

Implementing Soapmill

version 0.15.1

Eric Kow

30 April 2004

Contents

1	Introduction	2
1.1	General approach	2
2	The Basics	3
2.1	Sending messages	3
2.2	Receiving Messages	6
3	Extra Features	7
3.1	Handling responses	7
3.2	Visualisation	7
3.3	Control	8
4	The Java Implementation	10
4.1	Methodology	10
4.2	Tools used	11
4.3	Future work	12
5	Future Work	13
5.1	Subscription lists	13
5.2	Soapswitch	14
5.3	Extended Soapswitch	14
5.4	Soapcaster	15
A	General Glossary	17

Chapter 1

Introduction

This document is meant as a guide to implementing the Soapmill in your favourite language. It might also be helpful to anybody wishing to learn about, improve, or rewrite the Java implementation. Anybody else (project managers, soapmill users, etc) should read some combination of the *The Soapmill Report* [2] and *The Soapmill Manual* [2].

In chapters 2 and 3, I shall describe features to implement and some problems to solve. These chapters will also briefly mention how the Java implementation attacks these problem, if at all, and point you to the corresponding source files. If you are more interested in the Java implementation than in making your own Soapmill, you might also consider chapter 4 which includes information specifically about the Java implementation. In any case, this guide ends with future work in chapter 5, where you are invited to implement some potentially useful features and tools.

1.1 General approach

The only requirement of Soapmill software is that their communications with the outside world consist of document-style SOAP messages with Lush blocks (described in section ??). A cherry-picking attitude that treats all features in this document as strictly optional, as conveniences even, becomes very useful in coping with the number of problems that software interoperability engenders.

Another consideration is whether you wish to create a Soapmill implementation at all, that is, something which is to be used by other programmers in your favourite language(s) or merely a single piece of Soapmill-compatible software. Depending on looming project deadlines, it might be useful to first implement the Soapmill strictly for the needs of your software, and only attempt to generalise this implementation when the opportunity arises and you have the time to grapple with the problems that lie in making software usable by everybody.

You might be able to escape Soapmill implementation altogether. For example, your software might be simple enough that it only needs to receive messages, and for that matter, only one type of message. Be that the case, you could create a Java agent which responds to incoming messages by spawning a process and feeding XML through that process's STDIN. This alternative keeps things simple, but only if your software can function with a single-message lifespan. If it needs to stay in memory over the course of several messages, the advantages of a full-fledged Soapmill implementation become more attractive.

Chapter 2

The Basics

2.1 Sending messages

As a minimum, your Soapmill implementation should send and/or receive SOAP messages with a Lush block as described in *The Soapmill Report*. I repeat the following example of a Soapmill message for quick reference:

```
<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>
    <!--
    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>
```

It is recommended to reuse any SOAP implementations which may already exist for your language, and which provides at support for document-style services. Otherwise, you could combine HTTP client software with an XML implementation.

See also

```
etc/schema/lush.xsd
src/soapical/soapmill/SoapmillMessage.java
src/soapical/soapmill/engine/AxisHandler.java
```

2.1.1 Unique Agent Names (Lush)

Other Soapmill agents could conceivably require the ability to distinguish between different message senders. In order to help them out, you should make sure that the Lush `actor` attribute is unique to each agent. One reasonably useful tactic is to attach the IP address on which the agent is running, so that (for example) messages from `SomeAgent` on a particular machine have `actor` equal to “`SomeAgent@152.200.0.4`”. But this only goes as far as there being one only instance of the same agent name on a single IP. What you should really do for each machine is to assign a counter to each name. The first `SomeAgent` that comes along is “`SomeAgent.1@152.200.0.4`” and next “`SomeAgent.2@152.200.0.4`”, and so on. Ideally, your Soapmill implementation should even share its counters with other implementations on the same machine by consulting and updating a single file (I propose `/tmp/soapmill-names`).

