SAMP, the Simple Applications Messaging Protocol

M. B. Taylor[a], T. Boch[b], J.Taylor[c]

[a]*H. H. Wills Physics Laboratory, University of Bristol, UK*
[b]*CDS, Observatoire Astronomique de Strasbourg, France*
[c]*Google, USA*

Abstract

SAMP, the Simple Applications Messaging Protocol, is a hub-based communication standard for the exchange of data and control between participating client applications. It has been developed within the context of the Virtual Observatory with the aim of enabling specialised data analysis tools to cooperate as a loosely integrated suite, and is now in use by many and varied desktop and web-based applications dealing with astronomical data. This paper reviews the requirements and design principles that led to SAMP's specification, provides a high-level description of the protocol, and discusses some of its common and possible future usage patterns.

Keywords: interoperability message-passing publish-subscribe

`Version: 2fee459 (2014-10-23 17:40)`

1. Introduction

Astronomical research requires complex and flexible manipulation and processing of various different types of data. Images, spectra, catalogues, time series, coverage maps and other data types need their own special handling, typically provided by specialist tools. Data sets of different types meanwhile are usually related in various ways arising from their physical origin, for instance catalogues are often derived from images and best understood in conjunction with them, and spectra and time series usually originate from specific sky positions or regions which may be represented on images and described by catalogue entries. To extract scientific meaning from the data it is usually necessary to exploit these linkages between data items as well as the internal structure of each.

The working astronomer therefore uses a selection of different software items, each specialising in a particular type of data or manipulation, for different data sets and different tasks, and has to integrate these together in a way that takes account of the relationships of the data items under consideration. This situation is not specific to the Virtual Observatory, but the wealth of heterogeneous data available from the VO, sometimes requiring the use of additional tools for data discovery and acquisition, accentuates these issues.

For batch or pipeline-type processing the required tool integration is usually, in terms of data flow, fairly straightforward: the output of one step can be fed to the input of the next as a file, stream of bytes, or some kind of parameter list, often under the control of a script of some kind.

During the exploratory or interactive phase of data analysis however, this traditional model of tool integration is less satisfactory. Within a given GUI analysis application it is usual to interact with the data using mouse and keyboard gestures to perform actions like selection or navigation with instant visual feedback, in many cases with some kind of internal linkage between different data views. But communicating such actions or their results between different tools tends to be much more cumbersome. A way can often be found to reflect a result generated by one tool in the state of another, for instance by reading sky coordinates reported by one tool and typing or pasting them into another, or saving an intermediate result from one tool to temporary storage and reloading it into another, but it can be fiddly and tedious, especially if similar actions are required repeatedly. This lack of convenience is more than just an annoyance, it can interrupt the flow of the data exploration, reduce the parameter space able to be investigated, and effectively stifle discovery of relationships present in the data.

From this point of view, a single monolithic astronomical data analysis user application providing the best available facilities for interactive presentation, manipulation and analysis of all kinds of astronomical data and their interrelationships seems an attractive prospect. In reality of course, no such one-stop analysis tool exists. The obvious practical difficulties aside, it is not even clear that deviating so far from the Unix philosophy of "Make each program do one thing well" (McIlroy et al., 1978) would be desirable.

These considerations have driven the development of

Email address: `m.b.taylor@bristol.ac.uk` (M. B. Taylor) a framework for communication between independently-developed software items, written in different languages and running in different processes. Such applications can thus be made to appear to the user as a loosely integrated suite of cooperating tools, providing facilities such as data exchange, linked views and peer-to-peer or client-server remote control. Although communication between interactive desktop tools was the original stimulus for what is now SAMP, the framework is flexible enough to support other usage patterns as well.

Two previous papers on SAMP have been presented in the ADASS conference series: Taylor et al. (2012a) briefly outlines the architecture and explains the Web Profile, and Fitzpatrick et al. (2013) lists some existing client libraries. The current paper discusses the protocol, its communication model, and its current usage in sufficient detail to understand the design decisions taken and their consequences, particularly from the point of view of the usage scenario outlined above. Section 2 outlines the evolution of SAMP from its predecessor PLASTIC, section 3 outlines some high-level design principles, section 4 presents a description of the protocol itself along with some of the thinking behind it, section 5 considers its use in practice, and section 6 concludes by reviewing the current status and possible future directions for SAMP. For the complete and definitive details of the protocol, the reader is referred to the standard document itself (Taylor et al., 2012b).

2. Context

2.1. History

In the context of the emerging Virtual Observatory in the mid-2000s, the benefit of connecting client-side tools to improve productivity when working with multiple data types became apparent. In fact this problem was not specific to the VO, but the ease with which multiple related data products of diverse types within multiple wavelength regimes could be acquired using VO technologies amplified the benefits that such tool integration could deliver. Additionally, the new shared funding and communications channels between institutionally and geographically separated software developers that arose from various VO initiatives proved important in practice as a platform for experimentation and agreement in this area.

The external scripting capabilities of tools such as Aladin, SPLAT-VO and SAOImage ds9 already provided the option of tightly coupled master/slave control between pairs of applications, but did not lend themselves to the kind of cooperative interaction envisaged. The developers of Aladin experimented with Java interfaces designed for two-way communications; these delivered some limited integration, but were restricted to applications not only in java but residing in the same process. Meanwhile the Astro Runtime (Winstanley et al., 2007) developed by AstroGrid was providing a service interface designed for desktop tools based on a choice of communication technologies (XML-RPC, REST, Java RMI and JVM call).

From this background, in 2005 discussions between members of the AstroGrid, CDS and CINECA teams in the context of the Euro-VO framework and the SC4DEVO workshop series led to the development of a new communication protocol PLASTIC: the PLatform for AStronomical Tool InterConnection (Taylor et al., 2007; Boch et al., 2006). PLASTIC prototyped many features that were inherited later by SAMP, including a central hub, publish-subscribe messaging, use of XML-RPC, loosely-defined message semantics, and a pragmatic approach to providing "good-enough" communications. PLASTIC proved popular with developers and users, and was incorporated into a dozen or so tools, which could be used together effectively in productive and sometimes novel ways.

Interest in PLASTIC was however largely concentrated in Europe. Efforts to gain IVOA endorsement and extend the pool of applications that could communicate in this way led after some discussion to the drafting by European and US authors of a successor standard, the Simple Application Messaging Protocol, which was accepted as an IVOA Recommendation in 2008 (SAMP version 1.11). This standard was intentionally similar in many respects to PLASTIC, in order to avoid disrupting patterns of successful cooperation already in use, but the opportunity was taken to amend some decisions that experience had shown to be sub-optimal, and to expand its scope to accommodate other possible usage patterns. Changes made on the basis of lessons learned from PLASTIC included a simplification of the type system, complete language independence (though PLASTIC could be used from any language, certain parts of the protocol were defined with reference to Java), simplification of message targetting, improved security arrangements (security is still rudimentary in SAMP, but opportunities for trivial client spoofing have been removed), modification of message names (now both human-readable and wildcard-able rather than opaque URIs), definition of all message parameters and return values as named values rather than ordered lists, pervasive use of asynchronous messaging for robustness (though a synchronous façade is provided for convenience), improved error reporting, and better extensibility.

