The Soapmill Report

for version 0.15.1

Eric Kow

30 April 2004

1 Introduction

Software components should do exactly three things: receive, process and publish XML documents. If you believe this world view, you will find that building complex applications out of these components is not very hard. All we have to do is assure that documents that are published by one component end up getting received by another, and this is precisely what the Soapmill attempts to accomplish.

This report will provide you with a basic understanding of how the Soapmill works. I will elaborate on why we reject remote procedural call as a mechanism for software reuse and instead adopt a document-centric approach. Then, I will provide details on the Soapmill exchange format (using the SOAP protocol), and present a software architecture that makes use of this format.

2 RPC is difficult

Remote Procedural Call (RPC) mechanisms complicate the problem of software interoperability by allowing components to have multiple entry points. A hotel reservation service, for example, can have procedures for "inquire availability", "check prices", and "book room". This potential multitude of entry points forces other components to determine which entry points to use and when to use them, which becomes especially difficult when components have similar but subtly different interfaces ("inquire availability" versus "check availability").

One might try to alleviate the difficulties of RPC by encouraging the use of standard interfaces. We might declare in our previous example that all proper hotel reservation services must implement "inquire availability". In fact, this is the approach applied through mechanisms such as the Common Object Request Broker Architecture (CORBA) which even go so far as to formalise matters with interface description languages like the IDL [4] and the Web Services Description Language (WSDL) [5]. This specification approach has been used to some success, but we nevertheless feel that is possible to do far better than formal interfaces.

The main room for improvement lies in the difficulty of designing good interfaces. Good interface design requires at the same time, a strong grasp of the domain, and enough programming experience to understand how that domain might be split into set of distinct procedures, each with their own carefully designed roles, inputs and outputs. Furthermore, they require the designer to have a broad enough view on the domain to design an interface that is applicable by other people and other programs trying to accomplish similar goals. What would work better is an approach which does not demand such a broad set of skills from designers.

3 XML content languages

The time we spend developing interfaces in our individual corners of the world could be far better spent on developing special-purpose **content languages** such as SVG for vector graphics, OWL

for ontologies, or MPEG-7 for describing multimedia content. Rather than creating applications which expose a set of procedures to the world, we should create components which accept and treat documents in a domain language. Doing so allows us to chop away entry points to our components, leaving only a single point which accepts incoming documents.

But now we have substituted the time investment of designing good interfaces with one of designing good languages, so apparently, nothing has changed! On the contrary. One advantage of content languages is that they allow greater participation from people with a broad, but not necessarily deeper view of our applications. These people know roughly what to demand of several components at the same time. For example, the MIAMM project makes use of multi-modal-fusion component, an action planner, a visual-haptic interface; yet these components all speak the same language. Working with content languages essentially permits us to design the "interface" for many different applications at the same time.

On the other end of the participatory spectrum, content languages can increase the input from domain experts such as linguists and physicists. Where designing a procedural interface for a parser may prove daunting to the average syntactician; asking syntacticians to describe their favourite formalism in a language is far more likely to succeed. In fact, these experts already *are* designing languages, such as TAGML as as a storage and interchange format for Tree Adjoining Grammar, so it would be wise to profit from these efforts and not duplicate them in idle interface design.

To embrace this notion requires a certain faith in standardisation. One potential danger is that the language design process could be bogged down in committees, or in trying to be everything for everyone, but this is a fear which would equally apply to the process of interface design. Besides, the advent of the XML metastandard makes standardisation much easier: there is nothing to stop an individual from proposing a standard "EricML" to the community and finding it adopted, modified or rejected depending on its strengths and weaknesses.

The push for content languages is an attempt to introduce a new layer of abstraction over software, so that components are not defined by the tasks they perform, but by the documents they understand. This is not to say that we abandon interfaces or the software design process altogether. As long as your software understands my MathML, SVG, etc documents, how your design its innards, whatever interfaces you choose to adopt is strictly your business. Software design and software interoperability should be treated as two separate problems: interfaces are for programs and languages are for components.

4 Exchange format

We earlier insisted on the development and use of standard content languages. Along the same principles of adopting standards, we propose the SOAP protocol (a W3C recommendation) as a means of exchanging documents. SOAP consists of an envelope containing an optional header followed by a body. The header is composed of any number of **header blocks** which contain metadata for specific contexts or purposes. The body contains the actual content. The SOAP specification [6] actually contains more (such as an RPC mechanism which we conveniently ignore), but we feel that its appropriate to take this extremely simple view of the protocol.

4.1 The Soapmill format

The Soapmill format imposes two restrictions on the SOAP protocol. We require that the SOAP Body contain exactly one element, a document in some content language such as MathML or RDF. We also require the use of a header and that this header contain a Ludicrously Simple Header Block (Lush, defined below). Soapmill messages are one-way in nature.

