International
Virtual
Observatory
Alliance

SAMP, Simple Application Messaging Protocol
Version 1.00

IVOA Working Draft 2008-05-15

This version:

http://www.ivoa.net/...

Latest version:

http://www.ivoa.net/...

Previous versions:

??

Editor(s):

Authors:

T. Boch - boch@astro.u-strasbg.fr
M. Fitzpatrick - fitz@noao.edu
M. Taylor - m.b.taylor@bris.ac.uk
...

Draft version: $Revision: 1.17 $ $Date: 2008/04/30 10:17:48 $

Abstract

SAMP is a messaging protocol that enables astronomy software tools to interoperate and communicate.

IVOA members have recognised that building a monolithic tool that attempts to fulfil all the requirements of all users is impractical, and it is a better use of our limited resources to enable individual tools to work together better. One element of this is defining common

file formats for the exchange of data between different applications. Another important component is a messaging system that enables the applications to share data and take advantage of each other's functionality. SAMP is intended to build on the success of a prior messaging protocol, PLASTIC, which has been in use in over a dozen astronomy applications for two years and has proven popular with users and developers. SAMP is an IVOA-endorsed standard that builds on this success. It is also intended to form a framework for more general messaging requirements.

Contents

1 Introduction

1.1 History

SAMP is a direct descendent of the PLASTIC protocol, which in turn grew —in the VOTech [**?**] framework —from the interoperability work of the Aladin [1] and VisIVO [2] teams. We also note the contribution of the team behind the earlier XPA protocol [3]. For more information on PLASTIC's history and purpose see the IVOA note *PLASTIC — a protocol for desktop application interoperability* [4].

SAMP has similar aims to PLASTIC, but incorporates lessons learnt from two years of practical experience and ideas from partners who were not involved in PLASTIC's initial design.

Broadly speaking, SAMP is an abstract framework for loosely coupled asynchronous RPC-like and/or event-based communication with extensible

message semantics using structured but weakly-typed data and based on a central service providing multi-directional publish/subscribe message brokering. These concepts are expanded on below. It attempts to make as few assumptions as possible about the transport layer or programming language with which it is used. It also defines a "Standard Profile" which specifies how to implement this framework using XML-RPC as the transport layer. The result of combining this Standard Profile with the rest of the SAMP standard is deliberately similar in design to PLASTIC, and the intention is that existing PLASTIC applications can be modified to speak SAMP instead without great effort.

1.2 Requirements and Scope

SAMP aims to be a simple and extensible protocol that is platform and language neutral. The emphasis is on a simple protocol with a very shallow learning curve in order to encourage as many application authors as possible to adopt it. In other words SAMP is intended to do what you need most of the time. The SAMP authors believe that this is the best way to foster innovation and collaboration in astronomy applications.

It is important to note therefore that SAMP's scope is reasonably modest; it is not intended to be the perfect messaging solution for all situations. In particular SAMP itself has no support for transactions, guaranteed message delivery, message integrity or messaging beyond a single machine. However, by layering the SAMP architecture on top of suitable messaging infrastructures such capabilities could be provided. These possibilities are not discussed further in this document, but the intention is to provide an architecture which is sufficiently open to allow for such things in the future with little change to the basics.

1.3 Types of Messaging

SAMP is currently limited to inter-application desktop messaging with the idea that the basic framework presented here is extensible to meet future needs, and so it is beyond the scope of this document to outline the many types of messaging systems in use today (these are covered in detail in many other documents). While based on established messaging models, SAMP is in many ways a hybrid of several basic messaging concepts; the protocol is however flexible enough that later versions should be able to interact fairly easily with other messaging systems because of the shared messaging models.

The messaging concepts used within SAMP include:

Publish/Subscribe Messaging: A publish/subscribe (pub/sub) messaging system supports an event driven model where information consumers and producers participate in message passing. SAMP applications "publish" a message, while consumer applications "subscribe" to messages of interest and consume events. Sending applications associate messages with a specific meaning, and the underlying messaging system routes messages to consumers based on the message types in which an applications has registered an interest.

Point-to-Point Messaging: In point to point messaging systems, messages are routed to an individual consumer which maintains a queue of "incoming" messages. In a traditional message queue, applications send messages to a specified queue and clients retrieve them. In SAMP, the message system manages the delivery and routing of messages, but also permits the concept of a directed message meant for delivery to a specific application. SAMP does not, however, guarantee the order of message delivery as with a traditional message queue.

Event-based Messaging: Event-based systems are systems in which producers deliver events, and in which messaging middleware delivers events to consumers based upon a previously specified interest. One typical usage pattern of these systems is the publish-subscribe paradigm, however these systems are also widely used for integrating loosely coupled application components. SAMP allows for the concept that an "event" occurred in the system and that these message types may have requirements different from messages where the sender is trying to invoke some action in the network of applications.

Synchronous vs. Asynchronous Messaging: As the term is used in this document, a "synchronous" message is one which blocks the sending application from further processing until a reply is received. However, SAMP messaging is based on "asynchronous" message and response in that the delivery of a message and its subsequent response are handled as separate activities by the underlying system. With the exception of the synchronous message pattern supported by the system, sending or replying to a message using SAMP allows an application to return to other processing while the details of the delivery are handled separately.

1.4 About this document

This document contains the following main sections describing the SAMP protocol and how to use it. Section 2 covers the requirements, basic concepts and overall architecture of SAMP. Section 3 defines abstract (i.e. independent of language, platform and transport protocol) interfaces which clients

and hubs must offer to participate in SAMP messaging, along with data types and encoding rules required to use them, including an abstract API. Section 4 explains how the abstract API can be mapped to specific network operations to form an interoperable messaging system, and defines the "Standard Profile", based on XML-RPC, which gives a particular set of such mappings. Section 5 describes the use of the MType keys used to denote message semantics, and outlines an MType vocabulary.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

2 Architectural Overview

This section provides a high level view of the SAMP protocol.

2.1 Nomenclature

In the text that follows these terms are used:

Client: An application that understands SAMP. Could be a Sender, Recipient, or both.

Hub: A broker service for routing SAMP Messages.

Sender: A Client that can send SAMP Messages to Recipients via the Hub.

Recipient: A Client that can receive SAMP Messages from the hub. These may have originated from other Clients or be from the hub itself.

Message: A communication sent from a Sender to a Recipient via a SAMP Hub. May or may not provoke a Response.

Response: A communication which may be returned from a Recipient to a Sender in reply to a previous Message. A Response may be either a return value or an error object. In the terminology of this document, a Response is not itself a Message. A Response is also known as a *reply* in this document.