Also new in SAMP was the notion of a Profile to provide formal separation between the abstract messaging model and the transport layer. One reason for its introduction was to enable the possible future use of the protocol for messaging in less "PLASTIC-like" contexts. At the time, requirements for improved performance or security were envisaged; to date extensions in those directions have not been explored, but the Profile mechanism has paid off by supporting the later development of the Web Profile to support browser-based clients alongside desktop ones. The introduction of the Web Profile in SAMP version 1.3 (2012) has been the main change so far since the initial version.

2.2. Other Messaging Systems

Many other messaging infrastructures exist and it may be instructive to provide a short comparison between SAMP and some of the alternatives.

Several generic messaging frameworks share features with SAMP, for example AMQP, ZeroMQ, XPA, and D-Bus. To our knowledge, none satisfy SAMP's key requirements of an easily implementable platform-neutral standard supporting straightforward messaging between a shared community of clients in quite the way required, though some of these systems could be used as transport layers on which future SAMP profiles could be built, in the same way that XML-RPC has been used in the existing profiles. Although SAMP's design is not specific to astronomy, unlike many generic messaging libraries it deliberately presents a relatively inflexible interface, since restricting the decisions that need to be made to use it reduces the burden on client developers.

A couple of messaging systems however merit further mention: WAMP, the Web Application Messaging Protocol[1], bears some striking architectural similarities to SAMP including a combination of RPC and publish-subscribe messaging mediated by a central component known in WAMP terminology as the Router. However, it does not address the issue of router discovery, so there is no prescribed way to join the default community of clients; also, its development post-dated the publication of SAMP. An example of a domain-specific messaging framework is the Systems Biology Workbench (Sauro et al., 2003), which is close in spirit to SAMP, enabling platform-independent remote method invocation between components (known as "modules") in the field of systems biology, based around the SBML data format. One point of difference is that the SBW infrastructure itself orchestrates the loading of modules to provide the required functionality; in SAMP this choice of components is left to the user, though components like AppLauncher (Lafrasse et al., 2012) are available to layer this on top of the basic protocol where desired.

3. Design for Interoperability

The overriding objective for the design of SAMP has been to foster interoperability in practice. This requires not just a messaging system with sufficient communication capabilities, but also one which developers of popular analysis tools, and ideally private scripts as well, are actually willing and able to integrate into their software. To achieve this, a number of principles have been followed.

In the first place, it is as far as possible platform independent. The definition of the protocol is not dependent on or biassed towards use of particular implementation languages or operating systems. It differs in this respect from the earlier PLASTIC protocol, which while accessible from any language was in parts defined with reference to the Java platform, creating a perception which may have discouraged uptake by non-Java developers.

Second, ease of adoption. Application authors have in practice found basic use of SAMP to require little implementation effort. In practice, availability of SAMP client libraries developed within the SAMP community for a number of implementation languages have been an important factor in this. However the communications are, by design, simple enough that basic SAMP use is not hard to achieve given only an XML parser and HTTP access capabilities. Ease of use by end users is equally important, so that those running analysis tools can benefit from the integration capabilities that SAMP provides without needing to perform expert configuration (ideally, any explicit setup at all) or to understand the details of the messaging system.

Third, flexibility and extensibility. Building into the system the capability to use it in ways driven by the requirements of the client tools rather than just those forseen by the standard authors increases its likely usefulness.

Finally, the approach has been above all pragmatic, favouring the straightforward over the rigorous in cases of conflict. For instance message delivery is not guaranteed, but can be expected to work most of the time. The security model will prevent casual interference, but may be vulnerable to determined attack. Semantics are tagged using short readable strings on the assumption of sensible choices, rather than URIs with guaranteed private namespaces. Performance is easily good enough to handle exchange of short control messages on a timescale commensurate with user actions, but not for sustained throughput of high data volumes. (It may be noted that some, but not all, of these items could if required be ameliorated by future introduction of a new Profile with different transport characteristics).

These principles and their application, in some cases informed by positive and negative lessons from the experience of PLASTIC, might not be appropriate for all contexts but have led to a messaging infrastructure which ought to be easy for client developers to understand and adopt, and which has in fact been widely taken up.

4. Protocol Description

SAMP is based on a star topology, and its central component is a *Hub* through which all communications are passed (Figure 1). Clients first perform a resource discovery step to locate the Hub, and then *register* with it, establishing a private communication channel through which subsequent calls to the Hub's services can be made. These services include accepting metadata about the registering client, providing information about other registered clients, and forwarding *messages* to those clients. These messages may elicit responses, which may optionally be passed back to the message sender, again via the Hub. All clients are able to send messages in this way. Any client may optionally declare itself *callable*, in which case

[1]`http://wamp.ws`

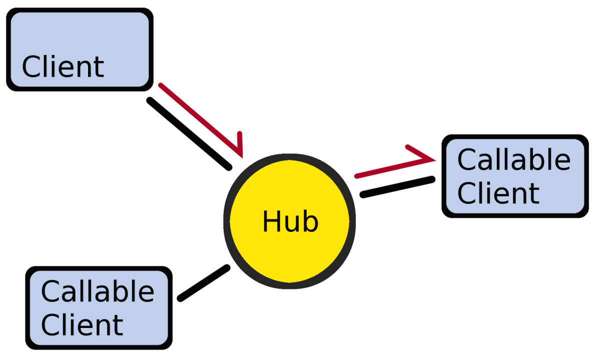


Figure 1: Schematic of SAMP *Hub* and *clients* joined in a star topology. Black lines indicate clients *registered* with the hub. The red half-arrows indicate the progress of a message from a sending client (which may or may not be *callable*) to a receiving client (which must be Callable), passing through the Hub.

it is also able to receive messages sent by others. Callability is optional since it is more difficult to achieve in client code, requiring some server-like capacity on top of the ability to invoke Hub services. In addition to declaring itself callable, a client wishing to receive messages must explicitly *subscribe*[2] to one or more *MTypes* (message types). Every message is labelled with an MType, and the Hub will only deliver messages to clients that have declared their interest in the MType in question with an appropriate subscription. When sending messages, clients may either *broadcast* them to all subscribed clients or target them to a named client, but in the latter case delivery will fail if the target client has not appropriately subscribed. If a client has no further use for SAMP communications (for instance on application exit), it can and should *unregister*.

This framework combines the notions of publish-subscribe (pub/sub) and Remote Procedure Call (RPC) messaging. Like publish-subscribe, messages are only delivered to appropriately subscribed recipients, but like RPC the sender may optionally target messages to a selected recipient, and may optionally receeve responses from the recipient(s). The targetting mode, response requirement, and message content are all decoupled from each other.

The details of this system are codified in a three-layer architecture:

Abstract API: defines the services provided by the Hub and clients

Profile: maps the Abstract API to specific communication operations, such as bytes on the wire

MTypes: provide semantics for the actual messages exchanged between clients

[2] The term "subscription" derives from the "publish/subscribe" messaging pattern. It may however be more helpful to think of subscription as *declaring support for* a particular message type.

Note that SAMP thus defines two distinct sets of Remote Procedure Call (RPC) operations: the functions declared by the Abstract API, concerning the mechanics of client-hub communication and message delivery, and SAMP Messages themselves classified by MType, bearing the application-level content that clients wish to exchange with each other. The syntax and semantics of the former are carefully defined by the SAMP standard, but the form and content of the latter are agreed outside of SAMP itself by cooperating client developers.

Because of the central rôle of the Hub in this pattern, it presents a single point of failure and potential bottle-neck. However, SAMP messages are usually short, and in practice performance issues have not generally been apparent.

