much more manageable and this is a pattern of service definition which we strongly encourage.

III. Implications

We believe that only two categories of web service developer exist: those who are comfortable with XML and want to work with it, and those who aren't but end up doing so anyway. JAX-RPC provides a sugar coated wrapping that encourages developers who are relatively unfamiliar with XML to bite. Yet, as anyone who has written a web service of any complexity knows, the XML must be faced and understood eventually. In practise, the task of creating a real web service is made more difficult, not less, by the huge volume of code JAX-RPC introduces into a project.

JAX-RPC only superficially benefits developers who do not want to work with XML: by hiding all the details, and giving developers a model of remote method calls via serialised Java graphs, JAX-RPC makes it harder to write true, interoperable SOAP services. Not only that, but it introduces the O/X mapping problem, while retaining an invocation model that is inappropriate for long-distance networks and slow communications.

We argue that JAX-RPC greatly complicates users' software by introducing a complex and fickle serialisation system. The generation of WSDL from Java code, which JAX-RPC encourages, makes it very difficult to maintain version consistency of an interface, and creates significant interoperability problems.

On top of all of this, for users who do want to work with the XML (typically those whose first project did not!) JAX-RPC is inappropriate because it hides everything. Trying to integrate custom XML documents with JAX-RPC serialisations is possible, but very hard work. In Apache Axis, DOM trees get recreated when assigning or extracting them from `SoapMessageElement` implementations.

A. The Future

JAX-RPC has become a cornerstone of Enterprise Java [15], alongside RMI and RMI-over-CORBA. That is not by itself a bad thing, but we believe that it creates the misconception that developers can trivially migrate from RMI to SOAP. If they attempt to do, they will fall into the traps that JAX-RPC creates for them.

The 2.0 revision of the JAX-RPC specification promises to correct some of these flaws, but we believe many problems remain. It uses a new O/X mapping, the Java Architecture for XML Binding 2.0 (JAXB 2.0)[16]. This is a sophisticated framework for mapping XML into Java, having broad support for the generation of classes from XML Schema, and of XML Schema from annotated classes. Despite many improvements over its predecessors, we believe JAXB is simply a more

comprehensive approach to a problem that is fundamentally ill-posed.

JAX-RPC 2.0 continues the attempt to provide a metaphor of method-invocation on object instances across SOAP transport. This fundamentally unmaintainable metaphor, coupled with the automated generation of WSDL from Java source code, means that in our opinion JAX-RPC has not outgrown its limitations.

It is easy to understand the rationale behind the decisions made in designs JAX-RPC and JAXB 2.0. Working with raw XML is hard. Writing good XML Schema documents is hard. WSDL is exceedingly painful to work with. Asynchronous messaging is more complex than blocking RPC. Nevertheless, we believe that all these things will be part of any solution enabling developers to realise the promised interoperability and loose coupling of SOAP-based systems.

WSDL itself is also evolving, with the 2.0 revision of the specification nearing completion [17]. It is to be hoped that a new WSDL syntax will make it easier to specify service contracts, and some new features of the language (for example the `interface` construct) promise to increase the opportunities for reuse and extension of service interfaces. Unfortunately, the sheer breadth of the WSDL goal set makes it exceedingly complex, and this, coupled with the needless verbosity that it enforces, is likely to make it a difficult language to work with for some time[11].

IV. Alpine: a proposed alternative

We are in the preliminary stages of designing an alternative SOAP stack for Java, which we are calling *Alpine*[12].

A. Manifesto

Our goal is to create a SOAP stack that is easy to use, robust, and maintainable. In order to do this, we are adopting an XML centric approach. Alpine will make no attempt to map between XML and custom Java classes, instead providing access to the SOAP messages using modern XML support libraries, which make it easy to navigate an XML document. By avoiding O/X mapping we greatly decrease the volume and complexity of our code. Some may argue this will make Alpine more difficult to use, but experience shows us that simpler systems are typically more straightforward to work with, as they react in more predictable ways.

If a WSDL/XSD description of an Alpine-hosted service is required, the user will be required to write it: as we concluded in section II-D generating these from Java introduces unwanted implementation dependencies and hampers interoperability.