MType: A key defining the semantics of a Message and of its arguments and return values (if any). Every Message contains exactly one MType, and a Message is only delivered to Clients subscribed to that MType.

Subscribed Client: A Client is said to be subscribed to a given MType if it has declared to the Hub that it is prepared to receive Messages with that MType.

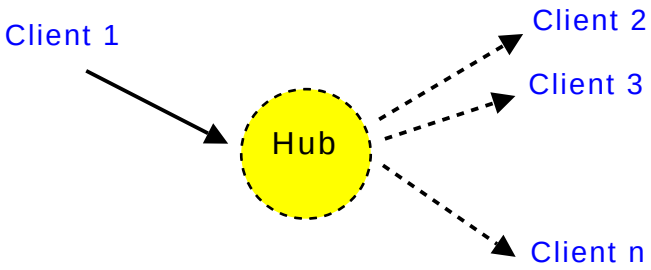

Figure 1: The SAMP hub architecture

Callable Client: A Client to which the Hub is capable of performing callbacks. Clients are not obliged to be Callable if they wish only to send messages, and not to receive messages or asynchronous replies.

Broadcast: To send a SAMP Message to all subscribed clients.

Profile: A set of rules which map the abstract API defined by SAMP to a set of network operations which may be used by Clients to send and receive actual Messages.

2.2 Messaging Topology

SAMP has a hub-based architecture (see Figure 1). The hub is a single service used to route all messages between clients. This makes application discovery more straightforward in that each client only needs to locate the hub, and the services provided by the hub are intended to simplify the actions of the client. A disadvantage of this architecture is that the hub may be a message bottleneck and the hub may be the single point of failure. However, it is not anticipated that message traffic will be such that the former is an issue, and the latter may be mitigated by an appropriate strategy for hub restart if failure is likely.

Note that the hub is defined as a service interface which may have any of a number of implementations. It may be an independent application running as a daemon, an adapter interface layered on top of an existing messaging infrastructure, or a service provided by an application which is itself one of the hub's clients. The only requirement is that exactly one hub must be running (per user-id) at a time for messaging to take place.

2.3 The Life cycle of a Client

A SAMP client goes through the following phases:

1. Determine whether a hub is running by using the appropriate hub discovery mechanism
2. If so, use hub discovery mechanism to work out how to communicate with the hub.
3. Register with the hub.
4. Store metadata such as client name, description and icon in the hub.
5. Subscribe to a list of MTypes to define messages which may be received.
6. Interrogate the hub for metadata of other clients.
7. Send and/or receive messages to/from other clients via the hub.
8. Unregister with the hub.

Phases 4–7 are all optional and may be repeated in any order.

2.4 The Life cycle of a Hub

A SAMP hub goes through the following phases:

1. Locate any existing hub by using the appropriate hub discovery mechanism.
   (a) Check whether the existing hub is alive.
   (b) If so, exit.
2. If no hub is running, or a hub is found but is not functioning, write/overwrite the hub discovery record and start up.
3. Await client registrations. When a client makes a legal registration, assign it a public id, and add the application to the table of registered clients under the public id. Broadcast a message [ref to messages section here] to all subscribed clients announcing the registration of a new client.
4. When a client stores metadata in the hub, broadcast a message [...] to all candidate clients and make the metadata available.
5. When a client updates its list of subscribed MTypes, broadcast a message[...] to all subscribed clients.
6. When the hub receives a message for relaying, pass it on to appropriate recipients which are subscribed to the message's MType. Broadcast messages are sent to all subscribed clients except the sender, messages with a specified recipient are sent to that recipient if it is subscribed.
7. Await client unregistrations. When a client unregisters, broadcast a message [...] to all subscribed clients announcing the unregistration and remove the client from the table of registered clients.
8. If the hub is unable to communicate with a client, it may unregister it as described in phase 7.

9. When the hub is about to shutdown, broadcast a message[...] to all subscribed clients.
10. Delete the hub discovery record.

Phases 3–8 are responses to events which may occur multiple times and in any order.

Readers should note that, given this scheme, race conditions may occur. We could have for instance a client trying to register with a hub which has just shut down, or an attempt to send to a recipient which has already unregistered. Specific profiles MAY define best-practice rules in order to manage at best these conditions, but in general clients should be aware that in the absence of guaranteed message delivery and timing, unexpected conditions are possible.

2.5 Message Delivery Patterns

Messages can be sent according to three patterns, differing in how and whether a response is returned to the sender:

1. Notification
2. Asynchronous Call/Response
3. Synchronous Call/Response

The Notification pattern is strictly one-way while in the Call/Response patterns the recipient returns a response to the sender.

If the sender expects to receive some useful data as a result of the receiver's processing, and/or if it wishes to find out whether and when the processing is completed, it should use one of the Call/Response variants. If on the other hand the sender has no interest in what the recipient does with the message once it has been sent, it may use the Notification pattern. Notification, since it involves no communication back from the recipient to the sender, uses fewer resources. Although typically "event"-type messages will be sent using Notify and "request-for-information"-type messages will be sent using Call/Response, the choice of which delivery pattern to use is entirely distinct from the content of the message, and is up to the sender; any message (MType) may be sent using any of the above patterns. Apart from the fact of returning or not returning a response, the recipient should process messages in exactly the same way regardless of which pattern is used.

From the receiver's point of view there are only two cases, Notification and Asynchronous Call/Response. However the hub provides a convenience method which simulates a synchronous call from the sender's point of view. The purpose of this is to simplify the use of the protocol in such situations as

scripting environments which cannot easily handle asynchronicity. However, it is recommended to use the asynchronous pattern where possible due to its greater robustness.

2.6 Use of Profiles

The design of SAMP is based on the abstract interfaces defined in Section 3. On its own however, this does not include the detailed instructions required by application developers to achieve interoperability. To achieve that, application developers must know how to map the operations in the abstract SAMP interfaces to specific I/O (in most cases, network) operations. It is these I/O operations which actually form the communication between applications. The rules defining this mapping from interface to I/O operations are what constitute a SAMP "Profile" (the term "Implementation" was considered for this purpose, but rejected because it has too many overlapping meanings in this context).

