Volume Title
ASP Conference Series, Vol. **Volume Number**
Author

Application Interoperability with SAMP

M. Fitzpatrick[1], O. Laurino[2], L. Paioro[3] and M. B. Taylor[4]

[1]*National Optical Astronomy Observatory, Tucson AZ, U.S.A.*

[2]*Harvard-Smithsonian Center for Astrophysics, 60 Garden Street, Cambridge, MA 02138, U.S.A.*

[3]*National Institute for Astrophysics, IASF, Milano, Italy*

[4]*H. H. Wills Physics Laboratory, University of Bristol, U.K.*

Abstract. The Simple Applications Messaging Protocol (SAMP) is a Virtual Observatory (VO) specification that enables astronomy software tools to exchange control information and data, allowing desktop applications to work as an integrated suite of tools rather than requiring complex functionality to be (redundantly) built into tools individually. In addition, SAMP allows new workflows to be created for the science user that leverages the advantages of each tool (e.g. visualization of tables or images, analysis, etc), greatly reducing the time needed to switch between applications and tasks. We present here a short introduction to the protocol itself, a survey of some toolkits for application authors who wish to introduce SAMP functionality into their tools, and some examples of real-world usage.

1. Introduction

The Simple Applications Messaging Protocol provides platform-independent messaging between tools based either on the desktop or the browser. The messaging architecture is based on a free-standing *hub* process that provides message brokering to external clients, providing the illusion of direct client-client interaction with the convenience of a single communication point. Messaging is built around the publish/subscribe model in which each client flags those message types (MTypes in the SAMP terminology), if any, it is willing to receive. SAMP is defined by the SAMP standard (Taylor et al. 2012), and the design principles are discussed further in Taylor et al. (2011).

Section 2 below lists a number of language-specific libraries and toolkits available that can help developers to work with SAMP and incorporate SAMP usage into their applications. Section 3 gives some diverse examples of how these tools can be used to deliver improved science workflows.

2. Toolkits and Implementations

A number of toolkits and libraries for use with SAMP are listed at `http://www.ivoa.net/samp`; this section describes some of them.

2.1. JSAMP

JSAMP is a hub implementation, toolkit and client library written in Java. As well as a basic interface to the SAMP Hub and Client APIs, JSAMP provides easy-to-use GUI components for integrating SAMP use into interactive Java applications. JSAMP also incorporates a number of diagnostic tools, including extensive message logging capabilities and a graphical hub view that shows the details of currently registered clients and recently transmitted messages.

2.2. SAMPy

SAMPy is a Python toolkit and hub implementation. SAMPy will be part of astropy (Tollerud et al. 2012). Astropy is a common effort to develop a single Python core package for astronomy, involving about 100 developers from around the world, and is available from PyPI.[1].

To start SAMPy's hub implementation it is sufficient to start the sampy executable, installed with the main distribution. As with other libraries, registering a client requires the instantiation of the client itself, its connection to the hub, and the binding of a Python function to specific MTypes. The function is used as a callback when a message with a bound MType is sent to the client. SAMPy also offers means to discover clients connected to the hub and send messages to them.

2.3. Libsamp

Libsamp is a library within the *VOClient* package (in development) that provides a C-language interface to enable applications to send and receive SAMP messages. The API is designed to simplify and hide the details of the SAMP protocol from the application writer, providing standard methods to initialize the interface, declare metadata, post message callbacks, send specific message MTypes, and startup/shutdown messaging. Details of the hub discovery and registration, as well as handling of specific messaging patterns, are handled internally and are also fully accessible through low-level procedures. These low-level procedures similarly allow application developers fine-grained control over the formatting of outgoing messages or parsing of return values. Because the interface is implemented in C, bindings for many other languages can be easily generated automatically using SWIG,[2] or custom interfaces can be hand-generated to provide a more language-specific interface (e.g., one that uses idioms of the language as in a *Pythonic* interface, or a binding for languages not supported by SWIG such as *SPP* used in IRAF).

2.4. sampjs

Sampjs is a small JavaScript library for use by browser-based applications that performs SAMP messaging using the Web Profile. Sampjs makes it easy to add SAMP messaging capabilities to web pages by adding a few lines of JavaScript, as well as allowing the possibility of fully SAMP-integrated web applications.

---

[1] `http://pypi.python.org/pypi/sampy/`

[2] `http://www.swig.org/`

3. Usage Examples

3.1. Integration of GUI tools

A common usage scenario for SAMP is integrated use of multiple interactive desktop applications specialised for different data types. SAMP's data exchange enables them to work together as a single integrated suite with the union of the capabilites of the component tools. An example workflow involving TOPCAT (a table analysis tool) and Aladin (a sky image analysis tool) might be:

1. display an image of a region of sky in Aladin
2. acquire a catalogue in Aladin with multi-band photometry corresponding to sources visible in the region
3. overplot the catalogue positions on the sky imagery
4. send the catalogue to TOPCAT using SAMP
5. plot a color-magnitude diagram in TOPCAT
6. identify a sub-population in TOPCAT from the color-magnitude plot
7. send the sub-population referencing the original catalogue back to Aladin using SAMP
8. Aladin displays the sub-population sources in a way which distinguishes them visually from the others

The SAMP send operations are typically initiated by the user simply hitting an appropriate "Send" button in the GUI. The loose semantics of the messages typically exchanged by SAMP applications mean that this workflow could work in just the same way if different image- and/or table-analysis tools were used.