With so much stripped out, Alpine will be a SOAP stack reduced to its essentials: a system for managing the flow of messages through a set of handlers, and libraries to handle transport across supported the protocols. Core compliance with the SOAP protocol will be provided, namely envelope validation and `mustUnderstand` processing of headers. Developers will be expected to use XPath specifications to work with contents of the message; we are considering basing our design upon the "XOM" XML framework [18].

This will not be a SOAP stack that attempts to make SOAP look like Java RMI, nor will it prevent developers from being aware of the format of the messages sent over the wire. Instead, Alpine will just provide the basic housekeeping and handler chain management to make simplify web service development, leaving the interpretation and mapping of the XML messages to the applications themselves.

B. Design Goals

The full design goals are as follows:

1) Stay in the XML space as much as possible.
2) Take advantage of as much leading edge infrastructure as we can.
3) Adopt the the handler chain pattern of Axis/JAX-RPC.
4) Target SOAP 1.2 (POST) only, WS-I Basic Profile 1.1 [19].
5) Document/literal mssages only, not RPC/encoded.
6) Support XSD and Relax NG schemas.
7) Run server-side, client-side, and as an intermediary.
8) No support for JAX-RPC or JAX-M/SAAJ APIs.
9) Configurable procedurally, through the Java Management API (JMX).
10) Permit dynamic handler chain configuration during message processing.
11) One supported parser.
12) Run on Java 1.5 and later.
13) No provision of side features such as a built in HTTP server, or a declarative configuration mechanism. These are delegated to other products.

We believe the core of this design is likely to resemble JAX-M/SAAJ in in terms of classes, integrated with a handler chain based on the JAX-RPC/Axis model.

C. XSD validation

Although we are still unsure as to how complete our WSDL support will be, we note that document/literal SOAP messages can be validated simply by comparing the incoming messages to the XML Schema that describes them. Mainstream SOAP stacks do not do this at present, usually for performance reasons, but this means that the set of XML documents an endpoint might receive is significantly larger than the set of those considered valid by its XML schema. With no automatic validation, developers must either write both validation logic and corresponding tests themselves, or choose to ignore the problem. Given that there is no warning of potential problems before deployment, we suspect that many developers remain unaware of the problems they face.

Errors caused by the absence of logic to detect and reject illegal documents are unlikely to show up in development, especially if a test-centric process is not followed, but become inevitable once a service goes live, and callers using other

[11] As any SOAP stack developer will attest, bug reports involving handwritten WSDL are the ones they fear.

[12] Its inspiration being derived in part from nimble, lightweight Alpine-style mountaineering approaches.

languages invoke the service. Such insidious defects, defects that only show up in production, are always unwelcome.

There is a trivial solution to this problem, one that is common to other XML stacks. It is: validate incoming messages against the XML schema of the service. We aim to implement a handler which will do this, which, if included on a handler chain, will reject invalid messages. It will also be able to validate outbound messages, which should be useful during development.

D. A Community SOAP Stack

An open source project, by its very nature, is written by its users, and so it is critical to such a project's success that as many users as possible are able to contribute to its implementation. The split between XML in JAX-RPC's internal API, and mapped object instances in the external API, creates a gulf between the implementation of the stack, and that of end user applications. This creates a significant barrier between external and internal developers, making it much more difficult for users to contribute to the system's development. With Alpine, we hope to avoid such a split, because the XML emphasis, and hence the terms of reference, are consistent between user applications and the SOAP stack itself.

E. The Implications of Alpine

If Alpine succeeds, it will be a SOAP stack that requires an understanding of XML before it can be used. This might appear to be a barrier to the widespread adoption of the tool, and perhaps it will prove so. Unlike commercial SOAP vendors, we have no financial incentive to make our product broadly usable. We will, however, have a SOAP implementation which all its users should be able to understand and maintain. Furthermore, we believe that a good understanding of XML is needed to develop any robust web service, and that by forcing developers to acquire that skill early on in the development process, we help them avoid being forced to learn it just before their shipping deadlines are missed.

This may seem somewhat ruthless: to deny the right to write web services to developers who are and wish to remain ignorant of XML. However, we have to ask: *if they do not want to know XML, why are they writing web services?* If the developers want to use a less portable, more brittle, remote method invocation system, they would be better off using a stable technology such as Java RMI or CORBA.