There are two ways in which such a Profile can be specified as far as client application developers are concerned:

1. By describing exactly what bytes are to be sent using what wire protocols for each SAMP interface operation
2. By providing one or more language-specific libraries with calls which equate to those of the SAMP interface

Although either is possible, SAMP is well-suited for approach (1) above given a suitable low-level transport library. This is the case since the operations are quite low-level, so client applications can easily perform them without requiring an independently developed SAMP library. This has the additional advantages that central effort does not have to be expended in producing language-specific libraries, and that there are no problems for application developers using "unsupported" languages.

Section 4 describes a Profile along the lines of (1) above, based on XML-RPC, which can be used directly by client and hub developers, in conjunction with the abstract interface description in Section 3 to write interoperable applications. This is at present the only SAMP Profile which has been defined.

Although splitting the abstract interface and Profile descriptions in this way complicates the document a little, it separates the basic design principles from the details of how to apply them, and it opens the door for other Profiles serving other use cases in the future.

3 Abstract APIs and Data Types

3.1 Hub Discovery Mechanism

In order to keep track of which hub is running, a hub discovery mechanism, capable of storing information about how to determine the existence of and communicate with a running hub, is needed. This is a Profile-specific matter and a specific prescription will be described in 4.3.

3.2 Communicating with the hub

The details of how a client communicates with the hub are Profile-specific and will be covered in section 4.

3.3 Registering with the hub

A client registers with the hub to:

1. establish communication with the hub
2. advertise its presence to the hub and to other clients

Immediately following registration, the client will typically perform some or all of the following optional operations:

3. supply the hub with metadata about itself, using the `setMetadata()` call
4. tell the hub how it wishes the hub to communicate with it, if at all (the mechanism for this is profile-dependent, and it may be implicit in registration)
5. inform the hub which MTypes it wishes to receive, using the `setMtypes()` call

3.4 SAMP Data Types

For all hub/client communication, including the actual content of messages, SAMP uses three conceptual data types:

1. `string` — a scalar value consisting of a sequence of characters; each character may be in the range 0x01–0x7f
2. `list` — an ordered array of data items
3. `map` — an unordered associative array of string-data item key-value pairs

These types can in principle be nested to any level, so that the elements of a list or the values of a map may themselves be strings, lists or maps.

There is no reserved representation for a null value, and it is illegal to send a null value in a SAMP context even if the underlying transport protocol permits this. However a zero-length string or an empty list or map may where appropriate be used to mean null.

Although SAMP imposes no maximum on the length of a string, particular transport protocols or implementation considerations may effectively do so; in general hub and client implementations are not expected to deal with data items of unlimited size. General purpose MTypes should therefore be specified so that bulk data is not sent within the message — in general it is preferred to define a message parameter as the URL (or filename) of a potentially large file rather than as the inline text of the file itself.

At the protocol level there is no provision for typing of scalars; unlike many RPC protocols SAMP does not distinguish syntactically between strings, integers, floating point values, booleans etc. This minimizes the restrictions on what underlying transport protocols may be used, and avoids a number of problems associated with using typed values from untyped languages such as Python and Perl.

Some MTypes will however wish to define parameters or return values which have non-string semantics, and conventions for encoding these as `string`s are therefore in practice required. Such conventions only need to be understood by the sender and recipient of a given message and so can be established on a per-MType basis, but to avoid unnecessary duplication of effort some commonly-used type encoding conventions are defined in the following section.

3.5 Scalar type encoding conventions

The following BNF productions are used in the type encoding conventions below:

```
<digit>          ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" |
                     "7" | "8" | "9"
<digits>         ::= <digit> | <digits> <digit>
<decimal_digits> ::= <digits> | <digits> "." | "." <digits>
                   | <digits> "." <digits>
<sign>           ::= "+" | "-"
```

- `<SAMP int> ::= [ <sign> ] <digits>`
  An integer value is encoded using its decimal representation with an op-

tional preceding sign and with no leading, trailing or embedded whitespace. There is no guarantee about the largest or smallest values which can be represented, since this will depend on the processing environment at decode time.

- `<SAMP float> ::= [ <sign> ] <decimal_digits> [ "e" | "E" [ <sign> ] <digits> ]`
  A floating point value is encoded as a mantissa with an optional preceding sign followed by an optional exponent part introduced with the character "`e`" or "`E`". There is no guarantee about the largest or smallest values which can be represented or about the number of digits of precision which are significant, since these will depend on the processing environment at decode time.
- `<SAMP boolean> ::= "0" | "1"`
  A boolean value is represented as an integer: zero represents false, and any other value represents true. 1 is the recommended value to represent true.

The numeric types are based on the syntax of the C programming language, since this syntax forms the basis for typed data syntax in many other languages. This list may be extended in the future if required.

Particular MType definitions may use these conventions or devise their own as required. Where the conventions in this list are used, message documentation should make it clear using a form of words along the lines "this parameter contains a *SAMP int*".

3.6 Application Metadata

A client may store metadata in the form of a `map` of key-value pairs in the hub for retrieval by other clients. Typical metadata will be the human-readable name of the application, a description and a URL to its icon, but other values are permitted. The following keys are defined for well-known metadata items:

`samp.name` - A one word title for the application.
`samp.description.text` - A short description of the application, in plain text.
`samp.description.html` - A description of the application, in HTML.
`samp.icon.url` - The URL of an icon in png, gif or jpeg format.
`samp.documentation.url` - The URL of a documentation web page.

All of the above are OPTIONAL, but `samp.name` is strongly RECOMMENDED. [?? should others be RECOMMENDED ??]
Applications may store metadata under any keys, except that keys beginning '`samp.`" may only be used as described here.

3.7 What is a message?

A message is an abstract container for the information we wish to send to another application. The message itself is that data which should arrive at the receiving application. It may be transmitted along with some external items (e.g. sender/recipient/message identifiers) required to ensure proper delivery or handling.

A message contains at least the following parts:

An MType: a string which defines the meaning of the message, for instance the instruction to load a table. It also, via external documentation, defines the names, types and meanings of any parameters as well as the type and meaning of the return value. MTypes are discussed in more detail in Section 5.

Parameters: zero or more named values giving the data required for the receiver to act on the message, for instance the URL of a particular table. The names and semantics of these parameters are determined by the MType.

It may also contain other items, for instance a timestamp, but such items are not currently described by this standard.

[?? ref-id ??]

It is legal to include parameters in the parameter object which are not documented by the MType. Any parameters whose name a receiver does not recognise in the context of the given MType should be ignored. The intention is that MTypes can evolve by having new parameters added which provide additional functionality if the need is identified during use. Although in some cases such refinement will require a redesign with completely different parameters (and a new MType), it is often the case that MTypes can be improved by adding new optional parameters which can be harmlessly ignored by clients only familiar with the older version.