See also

```
src/soapical/soapmill/SoapmillAgent.java
```

Notes

1. The Java implementation does not yet implement this, but it is on the TODO list.
2. Nobody seems to make use of the sender attribute yet, but neither this nor the Java implementation’s failure to implement this are any excuse!

2.1.2 Unique Message IDs (Lush)

Message IDs should also be generated such that no two messages have the same ID. A useful trick is to rely on the uniqueness of agent names and merely append a message counter for each agent. For example, the agent “`SomeAgent.3@152.61.2003.4`” might send messages with the IDs like “`SomeAgent.3@152.61.2003.4.42`”.

2.1.3 Knowing your Recipients

It may become extremely inconvenient as systems included more and more components to configure each component so that it knows where to send its messages. We are faced with a three-way tradeoff among implementation ease, simplicity in use, and efficiency. The easiest way (for programmers) to configure sending is manually, for example, by using a text file which tells your software where each recipient should be. This is the default, recommended approach, though it gives rise to this original problem of hand-configuring every module.

More conveniently for the user, we could also devise some scheme for Soapmill implementations to tell each other where they live. This introduces an undesirable amount of complexity because the whole point of Soapmill is to achieve widespread adoption by sticking with ideas that are easy to implement and explain. Still the sophistication might come in handy, especially if you want the ability to add or remove components during the system’s run time. To find out more, see future work section 5.4. Alternately, we might be able to provide an acceptable level of simplicity for both user and programmer by pushing this functionality into an analogue of network hubs. See future work section 5.2 if you are interested in doing this.

Notes

1. The Java implementation uses manual configuration through a text file. We chose implementation ease (and efficiency) over simplicity in use.
2. We are able to hide from the issue of convenience because most of our modules run as threads on the same machine and communicate through a local event handler. If the Soapmill idea spreads far enough, this approach alone will no longer be acceptable.

2.1.4 Avoiding Congestion

Soapmill places the primary burden of filtering unwanted messages on the receiver. In other words, receivers are expected to accept and silently drop any messages which are not of interest. This adds a measure of robustness and flexibility to Soapmill architectures. For example, an architecture composed of very simple and lazy message senders would have everybody blindly broadcasting messages to everyone else, and things should still work.

The question now is how to avoid abusing this division of labour. Just because we can theoretically send all our messages to everybody does not mean that we necessarily *should*, because that could create a lot of unwanted network traffic and a lot of filtering work for receivers. This is a refinement of the problem discussed in section 2.1.3. Earlier we dealt simply with knowing who your recipients are, now we would like to know which of your recipients are interested in what you have to say.

The easy times are over. While manual configuration would have been tolerated for the simpler case (discovery), extending this manual configuration is not acceptable. Experience with the Java implementation shows that you cannot expect the user to accept the glaring redundancy between configuring a receiver to subscribe to messages, and then turning around and hand-configuring senders so that they only send messages to the right place. This creates enough confusion and extra work for the user to spark a minor rebellion.

This leaves us with essentially two options. Diligent implementors can flesh out the ideas in section 5.1 for creating a mechanism that Soapmill implementations can use tell each other what message types they are subscribed to. Less diligent (read lazy) implementors can try ignoring the problem and hoping that it goes away by itself. This problem is possibly not serious enough to be worth solving and that we should be focused on more interesting matters. It also be possible to solve satisfactorily enough by creative use of the Soapswitch, as described in section 5.2.

Notes

1. The Java implementation takes the cowardly route (hoping that somebody else solves the problem). It sends all messages to all recipients it knows about, counting on the small number of recipients it has.
2. If your software sends out messages which are heard by only one receiver, you could sidestep the issue by only sending to a single recipient, but then you lose the flexibility of events for your module.

2.1.5 Efficient Sending

If your software does not already rely on DOM for dealing with XML, you might benefit greatly from a SOAP implementation that avoids DOM-like representations. These representations can cost a lot of time because each node in the tree requires an object to be constructed, and after the entire tree is built, it has to be reserialised into text for sending on the wire.