The following subsections present a more detailed account of these ideas, along with some of the considerations that influenced their design. Sections 4.1, 4.2 and 4.3 describe the three architectural layers listed above, and sections 4.4 and 4.5 describe the underlying type system and how it is used to underpin extensibility in SAMP.

4.1. Abstract API

The Abstract API defines the messaging capabilities of SAMP. It takes the form of a list of a dozen or so function definitions with typed arguments and return values, and well-defined semantics. Most of these functions represent services provided by the Hub, for instance `register` (which returns information required by the client for future communications, typically an identification token) and `notifyAll` (which requests forwarding a given message to all appropriately subscribed clients). The remainder represent services required from callable clients, such as `receiveNotification` (which consumes a given message originating from another client).

The messaging model in principle associates a response with every message, containing at least a completion status flag along with zero or more MType-defined return values. However it is up to the sending client whether a response is required from any given message, and if not the "send-and-forget" (*notification*) pattern may be used, with lower cost for sender, recipient and hub. Message processing is fundamentally asynchronous from the receiver's point of view, so that message/response times are not limited to the lifetime of an RPC call in the underlying transport mechanism, but the Hub provides an optional synchronous façade for sending messages when clients expect fast turnaround and wish to avoid the additional complication of asynchronous processing.

4.2. Profile

A particular SAMP Profile is what turns the Abstract API into a set of rules that a client can actually use to communicate with a running Hub, and hence with other clients.

It performs two main jobs: first, it describes how the functions defined by the API are turned into concrete communication operations, by specifying an RPC-capable transport mechanism and rules for mapping the SAMP data types into the parameters and responses used by that mechanism. Second, it defines a hub discovery mechanism, which tells clients how to establish initial communications with the Hub, usually involving some authentication step. Particular profiles may also specify additional profile-specific hub or client services exposed as functions alongside those mandated by the Abstract API.

Initially (SAMP 1.11, 2009) only a single profile was defined, the Standard Profile. This uses XML-RPC[3] as a transport mechanism, and allows hub discovery by storing the URL of the hub's XML-RPC server along with a secret randomly generated key in a private "lockfile" in the user's home directory.

Version 1.3 of the standard (2012) introduced a second, the Web Profile, for use by web-based clients. This is required for applications running within web pages, since the sandboxed environment imposed by the browser makes the Standard Profile inaccessible. It shares use of XML-RPC and some other characteristics with the Standard Profile, but hub discovery has to be done differently, and there are a number of complications to do with security, described in Taylor et al. (2012a) as well as the Standard.

This decoupling between the functionality of the service interface and its incarnation in a specific transport mechanism allows different transports to be introduced without changes to the core protocol or existing clients, and has a number of benefits. In a given SAMP session, a client may use the most appropriate Profile for its SAMP communications and exchange messages seamlessly with other clients using different profiles; a desktop application can exchange messages with a web page just as easily as with another desktop client. This works because clients only ever communicate directly with the Hub and not with each other, while the Hub performs lossless translation between profile-specific network operations and the messaging model defined by the Abstract API.

Future requirements may result in additional Profile definitions, and there is nothing in principle to prevent hub developers from implementing new ones outside the frame of the SAMP standard. However, from an interoperability point of view it is important that all profiles are supported by all common Hub implementations, so that a client can rely on the availability of a chosen profile in a SAMP environment, and for this reason unnecessary proliferation of profiles is discouraged.

4.3. MTypes

An MType (message type) is the description for a message with particular syntax and semantics. It is analogous to a function definition in an API, and consists of a labelling string (sometimes also known as the MType) along with a set of zero or more typed and named arguments, a set of zero or more typed and named return values, and some associated semantics indicating what the sender of such a message is trying to convey.

A commonly used MType is `image.load.fits`, defined like this:

> *Name:*
>   `image.load.fits`
> *Semantics:*
>   Loads a 2-d FITS image
> *Arguments:*
>   `url` (string):
>   URL of the image to load
>   `image-id` (string, optional):
>   Identifier for use in subsequent messages
>   `name` (string, optional):
>   Name for labelling loaded image in UI
> *Return Values:*
>   None.

The name is a short hierarchical string composed of atoms separated by the "." character. As well as identifying to a recipient the type of an incoming message, it is used by clients to *subscribe* to messages, that is to indicate to the Hub which messages they are prepared to receive. For the purpose of subscription a limited wildcarding syntax is available, so by using the MType patterns `image.load.fits`, `image.*` or `*` a client may declare interest in only the above message, or all image-related messages, or all messages respectively.

In general, a callable client will only subscribe to those MTypes on which it can meaningfully act, so for instance an image analysis tool typically would subscribe to `image.load.fits`, but not to `spectrum.load.ssa-generic`. A client that has an image FITS file to send can then either query the Hub for those clients subscribed to the image load message and offer its user the choice of which one to target, or *broadcast* the image load message to the Hub, which will take care of forwarding it to all (and only) the image-capable clients.

4.4. Type System

Supporting the function list defined by the Abstract API and the parameters and return values specified by MTypes is a type system defining the types of value permitted, as well as rules for encoding various structured objects using these types: message objects themselves, success and failure message responses, application metadata, and MType subscription lists.

This system contains only three types: string, list and map. A string is a sequence of 7-bit ASCII printable characters, a list is an ordered sequence of strings, lists or maps, and a map is an unordered set of associations of string keys with values of type string, list or map. Structured objects are specified by the use of well-known keys in maps, there is no special representation for null values,

[3] XML-RPC is a simple protocol for remote procedure calling based on HTTP and XML. It resembles a very much slimmed-down SOAP. Documentation can be found at `http://www.xmlrpc.com/`.

and non-string scalar types must be serialized as strings. (Obvious) conventions are suggested for serializing integer, floating point and boolean values into string form, but these suggestions are provided for the convenience of MType definitions that wish to exchange such values without reinventing the wheel, and are not a normative part of the protocol.

This restricted type system has been deliberately chosen to introduce minimal dependency of messaging behaviour on the details of non-core parts of the delivery system, in particular profile-specific transport mechanisms and language-specific client libraries. This both reduces the restrictions on what languages and transport layers may be used with SAMP, and ensures that values transmitted will not be modified during processing by parts of the messaging system outside of client control.

The type system is rich enough to represent complex structured data where required, but note it is not intended for use with binary data, and transmission of bulk data or large payloads in general is discouraged within SAMP messages in favour of passing URLs around instead, meaning that client and Hub implementations can work on the assumption of short message payloads.

This convention of out-of-band bulk data transfer does place an additional burden on sending clients however, since to transmit a bulk data item (such as a table or image) not already available from an existing URL it is necessary to make it so available, for instance by writing bytes to a temporary file or serving them from an embedded HTTP server. It can also present complications if the sending and receiving client are not able to see the same URLs, for instance due to different security contexts; in this case, additional Hub services may be required to assist with data transfer between domains.

Note also that the string type does not natively accommodate Unicode text, including XML. The restriction to 7-bit ASCII is driven by the requirement for use from non-Unicode-capable environments such as Fortran, IDL and and some shell scripts. This has not caused known problems to date, but inability to handle Unicode text without additional encoding could prove awkward in some cases, and it may be necessary to revisit this in a future revision of the standard.

4.5. Extensible Vocabularies

Extensibility is built into this system via the notion of an *extensible vocabulary* used when representing structured objects.