Some common MTypes and their parameter objects are described in section 5.

3.8 Message and Response Encoding

A message as described in Section 3.7 may be encoded in the terms of the datatypes described in Section 3.4 as a `map`. The defined keys and their associated values are as follows:

`mtype` The value is a `string` giving the MType for the message.

`params` The value is a `map` giving the parameter values for the message. Each parameter is represented by a key-value pair in which the key is the documented name of the parameter, and the value is its value, with the type as documented by the MType.

No other keys are currently defined. Keys which are not defined here are not illegal, but if an application encounters keys which it does not understand, it should generally ignore them.

The response to a message, if one is expected, is a `map` and may be one of two things:

1. a return value (successful completion)
2. an error object (failure)

A flag is passed with the response object to indicate which of these it is; clients do not have to examine the object to find out whether it represents a return value or an error.

In the case of a return value, the type and meaning of any data contained in it as key-value pairs SHOULD be documented by the MType.

In the case of an error response, the map may contain entries with the following keys:

`errortxt` (REQUIRED) — A short string describing what went wrong. This will typically be delivered to the user of the sender application.

`usertxt` (OPTIONAL) — A free-form string containing any additional text an application wishes to return. This may be a more verbose error description meant to be appended to the errortxt string, however it is undefined how this string should be handled when received.

`debugtxt` (OPTIONAL) — A longer string which may contain more detail on what went wrong. This is typically intended for debugging purposes, and may for instance be a stack trace.

`code` (OPTIONAL) — A string containing a numeric or textual code identifying the error.

No other keys are currently defined. Keys which are not defined here are not illegal, but if an application encounters keys which it does not understand, it should generally ignore them.

[?? should these keys be `samp.mtype` etc, with `samp.*` a reserved part of the namespace and others available for public use ??]

3.9 Sending and Receiving Messages

As outlined in Section 2.5, three messaging patterns are supported, differing according to whether and how the response is returned to the sender. For

a given MType there may be a messaging pattern that is most typically used, but there is nothing in the protocol that ties a particular MType to a particular messaging pattern.

From the point of view of the sender, there are three ways in which a message may be sent, and from the point of view of the recipient there are two ways in which one may be received. These are described as follows.

Notification: In the notification pattern, communication is only in one direction:

1. The sender sends a message to the hub for delivery to one or more recipients.
2. The hub forwards them to those requested recipients which are subscribed
3. No reply from the recipients is expected or possible

Notifications can be sent to a given recipient or broadcast to all recipients. The notification pattern for a single recipient is illustrated in Figure 2.

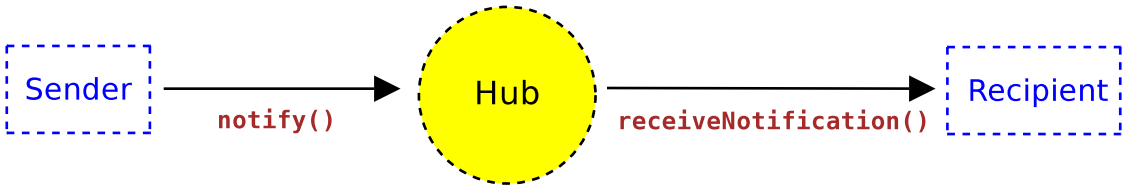

Figure 2: Notification pattern

Asynchronous Call/Response: In the asynchronous call pattern, *message IDs* are used to tie together messages and their replies:

1. The sender sends a message to the hub for delivery to one or more recipients, supplying along with the message an ID string, *sender-msg-id*.
2. The hub forwards the message to the appropriate recipients, supplying along with the message an ID string, *hub-msg-id*.
3. Each recipient processes the message, and sends its response back to the hub along with the ID string *hub-msg-id*.
4. Using a callback, the hub passes the response back to the original sender along with the ID string *sender-msg-id*.

The sender is free to use any value for the *sender-msg-id*. There is no requirement on the form of the *hub-msg-id* (it is not intended to be parsed by the recipient), but it must be sufficient for the hub to pair

messages with their responses reliably, and to pass the correct *sender-msg-id* back with the response to the sender[1]. Asynchronous calls may be sent to a given recipient or broadcast to all recipients. In the latter case, the sender should be prepared to deal with multiple responses to the same call. The asynchronous pattern is illustrated in Figure 3.

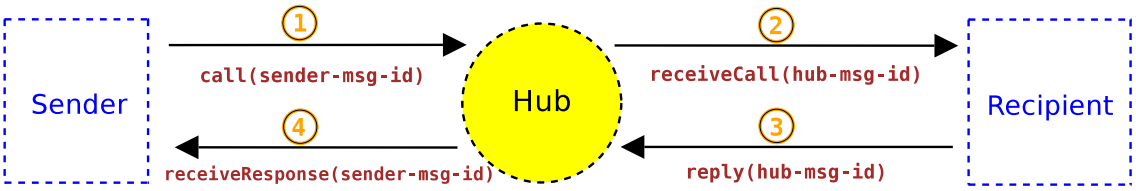

Figure 3: Asynchronous pattern

Synchronous Call/Response A synchronous utility method is provided by the hub, mainly for script environments where dealing with asynchronicity might be a problem. The hub will provide synchronous behaviour to the sender, interacting with the receiver in exactly the same way as for the asynchronous case above.

1. The sender sends a message to the hub for delivery to a given recipient. This call blocks until the response is available.
2. The hub forwards the message to the recipient, supplying along with the message an ID string, *hub-msg-id*.
3. The recipient processes the message, and sends its response back to the hub along with the ID string *hub-msg-id*.
4. The hub sends the response as the return value from the original blocking call made by the sender.

There is no broadcast counterpart for the synchronous call. This pattern is illustrated in Figure 4.

Note that the two different cases from the receiver's point of view, *Notification* and *Call/Response*, differ only in whether a response is returned to the hub. In other respects the receiver should process the message in exactly the same way for both patterns.

[1] One way a hub might implement this is to generate *hub-msg-id* by concatenating the sender's client id and the *sender-msg-id*. When any response is received the hub can then unpack the accompanying *hub-msg-id* to find out who the original sender was and what *sender-msg-id* it used. In this way the hub can determine how to pass each response back to its correct sender without needing to maintain internal state concerning messages in progress. Hub and client implementations may wish to exploit this freedom in assigning message IDs for other purposes as well, for instance to incorporate timestamps or checksums.