It could be a much better approach, considering how predictable Soapmill messages are to simply construct the message by concatenating pieces of template to the raw XML you wish to

send, but if you wish to do this, you should make sure you do not run into any trouble with character encoding issues and the prolog of your outgoing XML.

2.2 Receiving Messages

Most Soapmill implementations should be able to receive and process messages as described in section 2.1. As above, it is recommended to build this on top of SOAP implementation with good support for document-style services.

See also

```
src/soapical/soapmill/Delegator.java
```

2.2.1 Statefulness/Persistence

Does your software require a long-term memory? If your software does not require the ability to remember its state across multiple incoming messages, implementing the Soapmill could be a great deal simpler because you are able to make direct use of simple SOAP implementations that spawn a new process for every incoming message.

The most direct option if you need a long term memory is to have the HTTP server and your Soapmill implementation running in the same process. You can do this by either making your Soapmill implementation an HTTP server by itself or making using any plugin mechanism a pre-existing server might have.

Otherwise, you could run your software as a separate process and tie it to the SOAP implementation using a interprocess communication mechanism such as Unix-domain sockets.

Notes

1. In this section, we use HTTP as the primary example. Substitute SMTP or whatever transport you see fit whenever you see HTTP.
2. Naturally, the most sensible thing would be to understand how the SOAP implementation that you use relates to the server before plunging into any design decisions.
3. In the Java implementation, each agent runs as a separate thread in the same process. This has one disadvantage that individual agents can not be replaced without shutting down everyone else on the same virtual machine.

2.2.2 Efficient receiving

If at all possible, you should consider what steps you can take to ensure that your Soapmill implementation processes incoming messages as efficiently as possible. SAX-style implementations may be considerably faster than DOM or XML-binding implementations because they require less processing on uninteresting parts of the message. The best way to find out is to create and take advantage of benchmarking tools as described in section ??.

Chapter 3

Extra Features

There are a number of features that you could choose to implement in you have a large enough number of agents using your Soapmill implementation to merit it:

3.1 Handling responses

Soapmill messages have the option to refer to previous messages using the `ref` attribute. One feature that your Soapmill implementation could provide is the ability to treat messages as responses. You could keep a table of outgoing message IDs. Whenever an incoming message comes in that refers to one of these IDs, your implementation could perform some kind of special behaviour, such as awakening a sleeping thread.

See also

```
src/soapical/soapmill/engine/Publisher.java
```

Notes

1. Messages may have multiple responses.
2. The Java implementation provides a special function `publishAndAwait` that sends a message and blocks until a response to that message is received.

3.2 Visualisation

The Soapical Soapmeter is a useful tool for debugging the Soapmill and Soapmill agents. It offers basic TCP monitoring functionality for viewing SOAP messages, which means that you can get some rudimentary debugging without modifying your Soapmill. If you want to take advantage of Soapmeter's special features; however, you will have to do a little extra work.

3.2.1 Send

Before you publish a message, you should publish a message of type: `urn:soapical:monitor`

```
<monitor>  
  <send sender="BlahBlahBlahAgent" receiver="OtherBlah"/>  
</monitor>
```

3.2.2 Done

After your agent has finished processing a message, it should publish again a message also of type `urn:soapical:monitor`. This message should list out what the agent published while processing this message. This list of published messages is not used by the Soapmeter, but it will be useful for implementing SoapmillUnit.

```
<monitor>
  <done agent="BlahBlahBlahAgent">
    <published ref="message_1"/>
    <published ref="message_2"/>
    <published ref="message_3"/>
  </done>
</monitor>
```

3.2.3 Error

Finally, you can also publish `urn:soapical:monitor` messages to indicate that an agent encountered some kind of error

```
<monitor>
  <error agent="BlahBlahBlahAgent"/>
</monitor>
```

See also

```
etc/schema/soapmill_monitor.xsd
src/soapical/soapmill/engine/Publisher.java
```