Structured objects are represented as maps with well-known keys, but the rule is that additional keys are always permitted, and that hubs and clients must ignore any keys they do not understand, propagating them to downstream consumers where applicable (compare the NDF extension architecture described in Currie et al. (1989)). A corollary is that such non-well-known keys must be defined in such a way that ignoring them will result in reasonable behaviour. The Abstract API tends to prefer maps (unordered name/value pairs) over ordered parameter value lists, which makes this extensibility pervasive throughout the messaging system, applying for instance to client metadata and subscription declarations, message transmission information, and MType-specified message parameter lists and return values.

For instance, a client sending a message must pass it to the Hub as a map with two required keys: `samp.mtype` giving the MType label and `samp.params` giving the MType-specified parameter list (itself a map). But a client may optionally insert additional non-standard key/value pairs into that map, for instance using a non-standard key `priority` to associate a particular priority level with the message. If the Hub happens to support this non-standard feature, it is able to treat the message specially in view of this declaration; in any case it will propagate the message to recipient clients with the additional entry present, so if one of those supports this feature then it may use the value in processing. The same rule applies for instance to the MType-determined message parameter list; an MType like `image.load.fits` has a required parameter `url`, but a sending client may add a non-standard parameter like `colormap` (or `ds9.colormap`) alongside the well-known ones for the benefit of any client that happens to support it. Clients can therefore piggy-back experimental or application-specific instructions on top of generic messages to achieve more detailed control where available, falling back to the baseline functionality if it is not. Using this extensibility pattern, new or enhanced features of particular MTypes or of the protocol itself can be prototyped very easily, requiring no changes to the SAMP standard or infrastructure implementation beyond those components actually using the non-standard features, and imposing no negative impact on existing messaging operations. If they are found to be useful, they may be adopted in the future as (most likely optional) well-known keys alongside the original ones.

Some associated namespacing rules apply. Well-known keys defined by the SAMP standard are in the reserved `samp` namespace, meaning they begin with the string "`samp.`". When introducing non-standard keys it is not permitted to use this namespace, but any other syntactically legal string is allowed. The special namespace `x-samp` is available for keys proposed for future incorporation into the standard, and hubs and clients should treat keys which differ only in the substitution of `samp` for `x-samp` as identical, to ease standardisation of prototype features. In the case of MType parameters and return values, which are mostly not defined by SAMP itself, there is no reserved namespace.

5. Use in Practice

The protocol described above is capable of supporting a wide range of different messaging patterns. For use in a particular scenario, a number of practical considerations must also be worked out. This section discusses how the framework has been applied to date to support the original goal of helping to integrate data analysis tools used by astronomers.

Section 5.1 explains how hub provision is managed, section 5.2 describes some common patterns of message semantics, and section 5.3 addresses the social mechanisms by which these are agreed on by the SAMP community. Section 5.4 reviews the existing landscape of SAMP infrastructure software and SAMP-aware tools, and Section 5.5 provides some concrete examples of it in action.

5.1. Hub Provision

SAMP's star topology means that a Hub (in most cases, exactly one Hub) must be running for any messaging to take place. Ideally, a independent Hub process would be started as part of user session setup to ensure its constant availability. While this is quite possible and appropriate in some scenarios, even the minimal configuration required to establish it (a hub startup line in a session startup file) requires the kind of explicit user action which cannot always be relied upon. Simply put, if the functionality doesn't show up in the user interface with zero user effort, most users will never discover it.

It is common practice therefore, though by no means a requirement, for some SAMP-aware tools to come with an embedded hub. In this scenario, when a SAMP-aware tool starts up, it first checks for a running hub. If one exists, it registers with it; if not, and if it has the capability, it starts its own embedded Hub, and registers with that. Note that a client running an embedded Hub communicates with it in just the same way as with an independent one, it has no privileged access. Non-hub-capable clients may choose to check for a running Hub and connect on startup, on explicit user request, or when periodic polling indicates that one has become available. The effect is that usually when two or more SAMP-aware tools are running, a Hub will be present and those tools will find themselves connected to it, enabling messaging. Sometimes the application hosting the embedded hub will be shut down during a session taking the Hub down with it, and in that case another application may notice the fact and start one up, at which point some or all previously registered clients may notice the new hub and re-register with it.

This somewhat haphazard model of hub provision does not form a robust platform for high-reliability messaging, but, in accordance with SAMP's pragmatic approach, operates well enough most of the time, with a minimum of user effort; usually, it "just works". Note that where explicit control of an independent hub process can be arranged, for example as part of a managed user environment, more robust connectivity will result; an example is the Herschel Interactive Processing Environment (Balm, 2012).

5.2. MType Semantics

A messaging framework only serves any purpose if there exists a vocabulary of messages understood by the applications which are going to exchange them. In SAMP terms that means establishing a collection of more or less well-known MTypes (section 4.3). Choosing the right semantics for this collection is crucial to the utility and character of the messaging system in practice.

The most obvious approach for providing message-based control of an application is to identify (at least some of) the capabilities it offers and define a messaging interface with parameters and return values exposing those capabilities. An image display application might expose a set of MTypes allowing image load into a new window, zoom configuration, colour map choice, WCS display and so on. This allows other applications to control its behaviour in detail and is suitable for tight integration of a known set of tools with a good understanding of each other's capabilities, for instance to execute a pre-orchestrated sequence of processing steps.

However, this approach is less effective in less predictable environments. The controlling client needs to understand the capabilities of its partner client in order to control it. But if the set of tools in use at runtime is chosen by a user from an open-ended set rather than mandated by a developer, the identity of the partner client or clients is not known in advance. In general, different applications even of similar types have different capability sets and internal data models, and these cannot readily be encompassed by any single general abstraction. Different image display tools may support different data formats, may or may not support multiple loaded windows or images, may specify zooms in different ways, may offer different selections of colour maps, may provide WCS display with different options or not at all and so on, and the burden on a client wishing to control a range of different recipients quickly becomes large. Even if an application developer is prepared to study the messaging APIs offered by existing available tools and implement logic managing message dispatch for each case, the resulting code will not cope with applications unknown to the developer, for instance ones which have not yet been written.

For uncontrolled environments in which the user selects the range of cooperating tools at runtime therefore, a "loose integration" model has turned out to be more successful. This approach focuses on a messaging interface consisting of a fairly small number of MTypes with semantics that are non-client-specific and rather vague. The semantics of the most-used messages generally boils down to "*Here is an X*", where X may be some resource type such as a table, image, spectrum, sky position, coverage region, bibcode etc, or sometimes a reference to an X from a previous message, for instance a row selection relating to an earlier-sent table. The implication of such an MType is that the receiver should do something with the X in question: load, display, highlight, or otherwise perform some action which makes sense given the receiving application's capabilities. Callable SAMP clients should therefore advertise themselves (by subscribing to the appropriate MTypes) as X-capable tools only if they are in a position to do something sensible with an X should they be presented with one. Such an advertisement serves as a hint to potential X-senders, though it does not constitute a guarantee of any particular behaviour. This framework typically manifests itself in a client user interface as an option, for an X currently known by that client, either to *broadcast* it to all X-capable clients, or to target it to an entry selected by the user from a dynamically-discovered list of X-capable clients.