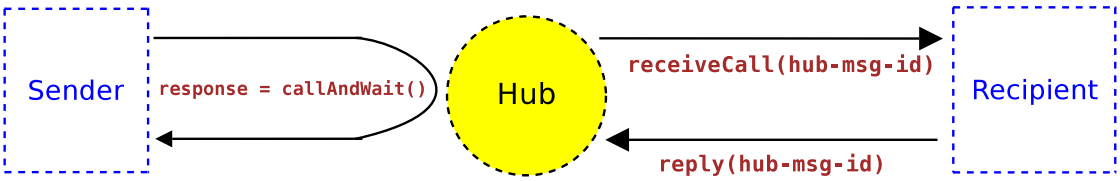

Figure 4: Synchronous pattern

3.10 Operations a hub must support

This section describes the operations that a hub must support and the associated data that must be sent and received. Message and response arguments are encoded as `map` objects as described in Section 3.8. The precise details of how these operations map onto method names and signatures is Profile-dependent. The mapping for the Standard Profile is given in section 4.2.

- `register()`
  Method called by a client wishing to register with the hub. Note that the form of this call may vary according to the requirements of the particular Profile in use. For instance authentication tokens may be passed in one or both directions to complete registration.

- `unregister()`
  Method called by a client wishing to unregister from the hub

- `setMetadata(map metadata)`
  Method called by a client to declare its metadata (name, icon, description, etc — see section 3.6 for details). May be called zero or more times to update hub state; the most recent call is the one which defines the client's currently declared metadata.

- `map metadata = getMetadata(string client-id)`
  Returns the metadata information for the client whose public id is `client-id`.

- `setMTypes(list mtypes)`
  Method called by a callable client to declare the MTypes it wishes to subscribe to. May be called zero or more times to update hub state; the most recent call is the one which defines the client's currently subscribed MTypes.
  [?? possibly wildcarding allowed here ??]

- `list mtypes = getMTypes(string client-id)`
  Returns the array of subscribed MTypes for the client whose public id is `client-id`.

- `list client-ids = getRegisteredClients()`
  Returns the list of public ids of other registered clients. The caller's id is not included.

- `list client-ids = getSubscribedClients(string mtype)`
  Returns the list of public ids of all other registered clients who are subscribed to the MType `mtype`. The caller's id is not included, even if it is subscribed.

- `string client-id = getHubId()`
  Returns the client-id which identifies the hub itself as a client. This is the id which the hub uses for instance to send administration messages to other candidate clients.

- `notify(string recipient-id, map message)`
  Method called by a client wishing to send a notification to a given recipient.

- `notifyAll(map message)`
  Method called by a client wishing to broadcast a notification to all recipients.

- `call(string recipient-id, string msg-id, map message)`
  Method called by a callable client wishing to send an asynchronous call to a given recipient.

- `callAll(string msg-id, map message)`
  Method called by a callable client wishing to broadcast an asynchronous call to all recipients.

- `map response = callAndWait(string recipient-id, map message)`
  Method called by a client wishing to make a synchronous call to a given recipient.

- `reply(string msg-id, string success, map response)`
  Method called by a client to send its response to a given message. `success` is a SAMP boolean ("`1`" for successful completion or "`0`" for

an error). In the case of success, `response` is the return value as documented for the MType of the message. In the case of failure, `response` is an error object as described in 3.8.

All these operations with the exception of `callAndWait()` should complete, and where appropriate return a result, quickly.

3.11 Operations a hub may call on a client

We list in this section the operations which may be called on a callable client. Note that not all clients may be callable; special (Profile-dependent) steps may be required for a client to inform the hub how it may be contacted, and thus become callable. Clients which are not callable are unable to receive messages or use the asynchronous call/response pattern.

Message and response arguments are encoded as `map` objects as described in Section 3.8. The precise details of how these operations map onto method names and signatures is Profile-dependent. The mapping for the Standard Profile is given in section 4.2.

- `receiveNotification(string sender-id, map message)`
  Method called by the hub when dispatching a notification to its recipient.

- `receiveCall(string sender-id, string msg-id, map message)`
  Method called by the hub when dispatching a call to its recipient. The client MUST at some later time make a matching call to `reply()` on the hub.

- `receiveResponse(string responder-id, string msg-id, string success, map response)`
  Method used by the hub to dispatch to the sender the response of an earlier asynchronous call. `success` is a SAMP boolean ("`1`" for successful completion or "`0`" for an error). In the case of success, `response` is the result value as documented for the MType of the original message. In the case of failure, `response` is an error object as described in 3.8.

All these operations should complete quickly.

3.12 General error processing

Hubs and clients should use the usual error reporting mechanisms of the transport protocol in use in the case of bad calls of the operations defined

in Sections 3.10 and 3.11, for instance use of syntactically invalid parameter types.

Errors produced by clients when processing call-type SAMP messages themselves (in response to a syntactically legal `receiveCall()` operation) should be signalled in the way the matching `reply()` call is made. In case of an error, the `success` flag should be set to "`0`" and the `response` object should be filled in with error information as described in Section 3.8.

4 Standard Profile

Section 2 defines the concepts and operations used in SAMP messaging. As explained in Section 2.6, in order to implement this architecture some concrete choices about how to instantiate these concepts are required.

This section gives the details of a SAMP Profile based on the XML-RPC specification [6]. Hub discovery is via a lockfile in the user's home directory.

XML-RPC is a simple general purpose Remote Procedure Call protocol based on sending XML documents over HTTP (it resembles a very lightweight version of SOAP). Since the mappings from SAMP concepts such as API calls and data types to their XML-RPC equivalents is very straightforward, it is easy for application authors to write compliant code without use of any SAMP-specific library code. An XML-RPC library, while not essential, will make coding much easier; such libraries are available for many languages.

4.1 Data Type Mappings

The SAMP argument and return value data types described in Section 3.4 map straightforwardly onto XML-RPC data types as follows:

SAMP type		XML-RPC element
`string`	$\longleftrightarrow$	`<string>`
`list`	$\longleftrightarrow$	`<array>`
`map`	$\longleftrightarrow$	`<struct>`

The `<value>` children of `<array>` and `<struct>` elements themselves contain children of type `<string>`, `<array>` or `<struct>`.