3.2. SAMP as a lightweight remote procedure call protocol

Some projects have used SAMP as a lightweight protocol for remote procedure calls. The advantage of this approach is that robust off the shelf SAMP libraries can be used to build a thin layer on top of existing applications in different programming languages in order to make them communicate. Such a private interface can also be exercised by different clients than those that were targeted originally.

Iris (Doe et al. 2012), the Virtual Astronomical Observatory tool for the analysis of Spectral Energy distributions, for example, employed SAMP to make the connection between a Java application for spectral analysis (Specview, by STScI, Busko (2000)) and a Python fitting engine (Sherpa, by SAO, Doe et al. (2007)). The design is straightforward and requires the specification of methods, identified by MTypes; arguments, in the form of SAMP dictionaries; exceptions, serialized as SAMP messages and as such propagated from one programming language to the other.

While Python offers natural means for deserializing dictionaries in the form of instances, a specific library was developed in Java for (de)serializing Java interfaces as SAMP messages. This makes the implementation of a simple inter-language remote API very straightforward and lightweight.

3.3. SAMP from the command line

The *Libsamp* library was used to fully SAMP-enable the IRAF Command Language (CL) as well as to build a command-line tool (called *vosamp*) to allow scripts to send (and optionally receive) messages. In both cases, a simplified command interface further hides the details of the SAMP protocol from the user. For example, a *load* com-

mand takes as a single argument the name of a local file or a URL. The IRAF CL or *vosamp* task determine whether this file is a FITS image or a VOTable and format the appropriate message type or supply additional arguments as needed. Options exist to send directed messages to specific applications or send messages using a particular message pattern.

For the *vosamp* command-line tool, the overhead of connecting to the Hub with each command in scripts is avoided by having the task run in the background as a persistent proxy. On the first invocation the task registers with the Hub and then forks itself to run in the background while remaining connected to the messaging session. Subsequent commands from the terminal or script are sent to this background proxy via IPC for execution, allowing a script to process many commands using a single application registration. This capability means that any scripting language (e.g. Python, Perl, IDL, Bourne or C-shell, etc) that can execute a host command can send SAMP messages without requiring detailed knowledge of the protocol by the script writer. In cases where tighter integration with the language is required, bindings can be generated as needed.

3.4. SAMP from archive query web pages

Many data centers provide web-based access to their data holdings along the lines of a form which a user fills in, resulting in a web page listing one or more data products such as images, spectra or catalogues, with the expectation that users will download these files to disk and then load them into a suitable viewer application.

Using the Web Profile introduced in SAMP 1.3 and a JavaScript library like sampjs, it is very easy (10–20 lines of JavaScript) to associate a button with each such link that sends the relevant file (in fact, its URL) directly to whatever suitable SAMP-aware viewer the user happens to be running, if any. It is straightforward to arrange for such buttons to be hidden in the absence of a SAMP hub, so non-SAMP-aware users do not experience unavailable functionality as increased clutter.

References

Busko, I. 2000, in ADASS IX, edited by N. Manset, C. Veillet, & D. Crabtree, vol. 216 of ASP Conf. Ser., 79

Doe, S., Bonaventura, N., Busko, I., D'Abrusco, R., Cresitello-Dittmar, M., Ebert, R., Evans, J., Laurino, O., McDowell, J., Pevunova, O., & Refsdal, B. 2012, in ADASS XXI, edited by P. Ballester, D. Egret, & N. P. F. Lorente, vol. 461 of ASP Conf. Ser., 893

Doe, S., Nguyen, D., Stawarz, C., Refsdal, B., Siemiginowska, A., Burke, D., Evans, I., Evans, J., McDowell, J., Houck, J., & Nowak, M. 2007, in ADASS XVI, edited by R. A. Shaw, F. Hill, & D. J. Bell, vol. 376 of ASP Conf. Ser., 543

Taylor, M. B., Boch, T., Fay, J., Fitzpatrick, M., & Paioro, L. 2011, in ADASS XXI, edited by P. Ballester, D. Egret, & N. P. F. Lorente (San Francisco: ASP), vol. 461 of ASP Conf. Ser., 279

Taylor, M. B., Boch, T., Fitzpatrick, M., Allan, A., Paioro, L., Taylor, J., & Fay, J. 2012, Simple Application Messaging Protocol, Version 1.3, Tech. rep., IVOA Recommendation

Tollerud, E., Greenfield, P., & Robitaille, T. 2012, in ADASS XXII, edited by D. Friedel, M. Freemon, & R. Plante (San Francisco: ASP), vol. TBD of ASP Conf. Ser., TBD