For clients to interoperate as reliably as possible in this scenario, it is not sufficient just to agree on the notion of a table or an image for exchange, it is also necessary to specify the exact data exchange format. In the case of tabular data, a variety of possible exchange formats is in common use: FITS binary and ASCII tables, VOTable, Comma-Separated Values and a host of others including many ASCII-based variants. Different choices are convenient in different usage contexts, suggesting the need for a variety of distinct format-specific MTypes. However, a proliferation of alternative exchange formats, though superficially convenient, erodes interoperability. If multiple exchange formats capaple of serializing the same thing are available, the sender has to choose which to send, and the receiver may or may not be able to receive it. Well-behaved recipients should include conversion code for as many formats as possible, and well-behaved senders should send data in a format dependent on what is supported by the intended recipient. For applications willing to expend a lot of effort on interoperability the work required at both ends increases rapidly with the number of available formats, and the rest may find themselves unable to exchange data of essentially compatible types, or the community of SAMP clients may fragment into format-specific sub-communities unable to communicate globally. As much as possible therefore, it is desirable to restrict the options to a single well-defined exchange format for each basic data type.

This can be a difficult balance to get right. In the case of images, astronomy is fortunate that FITS serves as the *lingua franca*, and the only commonly-used MType is `image.load.fits`. For tabular data, clients are strongly encouraged to use `table.load.votable` even if it means translating to/from some other format; however other `table.load.*` variants are in use for specialist purposes, for instance for the CDF format[4], which though tabular is not readily translated to VOTable without loss of information, and which tends to be used in communities not familiar with VOTable. In the case of spectra, for various reasons related to the form in which spectral data is typically obtained and the typical capabilities of spectrum-capable clients, the relevant MType is `spectrum.load.ssa-generic`, which permits any format to be used for the spectral data, with additional parameters specifying the format actually in use.

SAMP is capable of supporting both tight and loose integration, and both are in use, but for coupling interactive data analysis tools the loose integration model has proven the most productive, and able to support ways of working that have not been possible using other available messaging systems.

5.3. MType Definition Process

A suitable collection of MTypes must not only exist but be known by potential message senders in order for useful messaging to take place.

In the case of application-specific MTypes, the documentation of available MTypes and their definitions is clearly best handled as part of the documentation of the application itself. These typically provide functionality that only makes sense for a given tool, and make use of a suitably specific namespace, for instance `script.aladin.send`, which allows external applications to control Aladin (Bonnarel et al., 2000) by sending commands in its scripting language.

Developers are also free to define their own MTypes for use privately or in some closed group with locally agreed conventions for documentation, perhaps to support some tight-coupling-like usage.

However, for well-known MTypes intended for unrestricted use, for instance of the loose-coupling variety described above, some public process is required to establish and publicise their definitions, so that client developers can both become aware of the conventions currently in use by other tools, and contribute their requirements for new or modified functionality.

One possibility is to decide on a fixed list to form part of the SAMP standard. A small number of "administrative" MTypes, concerned with the messaging infrastructure, for instance `samp.hub.event.register` which informs existing clients when a new client has registered, have been written in to the standard in this way. All of these are in the reserved `samp.` namespace. However, for astronomy-specific MTypes this option was rejected, partly in order to avoid the introduction of astronomy-specific details into a standard which is otherwise quite suitable for use in other domains, and partly because the rather heavyweight IVOA process for standard review (Hanisch et al., 2010), in which draft to acceptance rarely takes less than 12 months, would impede introduction and updating of MTypes as required by implementation experience and new application demands. Another option is periodic publication of MType definitions in an IVOA Note. Such Notes may be issued at will without formal review, but no straightforward updating mechanism is in use, and this option was still felt to be undesirably cumbersome.

Instead, a wiki page[5] was set up on the IVOA web site listing currently agreed MTypes. An informal under-

[4] `http://cdf.gsfc.nasa.gov/`

[5] At time of writing, this wiki page can be found near `http://`

standing was adopted in which application developers are encouraged to discuss requirements for new MTypes or modifications to existing ones either privately or on the associated mailing list[6], and if consensus is reached, to edit the wiki page accordingly. This was intended as a provisional measure to be reviewed and modified as required, but, six years later, the need for a more formal process has not been apparent, and there are no current plans to modify this arrangement.

5.4. Existing Software

Since its first version in 2008, a wide range of SAMP-enabled infrastructure and application software has become available.

I don't think we want to list SAMP software by name do we?

Of infrastructure sofware that is actively maintained at the time of writing, interchangeable Hub implementations exist in Java and Python, and client toolkits in Java, Python, C and Javascript. Validation, debugging and development support tools are also available. Historical, partial or experimental SAMP functionality has also appeared in other languages including Perl, C Sharp and IDL.

Applications using SAMP number in the dozens, and include GUI analysis tools for images, catalogues, spectra, SEDs, time series and interferometry data, observation tools, outreach applications, command-line and graphical data access and manipulation suites, interactive processing environments, web archives either exposing simple results pages or offering sophisticated browser functionality, throwaway user scripts, and more. While initially developed and used mostly within stellar and galactic astronomy, use is now becoming common in related fields such as planetary science (Erard et al., 2014) and space physics (Génot et al., 2014). It is probably now the case that most astronomical applications that can benefit from interaction with other such tools include at least a basic SAMP capability. It is harder to ascertain to what extent this functionality is used in practice, but the enthusiasm of application developers to incorporate SAMP is presumably indicative of its utility.

5.5. Examples of Use

Current SAMP usage is most prevalent along the lines of the scenario outlined in the Introduction, allowing desktop and web-based clients to cooperate as a loosely integrated suite by employing the data-exchange MTypes in common use. Figure 2 shows some examples of the basic case where one client transmits a data item (spectrum, table or image) to another. Figure 3 illustrates a more sophisticated interaction in which two applications exchange data and control in both directions to provide linked views exploiting the capabilities of each.

SAMP has also been employed in other ways however, for instance to provide a private layer for RPC functionality required internally by Iris (Laurino et al., 2014) and to experiment with visualisation using on-demand data generation in Astro-WISE (Buddelmeijer and Valentijn, 2013).

6. Conclusions

SAMP provides a flexible and easy to use messaging framework, deployed in much current astronomical software, which supports various models of inter-tool communication.

The most productive of these models to date has been loose coupling between a user-selected set of independently developed interactive data acquisition and analysis applications, to deliver functionality approaching that of an integrated suite. This model is built on SAMP's combination of publish/subscribe messaging, vague message semantics, and ease of adoption by both developers and users leading to widespread uptake.

SAMP's flexibility means that it is capable of supporting other communication models, some in more marginal use now and some which may be explored further in the future. Introducing new Profiles, different MType libraries or alternative hub provision arrangements could render the same infrastructure suitable for contexts with different requirements for reliability, security or scalability. Another possible scenario is inter-host messaging to support collaborative work; this option has been under consideration throughout SAMP's history and is possible using existing profiles, though it is somewhat cumbersome to set up and has so far not received much user attention. Despite its development history, there is nothing in the protocol specific to either the Virtual Observatory or astronomy, so use in other problem domains is quite feasible, though the authors are not aware of effort currently being deployed in this direction.