Notes

1. Be careful when publishing monitor messages not to cause any infinite loops, where for example, publishing a monitor messages causes a monitor message to be published, ad infinitum.
2. See the Soapical site for my SoapmillUnit idea.
3. We should modify this schema so that message ids are included in the publish, done, and error tags

3.3 Control

Another useful feature for your Soapmill to implement is the ability to control agents from the exterior. Your Soapmill could treat messages of the type `urn:soapical:soapmill:control` specially. If you receive a message of the form

```
<control>
  <reset agent="SomeAgent" hard="true"/>
  <reset agent="SomeOtherAgent" hard="false"/>
  <reset agent="YetAnotherAgent" hard="true"/>
</control>
```

The agents “SomeAgent” and “YetAnotherAgent” should be hard-reset and the agent “SomeOtherAgent” should be soft-reset.

```
<control>  
  <reset hard="true"/>  
</control>
```

See also

```
etc/schema/soapmill_control.xsd  
src/soapical/soapmill/engine/ControlAgent.java
```

Notes

1. Whether or not you implement a distinction between hard/soft resets is up to you.
2. The only control functional which is currently available is the ability to reset agents.
3. The Java implementation uses an optional control agent which uses reflection to manipulate other agents on the virtual machine.

Chapter 4

The Java Implementation

This chapter contains details specific to the Java implementation.

4.1 Methodology

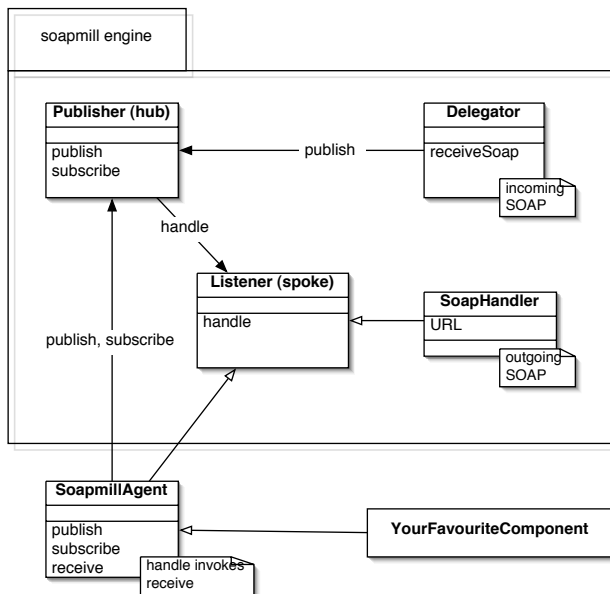


Figure 4.1: Java implementation of Soapmill

The Java implementations allows different pieces of software, called **agents**, to run together on the same virtual machine, but to communicate with the outside world using Soapmill messages. This implementation is based on a hub and spoke architecture (see figure 4.1 and 4.2), where a Soapmill object serves as a hub and each agent is a spoke. Following the blackboard architecture described in section ?? every message published by one spoke is received by every other spoke as long as the receiving spoke has subscribed to the published message type. This approach allows agents to avoid the overhead of parsing/generating SOAP when they are on the same machine.

What about the outside world? This is done by exploiting the hub and spoke architecture. We add an incoming spoke that receives SOAP messages and publishes them to the hub, and some number of outgoing spokes that subscribe to all messages from the hub and forwards them to a remote URLs. Naturally enough, this communication takes place using Soapmill messages as described in section ?. The hub and spoke also allows us to maintain and improve our

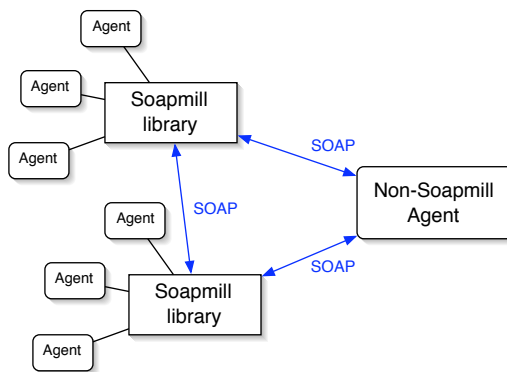


Figure 4.2: local agents need not use SOAP to communicate between themselves

handling of SOAP. Should we attempt to switch to a better SOAP implementation or otherwise improve efficiency, we can merely substitute the current SOAP spoke for something better.