4.2 API Mappings

The operation names in the SAMP hub and client abstract APIs (Sections 3.10 and 3.11) very nearly have a one to one mapping with those in the Standard Profile XML-RPC APIs. The differences are as follows:

1. The XML-RPC method names (i.e. the contents of the XML-RPC `<methodName>` elements) are formed by prefixing the hub and client abstract API operation names with "`samp.hub.`" or "`samp.client.`" respectively.
2. The `register()` operation takes the `samp.secret` value read from the lockfile (see Section 4.3) as an argument, and returns a new `private-key` string generated by the hub.
3. *All* other hub and client methods take the `private-key` as their first argument.
4. A new method, `setXmlrpcCallback()` is added to the hub API.

   - `setXmlrpcCallback(string private-key, string url)`

   This informs the hub of the XML-RPC endpoint on which the client is listening for calls from the hub. The client is not considered Callable intil it has invoked this method.
5. Another new method, `isAlive()` is added to the hub API. This may be called by registered or unregistered applications (as a special case the `private-key` argument may be omitted), and can be used to determine whether the hub is responding to requests. Any non-error return indicates that the hub is running.

The `private-key` string referred to above serves two purposes. First it identifies the client in hub/client communications. Some such identifier is required, since XML-RPC calls have no other way of identifying the sender's identity. Second, it prevents application spoofing, since the private key is never revealed to other applications, so that one application cannot pose as another in making calls to the hub.

The usual XML-RPC fault mechanism is used to respond to invalid calls as described in 3.12. The XML-RPC fault's `<faultString>` element should contain a user-directed message as appropriate and the `<faultCode>` value has no particular significance.

4.3 Lockfile and Hub Discovery

Hub discovery is performed by examining a lockfile in a well-known location. This has the consequence that in normal operation each user may run only one hub, and users do not share hubs. The name of the lockfile is "`.samp`" in the user's home directory. A "home directory" is a somewhat system-dependent concept: we define it as the value of the `$HOME` environment variable on Unix-like systems and as the value of the `%USERPROFILE%`

environment variable on Microsoft Windows[2].

The format of the file is given by the following BNF productions:

```
<file>          ::=   <lines>
<lines>         ::=   <line> | <lines> <line>
<line>          ::=   <line-content> <EOL> | <EOL>
<line-content>  ::=   <comment> | <assignment>
<comment>       ::=   "#" <any-string>
<assignment>    ::=   <name> "=" <any-string>
<name>          ::=   <token-string>
<token-string>  ::=   <token-char> | <token-string> <token-char>
<any-string>    ::=   <any-char> | <any-string> <any-char>
<EOL>           ::=   "\r" | "\n" | "\r" "\n"
<token-char>    ::=   [a-zA-Z0-9-_.]
<any-char>      ::=   [\x20-\x7f]
```

[?? are regular expressions OK in BNF? is there some other reasonably compact way of expressing this??]

The only parts which are significant to SAMP clients/hubs are (a) existence of the file and (b) `<assignment>` lines.

A legal lockfile must provide (in any order) unique assignments for the following tokens:

- `samp.secret` An opaque text string which must be passed to the hub to permit registration.
- `samp.hub.xmlrpc.url` The XML-RPC endpoint for communication with the hub.
- `samp.profile.version` The version of the SAMP Standard Profile implemented by the hub ("1.0" for the version described by this document).

it may optionally include other blank, comment or assignment lines, but tokens beginning "`samp.`" may only be assigned as described here.

The lockfile should normally be created with permissions which allow only its owner to read it. This provides a measure of security in that only processes with the same permissions as the hub process (hence presumably running under the same user ID) will be able to register with the hub, since only they will be able to provide the `samp.secret` required for registration.

[2] Note to Java developers: contrary to what you might expect, the `user.home` system property on Windows does *not* give you the value of `USERPROFILE`. See http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4787931.

Thus under normal circumstances all participants in a SAMP conversation can be presumed owned by the same user, and therefore not malicious.[3]

An example lockfile might therefore look like this:

```
# SAMP lockfile written 2008-29-02T17:45:01
# Required keys:
samp.secret=734144fdaab8400a1ec2
samp.hub.xmlrpc.url=http://andromeda.star.bris.ac.uk:8001/xmlrpc
samp.profile.version=1.0
# Info stored by hub for some private reason:
com.yoyodyne.hubid=c80995f1
```

The existence of the file MAY be taken (e.g. by a hub deciding whether to start or not) to indicate that a hub is running. However it is RECOMMENDED to attempt to contact the hub at the given XML-RPC URL (e.g. by calling `isAlive()`) to determine whether it is actually alive.

The hub discovery sequences are therefore as follows:

- Client startup:
  - Determine hub existence as above
  - If no hub, client MAY start its own hub
  - Acquire `samp.secret` value from lockfile
  - If pre-existing or own hub is running, call `register()` and zero or more of `setXmlrpcCallback()`, `setMetadata()`, `setMTypes()`
- Hub startup:
  - Determine hub existence as above
  - If hub is running, exit
  - Otherwise, start up XML-RPC server
  - Write lockfile with mandatory assignments including XML-RPC endpoint
- Hub shutdown:
  - Notify candidate clients that shutdown will occur
  - Remove lockfile (it is RECOMMENDED to first check that this is the lockfile written by self)
  - Shut down services

Hub implementations SHOULD make their best effort to perform the shutdown sequence above even if they terminate as a result of some error condition.

[3] Of course they may be owned by the same user and still be malicious, but in this case SAMP represents no additional security risk.

Note that manipulation of a file is not atomic, so that race conditions are possible. For instance a client or hub examining the lockfile may read it after it has been created but before it has been populated with the mandatory assignments, or two hubs may look for a lockfile simultaneously, not find one, and both decide that they should therefore start up, one presumably overwriting the other's lockfile. Hub and client implementations should be aware of such possibilities, but may not be able to guarantee to avoid them or their consequences. In general this is the sort of risk that SAMP and its Standard Profile are prepared to take — an eventuality which will occur sufficiently infrequently that it is not worth significant additional complexity to avoid. In the worst case a SAMP session may fail in some way, and will have to be restarted.