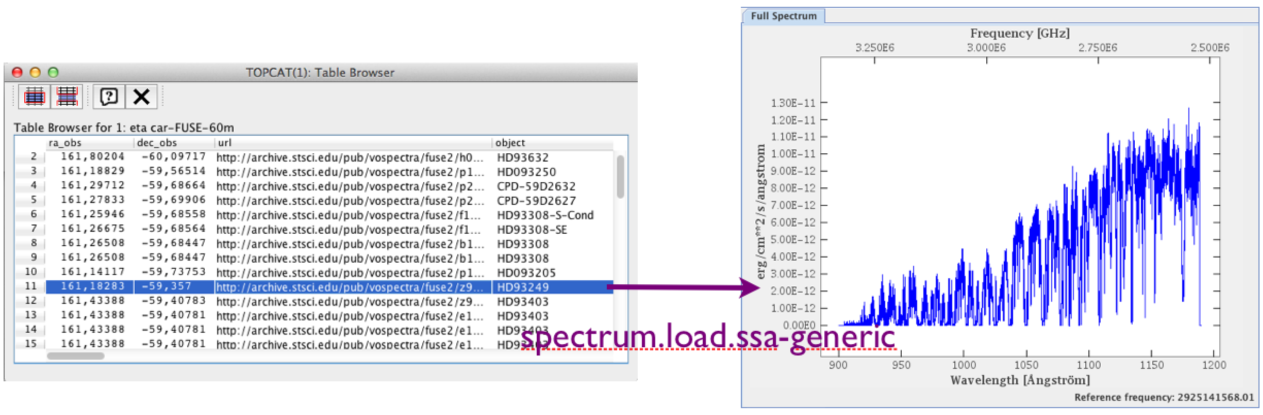
SAMP's design has been informed by the requirements and experimentation of the SAMP developer community, largely within the context of the Virtual Observatory, including positive and negative lessons learned from its predecessor PLASTIC. Some aspects of this design that have proved particularly successful include the decoupling of architectural concerns into API, transport mechanism and semantics, the lightweight, bottom-up process for agreement of semantics, and the built-in extensibility provided by pervasive use of extensible vocabularies.

Together, these fall under the heading of standardising only those things which need to be defined at a given stage, and leaving the option of filling in the details until a time and in a context when the requirements will be clearer. The need for the Web Profile was not forseen when the first version of the standard was published, but the transport/API decoupling meant it could be retrofitted with no

`www.ivoa.net/samp/`. If the process for MType publication changes in the future, that URL should still indicate where to look for a list.

[6]apps-samp@ivoa.net

(a)

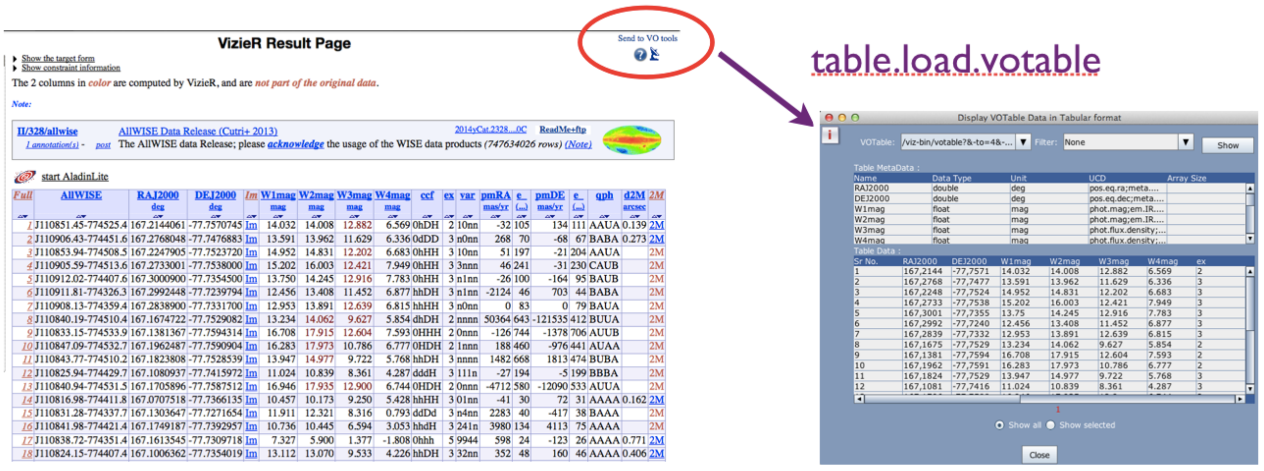


TOPCAT(1): Table Browser

Table Browser for 1: eta car-FUSE-60m

	ra_obs	dec_obs	url	object
2	161,80204	-60,09717	http://archive.stsci.edu/pub/vospectra/fuse2/h0...	HD93632
3	161,18829	-59,56514	http://archive.stsci.edu/pub/vospectra/fuse2/p1...	HD093250
4	161,29712	-59,68664	http://archive.stsci.edu/pub/vospectra/fuse2/p2...	CPD-59D2632
5	161,27833	-59,69906	http://archive.stsci.edu/pub/vospectra/fuse2/p2...	CPD-59D2627
6	161,25946	-59,68558	http://archive.stsci.edu/pub/vospectra/fuse2/f1...	HD93308-S-Cond
7	161,26675	-59,68564	http://archive.stsci.edu/pub/vospectra/fuse2/f1...	HD93308-SE
8	161,26508	-59,68447	http://archive.stsci.edu/pub/vospectra/fuse2/b1...	HD93308
9	161,26508	-59,68447	http://archive.stsci.edu/pub/vospectra/fuse2/b1...	HD93308
10	161,14117	-59,73753	http://archive.stsci.edu/pub/vospectra/fuse2/p1...	HD093205
11	161,18283	-59,357	http://archive.stsci.edu/pub/vospectra/fuse2/z9...	HD93249
12	161,43388	-59,40783	http://archive.stsci.edu/pub/vospectra/fuse2/z9...	HD93403
13	161,43388	-59,40781	http://archive.stsci.edu/pub/vospectra/fuse2/e1...	HD93403
14	161,43388	-59,40781	http://archive.stsci.edu/pub/vospectra/fuse2/e1...	HD93403
15	161,43388	-59,40781	http://archive.stsci.edu/pub/vospectra/fuse2/e1...	HD93403

(b)



VizieR Result Page

- Show the target form
- Show constraint information

The 2 columns in *color* are computed by VizieR, and are *not part of the original data*.

Note:

II/328/allwise — AllWISE Data Release (Cutri+ 2013) — 2014yCat.2328....0C — ReadMe+ftp