4.2 Tools used

4.2.1 SOAP

Apache Axis

Soapmill currently uses Apache Axis, along with the standard SAAJ APIs and the Soapical Soapkit to handle the generation and parsing of SOAP. This yields performances issues which could be resolved by creating a customised SOAP implementation in the future. For example, the implementation could use simple Strings and templates to generate messages.

Soapkit

The Soapkit makes it easier to deal with XML and SOAP. See <http://soapical.sourceforge.net/soapkit>.

4.2.2 XML

Castor

Castor was used to create the Java objects for Lush (see subsection ??).

Xerces - to be removed?

One unfortunate drawback is that Castor relies on Xerces, which apparently causes Axis to fail its self-tests. So far, this has not caused any serious problems, but it would be useful later on to experiment with a version of Castor which allows usage of JAXP.

Xalan - to be removed?

Xalan was included because many agents use it and the version included with Java 1.4.1 is not sufficiently recent. When later versions of Java come out and integrate Xalan, we should remove this.

4.2.3 Miscellaneous

Unannoy

See <http://unannoy.sourceforge.net>

Shell scripts

Soapmill, while largely implemented in Java, also relies on a heavy use of portable Unix shell scripts. Scripts are rather problematic because it is harder to maintain portability across shell implementations and requires Windows users to install the Cygwin environment or similar software. It would be useful to wean the Soapmill off these scripts; ideally, a programmer should be able to run Soapmill-based agents by double-clicking on some icon.

4.3 Future work

4.3.1 Efficient SOAP

The current SOAP implementation (Axis 1.1) builds a DOM-like structure (SAAJ) for SOAP messages, which is very costly since it receives outgoing XML as a raw stream. It will be useful to implement more efficient SOAP as described in sections 2.1.5 and 2.2.2. The following tool may prove useful:

DocSOAP

<http://www.commerceone.com/developers/docsoapjdk>

A high-performance SOAP implementation of specific interest to document-centric web services such as those provided by the Soapmill.

Jamsterdam SPAX

<http://www.jamsterdam.com/dist/spax>

One way for improving performance would be to write the SOAP reader with Jamsterdam SPAX. I shall revisit this possibility later.

Chapter 5

Future Work

As described so far Soapmill implementations are very simple to write and deploy. However, when building a system that has, say more than 5 components, configuring each of these components to communicate with each other can become inconvenient. This chapter proposes several complementary solutions, some possible tools we can use to improve the user's convenience and systems' efficiency. None of these solutions have yet been implemented.

Before we plunge into these, let me insist again the proposed sophistication, like Soapmill features are *strictly optional*. For example, a Soapmill implementation might not opt to implement subscription lists (section 5.1). It could send directly to an interested listener and some senders (or the Soapswitch) can be hand-configured to send messages directly to it. Likewise, the Soapcaster allows us to bypass any use of a central hub, but software should have the option of passing through on or communicating directly with other as need be. No matter how many of the following features are implemented, we should leave as much room as possible for the most primitive of Soapmills to function.