If Alpine is not adopted, then either the design was flawed, or it did not appeal to a sufficiently large developer community to survive. At present it seems the design problem is the most pressing one. We have argued that JAX-RPC is the wrong API for working with SOAP in Java. If an XML-centric design were to prove equally unworkable, this might well mean that the promised flexibility of XML messaging infrastructures is not easily accessible to languages of the "Java generation" (in which we include C# and VB.NET), all of which share a similar static type system and object model. Should these languages not prove flexible enough to exploit the full potential of XML, then it may be that the promise of XML messaging systems, both REST and SOAP, will only be realised by the next generation of platforms, be they extensions of existing languages, such as Cω, or XML runtimes such as Apache Cocoon and NetKernel by 1060 Research [20], [21].

V. Acknowledgements

The authors would like to thank Tim Ewald and Ted Neward for their valuable feedback and corrections. One implementation of the CDDLM deployment service was created for the OGSAConfig project, which is funded by a grant from the Joint Information Systems Committee (JISC). The other was produced in HP Laboratories, integrating with the SmartFrog deployment infrastructure.

References

[1] N. Kassem, A. Vijendran, and Rajiv.Mordani, "Java API for XML Messaging (JAXM)," Sun Microsystems, Tech. Rep., 2003, http://java.sun.com/xml/downloads/jaxm.html.

[2] R. Chinnici, "Java API for XML-Based RPC (JAX-RPC)," Java Community Process, Tech. Rep., 2003, http://java.sun.com/xml/downloads/jaxrpc.html.

[3] Sun Microsystems, "SOAP with Attachments API for Java (SAAJ)," Sun Microsystems, Tech. Rep., 2004, http://java.sun.com/xml/downloads/saaj.html.

[4] V. Apparao *et al.*, "Document Object Model (DOM)," W3C, Tech. Rep., 1998, http://www.w3.org/DOM/.

[5] Sun Microsystems, "Java Remote Method Invocation - Distributed Computing for Java," Sun Microsystems, Tech. Rep., 1997, http://java.sun.com/products/jdk/rmi/reference/whitepapers/javarmi.html.

[6] Axis, *Apache Axis*, 2001, http://xml.apache.org/axis/.

[7] S. Loughran, "A SOAP API for Deployment," online, 2004. [Online]. Available: http://forge.gridforum.org/projects/cddlm-wg/document/Deployment_API_-_Draft_0/en/1

[8] D. C. Fallside and P. Walmsley, "XML Schema 1.0," W3C, Tech. Rep., 2004, http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/.

[9] E. Christensen *et al.*, "Web Services Description Language (WSDL) 1.1," W3C, Tech. Rep., 2002, http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[10] D. Box *et al.*, "SOAP version 1.1," W3C, Tech. Rep., 2000, http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[11] T. Neward, *Effective Enterprise Java*. Addison-Wesley, 2004.

[12] D. Winer, "XML-RPC," 2000, http://www.xmlrpc.com/spec.

[13] D. Box, *Essential COM*. Addison-Wesley, 1997.

[14] M. Henning and S. Vinoski, *Advanced CORBA(R) Programming with C++*. Addison-Wesley, 1999.

[15] Sun Microsystems, "Java 2 Platform, Enterprise Edition Specification," Sun Microsystems, Tech. Rep., 2003, http://java.sun.com/j2ee/.

[16] N. Kassem, A. Vijendran, and Rajiv.Mordani, "Java API for XML Binding (JAXB)," Java Community Process, Tech. Rep., 2004, http://java.sun.com/xml/downloads/jaxrpc.html.

[17] R. Chinnici, "Web services description language (WSDL) version 2.0 part 1: Core language," W3C, Tech. Rep., 2004, http://www.w3.org/TR/2004/WD-wsdl20-20040803.

[18] E. R. Harold, "What's Wrong with XML APIs (and how to fix them)," 2002, http://www.cafeconleche.org/XOM/whatswrong/.

[19] K. andeith Ballinger, "Basic Profile," WS-i, Tech. Rep., 2004, http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html.

[20] L. Cardelli, "Transitions in Programming Models," online presentation, 2003. [Online]. Available: http://research.microsoft.com/Users/luca/Slides/2003-11-13%20Transitions%20in%20Programming%20Models%20(Lisbon).pdf

[21] P. Rodgers, "Introducing NetKernel," online, 2005, http://www.xml.com/pub/a/2005/04/27/netkernel.html.