1 annotation(s) - *post* The AllWISE data Release; please ***acknowledge*** the usage of the WISE data products *(747634026 rows) (Note)*

start AladinLite

Full	AllWISE	RAJ2000	DEJ2000	Im	W1mag	W2mag	W3mag	W4mag	ccf	ex	var	pmRA	e_	pmDE	e_	qph	d2M	2M
		deg	deg		mag	mag	mag	mag				mas/yr	(...)	mas/yr	(...)		arcsec	
1	J110851.45-774525.4	167.2144061	-77.7570745	Im	14.032	14.008	12.882	6.569	0hDH	2	10nn	-32	105	134	111	AAUA	0.139	2M
2	J110906.43-774451.6	167.2768048	-77.7476883	Im	13.591	13.962	11.629	6.336	0dDD	3	n0nn	268	70	-68	67	BABA	0.273	2M
3	J110853.94-774508.5	167.2247905	-77.7523720	Im	14.952	14.831	12.202	6.683	0hHH	3	10nn	51	197	-21	204	AAUA		2M
4	J110905.59-774513.6	167.2733001	-77.7538000	Im	15.202	16.003	12.421	7.949	0hHH	3	3nnn	46	241	-31	230	CAUB		2M
5	J110912.02-774407.6	167.3000900	-77.7354500	Im	13.750	14.245	12.916	7.783	0hHH	3	n1nn	-26	100	-164	95	BAUB		2M
6	J110911.81-774326.3	167.2992448	-77.7239794	Im	12.456	13.408	11.452	6.877	hhDH	3	n1nn	-2124	46	703	44	BABA		2M
7	J110908.13-774359.4	167.2838900	-77.7331700	Im	12.953	13.891	12.639	6.815	hhHH	3	n0nn	0	83	0	79	BAUA		2M
8	J110840.19-774510.4	167.1674722	-77.7529082	Im	13.234	14.062	9.627	5.854	dhDH	2	nnnn	50364	643	-121535	412	BUUA		2M
9	J110833.15-774533.9	167.1381367	-77.7594314	Im	16.708	17.915	12.604	7.593	0HHH	2	0nnn	-126	744	-1378	706	AUUB		2M
10	J110847.09-774532.7	167.1962487	-77.7590904	Im	16.283	17.973	10.786	6.777	0HDH	2	1nnn	188	460	-976	441	AUAA		2M
11	J110843.77-774510.2	167.1823808	-77.7528539	Im	13.947	14.977	9.722	5.768	hhDH	3	nnnn	1482	668	1813	474	BUBA		2M
12	J110825.94-774429.7	167.1080937	-77.7415972	Im	11.024	10.839	8.361	4.287	dddH	3	111n	-27	194	-5	199	BBBA		2M
13	J110840.94-774531.5	167.1705896	-77.7587512	Im	16.946	17.935	12.900	6.744	0HDH	2	0nnn	-4712	580	-12090	533	AUUA		2M
14	J110816.98-774411.8	167.0707518	-77.7366135	Im	10.457	10.173	9.250	5.428	hhHH	3	01nn	-41	30	72	31	AAAA	0.162	2M
15	J110831.28-774337.7	167.1303647	-77.7271654	Im	11.911	12.321	8.316	0.793	ddDd	3	n4nn	2283	40	-417	38	BAAA		2M
16	J110841.98-774421.4	167.1749187	-77.7392957	Im	10.736	10.445	6.594	3.053	hhdH	3	241n	3980	134	4113	75	AAAA		2M
17	J110838.72-774351.4	167.1613545	-77.7309718	Im	7.327	5.900	1.377	-1.808	0hhh	5	9944	598	24	-123	26	AAAA	0.771	2M
18	J110824.15-774407.4	167.1006362	-77.7354019	Im	13.112	13.070	9.533	4.226	hhDH	3	32nn	352	48	160	46	AAAA	0.406	2M

Display VOTable Data in Tabular format

VOTable: /viz-bin/votable?&-to=4&-... Filter: None Show

Table MetaData :

Name	Data Type	Unit	UCD	Array Size
RAJ2000	double	deg	pos.eq.ra;meta....	
DEJ2000	double	deg	pos.eq.dec;meta....	
W1mag	float	mag	phot.mag;em.IR....	
W2mag	float	mag	phot.mag;em.IR....	
W3mag	float	mag	phot.flux.density;...	
W4mag	float	mag	phot.flux.density;...	

Table Data :

Sr No.	RAJ2000	DEJ2000	W1mag	W2mag	W3mag	W4mag	ex
1	167,2144	-77,7571	14.032	14.008	12.882	6.569	2
2	167,2768	-77,7477	13.591	13.962	11.629	6.336	3
3	167,2248	-77,7524	14.952	14.831	12.202	6.683	3
4	167,2733	-77,7538	15.202	16.003	12.421	7.949	3
5	167,3001	-77,7355	13.75	14.245	12.916	7.783	3
6	167,2992	-77,7240	12.456	13.408	11.452	6.877	3
7	167,2839	-77,7332	12.953	13.891	12.639	6.815	3
8	167,1675	-77,7529	13.234	14.062	9.627	5.854	2
9	167,1381	-77,7594	16.708	17.915	12.604	7.593	2
10	167,1962	-77,7591	16.283	17.973	10.786	6.777	2
11	167,1824	-77,7529	13.947	14.977	9.722	5.768	3
12	167,1081	-77,7416	11.024	10.839	8.361	4.287	3

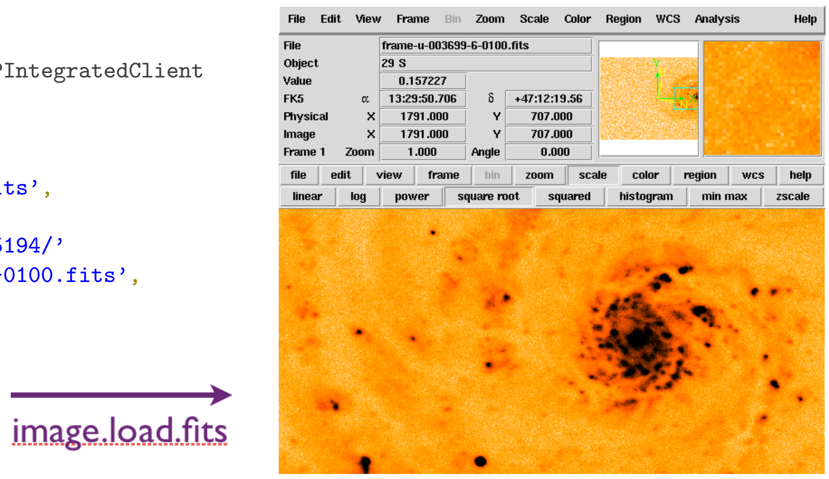
1

[✓ Show all] Show selected

Close

(c)

```
>>> from astropy.vo.samp import SAMPIntegratedClient
>>> client = SAMPIntegratedClient()
>>> client.connect()
>>> client.notify_all({
...       'samp.mtype': 'image.load.fits',
...       'samp.params': {
...           'url': 'file:///data/ngc5194/'
...                  'frame-u-003699-6-0100.fits',
...           'name': 'SDSS band u'
...       }
...   })
```



File	frame-u-003699-6-0100.fits			
Object	29 S			
Value	0.157227			
FK5	α	13:29:50.706	δ	+47:12:19.56
Physical	X	1791.000	Y	707.000
Image	X	1791.000	Y	707.000
Frame 1	Zoom	1.000	Angle	0.000

Figure 2: Transmission of data items from one SAMP client to another. (a) A table has been acquired using Simple Spectral Access Protocol by the TOPCAT table analysis tool, in which each row references an external spectrum by URL. When the user selects a row of this table, the spectrum is sent to the CASSIS spectrum tool. (b) The source catalogue resulting from a VizieR query, displayed in a web page, is sent to the VOPlot application. (c) A local FITS image is loaded into the ds9 image viewer from the AstroPy command line.