5.1 Subscription lists

Task

- Avoiding congestion (section 2.1.4)

Description

Avoiding congestion requires some cooperation from subscribers, namely that they have to announce somewhere what messages they subscribe to. Soapmill implementations can add this functionality as an optional feature (like visualisation and control, sections 3.2 and 3.3) by exploiting the normal Soapmill message type mechanism. They could send simple messages that draw a many-to-many mapping from message types to recipients:

```
<soapmill>
  <subscribed type="urn:foo:sometype"
    receiver="http://localhost:8080/SometypeParser"/>
  <subscribed type="urn:foo:sometype"
    receiver="http://localhost:8080/OthertypeParser"/>
  <subscribed type="urn:bar:othertype"
    receiver="http://localhost:8080/OthertypeParser"/>
</soapmill>
```

Notes

1. Type `urn:soapical:soapmill:subscribers` is reserved for this purpose.

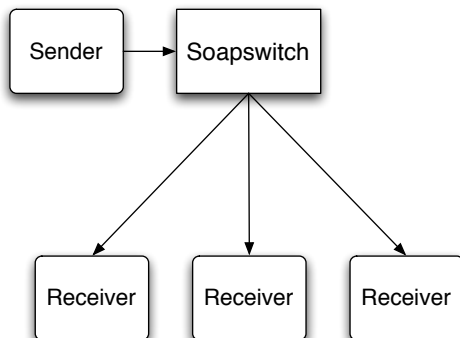


Figure 5.1: Soapswitch simplifies sending to multiple recipients

2. These messages are not yet formally defined in a schema.
3. It might be interesting to merge this with the Soapmill control mechanisms to prevent undue schema proliferation.

Current status

Unimplemented. Not sure if worth implementing.

5.2 Soapswitch

Task

- Knowing your recipients (section 2.1.3)

Description

The Soapswitch would be an easy to deploy, standalone application that forwards Soapmill messages to a predefined list of recipients (for example, defined through a text file). It should be possible to implement this tool very simply and efficiently. For example, an HTTP implementation could be a simple proxy server which forwards all messages to a group of recipients. This service would be most useful if it were accompanied with test data that allows users to determine if the increased overhead is acceptable for the applications they build. For the analogically minded, the Soapswitch would be similar to a network hub.

Current Status

Not implemented. Looking for implementors or third-party tools that do a similar job.

5.3 Extended Soapswitch

Task

- Knowing your recipients (section 2.1.3)
- Avoiding congestion (section 2.1.4)

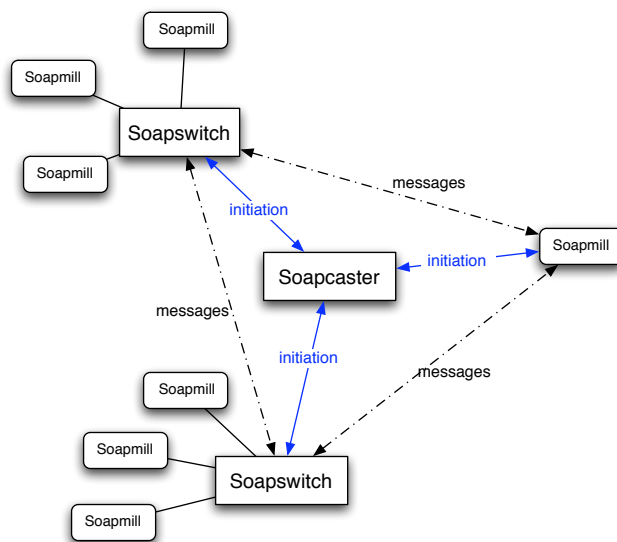


Figure 5.2: Combining Soapcaster with Soapswitch

Description

Section 5.1 proposed a mechanism for subscribers to announce which messages they are interested in. How do we exploit this information? We could extend the Soapswitch (5.2) to itself be a Soapmill subscriber. Any subscription lists it receives are used for configuration of the Soapswitch, and any other message is forwarded to the right place.

If it turns out that a Soapswitch is presenting a bottleneck, we could further extend the concept so that Soapswitches can themselves be connected to Soapswitches. To do this, the Soapswitch implementation will to send a combined subscription list to its switch. For the analogically minded, the extended Soapswitch would be similar to a network switch.

Current status

Unimplemented. Not sure if worth implementing.