4.4 Examples

Here is an example in pseudo-code of how an application might locate and register with a hub, and send a message requiring no response to other registered clients.

```
# Read information from lockfile to locate and register with hub.
string hub-url = readFromLockfile("samp.hub.xmlprc.url");
string samp-secret = readFromLockfile("samp.secret");

# Establish XML-RPC connection with hub (uses some generic XML-RPC library)
xmlrpcServer hub = xmlrpcConnect(hub-url);

# Register with hub.
string private-key = hub.xmlrpcCall("samp.hub.register", samp-secret);

# Store metadata in hub for use by other applications.
map metadata = ("samp.name" -> "dummy",
                "samp.description" -> "Test Application",
                "dummy.version" -> "0.1-3");
hub.xmlrpcCall("samp.hub.setMetadata", private-key, metadata);

# Send a message requesting file load to all other registered clients,
# not wanting any response.
map loadParams = ("filename" -> "/tmp/foo.bar");
map loadMsg = ("mtype" -> "file.load",
               "params" -> "loadParams");
hub.xmlrpcCall("samp.hub.notifyAll", private-key, loadMsg);
```

```
# Unregister
hub.xmlrpcCall("samp.hub.unregister", private-key);
```

The first few XML-RPC documents sent over the wire for this exchange would look something like the following. The registration call from the client to the hub:

```
POST /xmlrpc HTTP/1.0
User-Agent: Java/1.5.0_10
Content-Type: text/xml
Content-Length: 189

<?xml version="1.0"?>
<methodCall>
  <methodName>samp.hub.register</methodName>
  <params>
    <param><value><string>734144fdaab8400a1ec2</string></value></param>
  </params>
</methodCall>
```

which leads to the response:

```
HTTP/1.1 200 OK
Connection: close
Content-Type: text/xml
Content-Length: 148

<?xml version="1.0"?>
<methodResponse>
  <params>
    <param><value><string>app-id:1a52fdf-2</string></value></param>
  </params>
</methodResponse>
```

The client might then declare its metadata: the response to this call has no useful content so can be ignored or discarded.

```
POST /xmlrpc HTTP/1.0
User-Agent: Java/1.5.0_10
Content-Type: text/xml
Content-Length: 596
```

```
<?xml version="1.0"?>
<methodCall>
  <methodName>samp.hub.setMetadata</methodName>
  <params>
    <param><value><string>app-id:1a52fdf-2</string></value></param>
    <param><value><struct>
      <member>
        <name>samp.name</name>
        <value><string>dummy</string></value>
      </member>
      <member>
        <name>samp.description</name>
        <value><string>Test application</string></value>
      </member>
      <member>
        <name>dummy.version</name>
        <value><string>0.1-3</string></value>
      </member>
    </struct></value></param>
  </params>
</methodCall>
```

Calls from the hub to the client are along similar lines.

5 MTypes: Message Semantics and Vocabulary

As stated earlier, a message contains an *mtype* string that defines the semantic meaning of the message, for example a request for another application to load a table. It also has other attributes and optional parameters to form the complete message that we will outline below. These messages also play a number of roles in a messaging system, although not every message is appropriate for every role. Below we will discuss the types of messages used in SAMP and see how specific *mtypes* are used.

In this section we will also discuss the form and required elements of a message, and the types of messages needed by many common desktop applications. This discussion is by no means complete; within the rules of how messages are defined here and the delivery mechanisms described, developers are free to create message types and usage patterns not detailed here. New messages affecting the messaging system itself (e.g. those used

in communications with the Hub or those needed to ensure interoperability such as a return status code) should be formalized for wider use later using a (TBD) procedure outlined below. Application message types, e.g. those that expose some functionality of a particular application, are entirely open-ended and require no formal process to be adopted by developers.

5.1 The MType of a Message

A key concept of a message is the *Mtype* attribute that defines the semantic meaning of the message. The concept behind the Mtype is similar to that of a UCD in that a small vocabulary is sufficient to describe the expected range of concepts required by a messaging system within the current scope of the SAMP protocol. As noted earlier, developers are free to introduce new MTypes for use within applications without restriction; new MTypes intended to be used for Hub messaging or other administrative purposes within the messaging system should be discussed within the IVOA for approval as part of the SAMP standard. The details and policy for adopting new standard MTypes are TBD.

5.1.1 The Form of an Mtype

Like a UCD, an Mtype is made up of *atoms*. These are not only meaningful to the developer, but form the central concept of the message. Because we wish to loosely couple the capabilities one application is searching for from the details of what another may provide, we don't create a rigorous definition of the *behavior* that an MType must provoke in a receiver. Instead, the Mtype defines a specific semantic message such as "display an image", it is up to the receiving application to determine how it chooses to do the display (e.g. a rendered greyscale image within an application or displaying the image in a web browser might both be valid for the recipient and faithful to the meaning of the message).

The ordering of the words in an Mtype should normally use the object of the message followed by the action to be performed (or the information about that object). For example, the use of "image.display" is preferred to "display.image" in order to keep the number of toplevel words (and thus message classes) like 'image' small, but still allow for a wide variety of messages to be created that can perform many useful actions on an image. If no existing MType exists for the required purpose, developers can agree to the use of a new Mtype such as 'image.display.extnum' if e.g. the ability to display a specific image extension number warrants a new Mtype.

The syntax of an MType is given by the following BNF:

```
<mchar>  ::= [0-9a-z] | "-" | "_"
<atom>   ::= <mchar> | <atom> <mchar>
<period> ::= "."
<mtype>  ::= <atom> | <atom> <period> <atom>
```

5.1.2 The Description of an MType

In order that senders and recipients can agree on what is meant by a given message, the meaning of an MType must be clearly documented. This means that for a given MType the following information must be available:

- The MType string itself.
- A list of zero or more parameters. For each one:
  - name
  - data type (`map`, `list` or `string` as described in 3.4) and if appropriate scalar sub-type (see 3.5)
  - meaning
  - whether it is REQUIRED or OPTIONAL
  - OPTIONAL parameters MAY specify what default will be used if the value is not supplied
- A list of zero or more returned values. For each one:
  - name
  - data type (`map`, `list` or `string` as described in 3.4) and if appropriate scalar sub-type (see 3.5)
  - meaning
  - whether it is REQUIRED or OPTIONAL
  - OPTIONAL return values MAY specify what default is intended if the value is not supplied
- A description of the meaning of the message. This should convey the semantic meaning of the message, e.g. that an event of some type has occurred, or that a specific request is is being made.

This is just the same information as one ought to supply for documentation of public interface method in a weakly-typed programming language.

Note that it is possible for the MType to have no returned values. This is actually quite common if the MType does not represent a request for data. It is not usually necessary to define a status-type return value (success or failure), since a sender which is interested in whether the message processing was successful can wait to see whether the response from the message is an error or not (the client `receiveResponse()` method has a `success` flag set).

So return values only need to be defined if there is data to return. If there is not, the response object will be a `map` with no entries.

As explained in Section 3.7, parameters and returned values which are not described in the MType may be passed as well as those which are. Clients which do not recognise these should usually ignore them.