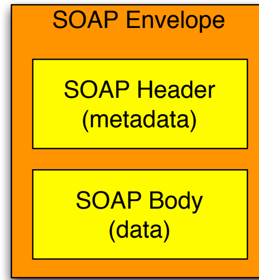

Figure 1: SOAP at a glance

4.1.1 Lush - Ludicrously Simple Header

The Lush block is a single element with several attributes:

1. `type` identifies what kind of information the message contains. An elegant use for this attribute is a URI that builds off the content language's namespace URI.
2. `id` is the unique identifier for the message.
3. `ref` (optional) refers to a previous message. This is useful for maintaining a notion of response.

A Lush Block also incorporates the `actor` and `encodingStyle` from the SOAP specification. These attributes are defined in the SOAP 1.2 specification [6]:

1. `actor` is the component that first generated this message.
2. `encodingStyle` is normally used to specify whether the document will use SOAP-defined datatypes for primitives (such as int, char, arrays) or whether it will use some other form of XML. It would likely be the latter, but there might be some simple cases, where it would be silly to invent a content language where there are perfectly good primitives lying around.

4.2 Soapmill Message Example

The following example shows what a Soapmill message would look like:

```
<soap:Envelope xmlns:soap="http://www.w3.org/2002/06/soap-envelope">
  <soap:Header>
  <lush:transaction xmlns:lush="urn:soapical:lush"
        id="EquationSolver_1"
        ref="EquationGenerator_3"
        type="http://www.w3.org/1998/Math/MathML/answer"

        soap:actor="SomeAgent.3@152.244.1.1"
        soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding"
        soap:mustUnderstand="false"/>
  </soap:Header>

  <soap:Body>
    <!--
```

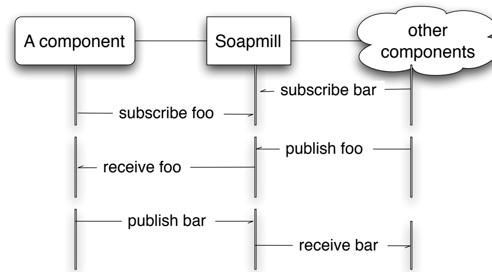

Figure 2: interacting with other components through the blackboard

```
    Your favourite XML content language.
    Here, for example, we include a block of MathML
    -->
    <mathml:math xmlns:mathml="http://www.w3.org/1998/Math/MathML">
      <mathml:mrow>
        <mathml:mi>f(1)</mathml:mi>
        <mathml:mo>=</mathml:mo>
        <mathml:mn>1</mathml:mn>
      </mathml:mrow>
    </mathml:math>
  </soap:Body>
</soap:Envelope>
```

5 Blackboard architecture

Now that we have a common format for exchanging documents, we can propose a way of using it to construct complex applications. We adopt a blackboard architecture, similar to the one used by the MULTIPLATFORM system originally developed for the SmartKom project [1].

This architecture should be familiar to anybody who has programmed in a GUI toolkit such as Swing. A graphical user interface widget such as a button on the screen has no idea what software it would interact with. To achieve flexibility, these toolkits simply ask programmers to “listen” to a number of possible “events” (“A mouse button was pressed”). This provides encapsulation in both directions: the GUI does not concern itself with what happens after it generates the event, and the programmer does not concern himself with how the event got there in the first place.

We can implement this architecture for the Soapmill as well (see figure 2). Components publish documents which are marked with message type indicators via the `type` attribute discussed in section 4.1.1. If other components have subscribed to these types, they will receive these messages. This indirectness forces components to be relatively flexible by virtue of knowing less about their surrounding environment. Rather than individually adapting each component to its peers we simply require that they be able to receive publish messages on the blackboard. Inserting new components into the system becomes very easy, as there is no need to rewrite the other components to recognise the newcomer. Similarly, replacing one component with another requires no modifications to the surrounding modules.

6 Further Information

There currently exists one implementation of the Soapmill. You can find out more about this (Java) implementation by reading *The Soapmill Manual* [3] and *Implementing Soapmill* [2]. The latter should also be useful if you would like to implement the Soapmill in your another language. All documents and software can be found at the Soapical site (`http://soapical.sourceforge.net`).

References

[1] Herzog, Kirchmann, Merten, Ndiaye, and Poller. MULTIPLATFORM Testbed: An Integration Platform for Multimodal Dialog Systems. In *Proceedings of the HLT-NAACL'03 Workshop on the Software Engineering and Architecture of Language Technology Systems (SEALTS)*, 2003.

[2] Eric Kow. Implementing Soapmill. `http://soapical.sourceforge.net` or `docs/technical_report` in your Soapmill distribution.

[3] Eric Kow. The Soapmill Manual. `http://soapical.sourceforge.net` or `docs/technical_report` in your Soapmill distribution.

[4] OMG. OMG IDL. `http://www.omg.org/gettingstarted/omg_idl.htm`.

[5] W3C. Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`.

[6] W3C. SOAP Version 1.2 Part 1: Messaging Framework, Jun 2002.