5.4 Soapcaster

Task

- Knowing your recipients (section 2.1.3)

Description

The Soapcaster presents an alternative or a complement to the Soapswitch (sections 5.2). It provides a means for a group of web services to communicate among themselves. When a new service comes online, it initiates contact with the Soapcaster. Periodically, the Soapcaster updates every service it knows, with the status and location of these services. Any communication between services does *not* pass through the Soapcaster (as it would in the Soapswitch); it only serves to help services identify each other. This decentralised nature makes the entire more robust. If the Soapcaster shuts down, is removed or otherwise becomes unavailable, all the other servers can still function normally.

This decentralisation gives the Soapcaster a distinct advantage over the Soapswitch both in terms of efficiency and robustness, but it comes at a price that services have to be adapted so that they are also Soapcaster clients.

Current status

There is an experimental Soapcaster on the Soapical Sourceforge project homepage (downloadable by CVS only) which is written for one sample application but in a general enough fashion that we may eventually adapt it to the Soapmill. To do this work, we will need to answer the following questions:

1. What work is necessary in generalising the tool?
2. Will it be useful to upgrade its SOAP implementation to take advantage of the recently released standard APIs (SAAJ)?

Appendix A

General Glossary

These definitions will be highly informal and incomplete for the time being. I attempt to provide the origin for each word, to avoid giving the false impression, for example, that words I invent would be recognised by the rest of the community.

agent Any autonomous piece of software in a complex application. The idea is for agents to be easily transferable between different applications.

Note: an agent is different from a server. A server usually refers to an individual computer; it can host an arbitrary number of agents.

Origin: AI literature (abused?)

facilitator Supporting tools for building complex applications. Where agents are components in any architecture, facilitators are metacomponents and need not function as agents.

If you wanted to be consistent about it, you could consider the Soapmill to be a facilitator, but for clarity in this document we do not do this.

Origin: Similar agents work (Open Agent Architecture). Note: Soapmill's usage of this word differs from OAA in that there could be different facilitators for different tasks.

MIAMM Multidimensional Information Access using Multiple Modalities. The project in which Soapmill was originally conceived.

Origin: MIAMM project

MMIL The Multimodal Interface Language (MMIL) is the central representation format in the MIAMM project. See MIAMM Deliverable 6.3 [3] for more details.

Origin: MIAMM project

MPEG-7 The MPEG-7 standard has a portion that describes word lattices. We have selected the MPEG-7 audio component, and further pared down anything which does not appear related to word lattices. This reduced schema is available on the MIAMM website (<http://www.miamm.org>). Note: MIAMM documents will use the words **wordgraph** and **word lattice** interchangeably.

RPC “Remote Procedural Call. A protocol which allows a program running on one host to cause code to be executed on another host without the programmer needing to explicitly code for this. RPC is an easy and popular paradigm for implementing the client-server model of distributed computing . An RPC is initiated by the caller (client) sending request message to a remote system (the server) to execute a certain procedure using arguments supplied. A result message is returned to the caller. There are many variations and subtleties in various implementations, resulting in a variety of different (incompatible) RPC protocols.” [1]

Origin: computer programming community.

server In Soapmill terms, this would correspond to a SOAP node. A server can host an arbitrary number of agents.

SOAP A minimal set of conventions for sending XML data, the default being over HTTP.

Origin: SOAP protocol authors (DevelopMentor, Microsoft and UserLand Software)

Soapmill A library for constructing complex applications out of reusable parts. This library is used by a single agent to communicate with other agents, but emphasis placed on each agent not needing to know the details about the others.

Origin: Eric

Bibliography

- [1] Free On-Line Dictionary of Computing. <http://foldoc.doc.ic.ac.uk/foldoc/>.
- [2] Eric Kow. The Soapmill Report. <http://soapical.sourceforge.net> or `docs/technical_report` in your Soapmill distribution.
- [3] Laurent Romary (LORIA). MIAMM Deliverable D6.3: MMIL Specification.