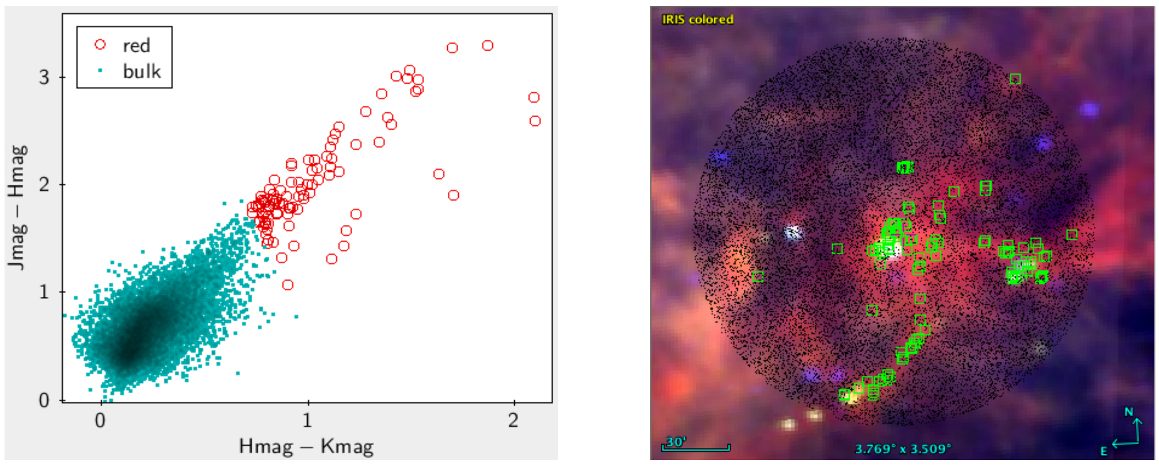


Figure 3: Linked view of data in TOPCAT and Aladin showing red objects in the vicinity of V* V410 Tau. A full list of sources in the region has been loaded into Aladin, then transferred to TOPCAT (using SAMP MType `table.load.votable`). The user plots a colour-colour diagram in TOPCAT and selects the reddest objects graphically, causing them to be displayed as red circles, then passes the selection back to Aladin (MType `table.select.rowList`) where they are shown as green squares. Clicking on one of the points in either application can highlight the corresponding point in the other (MType `table.highlight.row`). The SAMP interaction is mediated by the client GUIs, so the user does not need to understand the details. This example illustrates how SAMP enables seamless exploration of both parameter space and physical space.

disruption to existing client code. The fact that MType semantics are excluded from the standard itself means that these can be defined and iteratively adjusted with experience of a working transport infrastructure, rather than specifying them up front by committee decision as part of the protocol design, only to find them ill-adapted to tool deployment in practice.

SAMP is not a magic bullet. In typical current use the level of integration it offers between independently developed tools falls short of what would be available from a monolithic application, its pragmatic approach to communications can lead to patchy reliability, and its security model would not be appropriate for use with commercially sensitive data. However its ease of use and widespread uptake have delivered in practice an improved environment for desktop data analysis, allowing working astronomers to get more done.

References

Balm, P., 2012. Herschel Interactive Processing Environment (HIPE): Open to the World and the Future, in: Ballester, P., Egret, D., Lorente, N.P.F. (Eds.), Astronomical Data Analysis Software and Systems XXI, p. 733.

Boch, T., Comparato, M., Taylor, J., Taylor, M., Winstanley, N., 2006. PLASTIC - A Protocol for Desktop Application Interoperability. Technical Report. IVOA Note.

Bonnarel, F., Fernique, P., Bienaymé, O., Egret, D., Genova, F., Louys, M., Ochsenbein, F., Wenger, M., Bartlett, J.G., 2000. The ALADIN interactive sky atlas. A reference tool for identification of astronomical sources. Astron. Astrophys. Suppl. Ser. 143, 33–40. doi:`10.1051/aas:2000331`.

Buddelmeijer, H., Valentijn, E.A., 2013. Query driven visualization of astronomical catalogs. Experimental Astronomy 35, 283–300. doi:`10.1007/s10686-011-9263-0`, `arXiv:1110.2294`.

Currie, M.J., Wallace, P.T., Warren-Smith, R.F., 1989. Starlink General Paper 38. Starlink Project.

Erard, S., Cecconi, B., Le Sidaner, P., Berthier, J., Henry, F., Chauvin, C., André, N., Génot, V., Jacquey, C., Gangloff, M., Bourrel, N., Schmitt, B., Capria, M.T., Chanteur, G., 2014. Planetary science virtual observatory architecture. Astronomy and Computing doi:`DOI:10.1016/j.ascom.2014.07.005`.

Fitzpatrick, M., Laurino, O., Paioro, L., Taylor, M.B., 2013. Application interoperability with samp, in: Friedel, D., Freemon, M., Plante, R. (Eds.), ADASS XXII, ASP, San Francisco. p. 395.

Génot, V., André, N., Cecconi, B., Bouchemit, M., Budnik, E., Bourrel, N., Gangloff, M., Dufourg, N., Hess, S., Modolo, R., Renard, B., Lormant, N., Beigbeder, L., Popescu, D., Toniutti, J.P., 2014. Joining the yellow hub: Uses of the Simple Application Messaging Protocol in Space Physics analysis tools. Astronomy and Computing doi:`10.1016/j.ascom.2014.07.007`.

Hanisch, R.J., Arviset, C., Genova, F., Rino, B., 2010. IVOA Document Standards, Version 1.2. Technical Report. IVOA Recommendation.

Lafrasse, S., Bourges, L., Mella, G., 2012. SAMP App Launcher: An On-Demand VO Application Starter by JMMC, in: Ballester, P., Egret, D., Lorente, N.P.F. (Eds.), Astronomical Data Analysis Software and Systems XXI, p. 379.

Laurino, O., Budynkiewicz, J., D'Abrusco, R., Bonaventura, N., Busko, I., Cresitello-Dittmar, M., Doe, S.M., Ebert, R., Evans, J.D., Norris, P., Pevunova, O., Refsdal, B., Thomas, B., Thompson, R., 2014. Iris: An extensible application for building and analyzing spectral energy distributions. Astronomy and Computing doi:`10.1016/j.ascom.2014.07.004`.

McIlroy, M.D., Pinson, E.N., Tague, B.A., 1978. Unix time-sharing system: Foreword. Bell System Technical Journal 57, 1899.

Sauro, H.M., Hucka, M., Finney, A., Wellock, C., Bolouri, H., Doyle, J., Kitano, H., 2003. Next generation simulation tools: the Systems Biology Workbench and BioSPICE integration. OMICS 7, 355–72.

Taylor, J.D., Boch, T., Comparato, M., Taylor, M., Winstanley, N., Mann, R.G., 2007. Binding Applications Together with PLASTIC, in: Shaw, R.A., Hill, F., Bell, D.J. (Eds.), Astronomical Data Analysis Software and Systems XVI, p. 511.

Taylor, M.B., Boch, T., Fay, J., Fitzpatrick, M., Paioro, L., 2012a. Samp: Application messaging for desktop and web applications, in: Ballester, P., Egret, D., Lorente, N.P.F. (Eds.), ADASS XXI, ASP, San Francisco. p. 279.

Taylor, M.B., Boch, T., Fitzpatrick, M., Allan, A., Paioro, L., Taylor, J., Fay, J., 2012b. Simple Application Messaging Protocol, Version 1.3. Technical Report. IVOA Recommendation.

Winstanley, N., Taylor, J.D., Taylor, M.B., Noddle, K., Gonzalez-Solares, E., Lindroos, J., 2007. Astro Runtime: An API to the Virtual Observatory, in: Shaw, R.A., Hill, F., Bell, D.J. (Eds.), Astronomical Data Analysis Software and Systems XVI, p. 571.