5.2 Mtype Vocabulary

In the description below the *mtype* attribute of the message is constructed from the toplevel and secondary words. Any arguments required by the message MUST be encoded in the message *params* attribute. The Mtypes presented here are intended to be suggestions that we hope will be adopted by developers; concepts and Mtypes not shown here may also be used and may appear in later versions of this document if they become widely accepted.

mtype	args	returns	meaning
app			
event			
register	id		app has registered
unregister	id		app has unregistered
starting	id		app starts processing
stopping	id		app stops processing
mtype	mtypes id		app declares new mtype
metadata	meta id		app declares new metadata
status(??)			
ok			app executing normally
error			app encountered error
???			

5.2.1 Application Messages

These message types have the toplevel word "app". They are intended to convey some change in the state of a sending application.

5.2.2 Set/Get Messages

These message types have the toplevel word "set" or 'get". They are intended to provide a general mechanism for setting/getting values from a

remote application beyond the scope of what is available using either the Client or the Hub.

```
set
    mtype           mtype                   set list of mtypes
    metadata        meta                    set list of metadata
    param           param                   set parameter to value
                    value

get
    mtype                           mtypes      get list of supported mtypes
    metadata        param       value       get metadata item from client
    param           param       value
```

5.2.3 Status Messages

These message types have the toplevel word "`status`". They are intended to convey information about the state of an application, or its response to a Request message.

```
status
    ok                                  message processed normally
    invalid                             message invalid (badly formed)
    unknown                             message unknown (bad mtype)
    delivery        stat_str            delivery status
    progress
        percent     percent             percentage completed (float)
        timeLeft    time                est. time remaining (sec)
    error           err_str             error message
```

5.2.4 File Messages

These message types have the toplevel word "`file`". They are intended to perform some action on a file regardless of format.

```
file
    event
        load        filename            the 'filename' was loaded
        save        filename            the 'filename' was saved
    load            filename            load this file 'filename'
    save            filename            save to 'filename'
```

5.2.5 Image Messages

These messages have the toplevel word "`image`". They are intended to invoke operations on image data objects.

```
image
    event
        load        imname              the 'imname' was loaded
        save        imname              the 'imname' was saved
    load            imname              load image 'imname'
    save            imname              save image to 'imname'

    display         imname              display image in 'imname'
    panTo           x, y                pan display (arb coords)
        pixel       x, y                pan display to pixel coords
        sky         ra, dec             pan display to sky coords
    zoom            level               zoom to given level (+/-N level)
    highlight
        pixel       x, y                highlight point at pixel coords
        sky         ra, dec             highlight point at sky coords
```

5.2.6 Query Messages

These message types have the toplevel word "`query`". They are intended to perform some action on a query object. Queries will typically be ADQL, but no specification is made about the format of a query.

```
query
    exec
        adql        adql_str    result    execute the ADQL query
        sql         sql_str     result    execute the SQL query
        expr        expr_str    result    evaluate the expression
```

5.2.7 Spectrum Messages

These message types have the toplevel word "`spectrum`". They are intended to invoke operations on spectrum data objects.

```
spectrum
    event
        load        specname            the 'tblname' was loaded
            table   spectable           the given table was loaded
```

```
            image    specimage              the given image was loaded
        save         specname               the 'tblname' was saved
            table    spectable              table was saved to fname
                     fname
            image    specimage              image saved to imname
                     imname
    load             specname               load table 'tblname'
        table        spectable              table was loaded
        image        specimage              image was loaded
    save             specname               spectrum saved to 'specname'
        table        spectable              table saved to 'fname'
                     fname
        image        specimage              image saved to 'imname'
                     imname
```

5.2.8 Table Messages

These message types have the toplevel word "`table`". They are intended to invoke operations on table (any format) data objects.

```
table
    event
        load         tblname                the 'tblname' was loaded
        save         tblname                the 'tblname' was saved
    load             tblname                load table 'tblname'
    save             tblname                save table to 'tblname'

    highlight
        row          row                    highlight specified row
        col          col                    highlight specified column
        cell         row, col               highlight cell at position
    select
        row          row                    select (subset) named row
        col          col                    select (subset) named column
        rowList      rows                   select (subset) named rows
        colList      cols                   select (subset) named columns
```

5.2.9 URL Messages

These message types have the toplevel word "`url`". They are intended to perform some action on a URL.

```
url
    event
        load        url                 'url' was loaded
        save        url                 'url' was saved to 'filename'
                    filename
    load            url                 load url at 'url'
    save            url                 save 'url' to 'filename'
                    filename
```

5.2.10 Coordinate Messages

These message types have the toplevel word "`coord`". They are intended to provide a general method for using coordinates. Specific behavior depends on the application.

A Changes from PLASTIC document

In order to facilitate the transition from PLASTIC to SAMP protocol from applications' developers point of view, we summarize in this appendix the main changes - arranged in order of decreased importance - between the two documents, ie the main updates from PLASTIC to SAMP.

1. `Transport choice` - The main drawback of PLASTIC was the Java dependency for hub writers. In SAMP, Java RMI as transport mechanism has been dropped, which allows hubs to be written in virtually any language.

2. `Refactoring of registration API` - Registration, declaration of metadata, and declaration of understood messages are clearly separated steps in SAMP, whereas they were mixed in PLASTIC.

3. `Application identity`

TODO : complete this list. Not so easy to sort the importance of changes !!!

B SAMP/PLASTIC interoperability

This section will detail how interoperability between SAMP-compatible and PLASTIC-compatible clients can be achieved, using hubs able to translate PLASTIC messages into SAMP Mtypes.

References

[1] F. Bonnarel, P. Fernique, O. Bienaymé, D. Egret, F. Genova, M. Louys, F. Ochsenbein, M. Wenger, and J. G. Bartlett. The ALADIN interactive sky atlas. A reference tool for identification of astronomical sources. *A&AS*, 143:33–40, April 2000.

[2] U. Becciani, M. Comparato, A. Costa, C. Gheller, B. Larsson, F. Pasian, and R. Smareglia. VisIVO: an interoperable visualisation tool for Virtual Observatory data. *Highlights of Astronomy*, 14:622–622, 2007.

[3] `http://hea-www.harvard.edu/RD/xpa/`.

[4] J. Taylor, T. Boch, M. Comparato, M. Taylor, and N. Winstanley. PLASTIC - a protocol for desktop application interoperability. http://ivoa.net/Documents/latest/PlasticDesktopInterop.html, June 2006. IVOA note.

[5] http://www.json.org/.

[6] http://www.xmlrpc.com/.