International
Virtual
Observatory
Alliance

SAMP — Simple Application Messaging Protocol
Version 1.00

IVOA Working Draft 2008-06-25

This version:
http://www.ivoa.net/Documents/WD/App/SAMP-20080625.html

Latest version:
http://www.ivoa.net/Documents/latest/SAMP.html

Previous versions:

Working Group:
Applications

Editor(s):
T. Boch, M. Fitzpatrick, M. Taylor

Authors:
A. Allan (aa@astro.ex.ac.uk)
T. Boch (boch@astro.u-strasbg.fr)
M. Fitzpatrick (fitz@noao.edu)
L. Paioro (luigi@lambrate.inaf.it)
J. Taylor (jontayler@gmail.com)
M. Taylor (m.b.taylor@bristol.ac.uk)
D. Tody (dtody@nrao.edu)

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 since 2006 in over a dozen astronomy applications 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.

Status of this Document

This is an IVOA Working Draft for review by IVOA members and other interested parties. It is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to use IVOA Working Drafts as reference materials or to cite them as other than "work in progress".

Contents

1 Introduction

1.1 History

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

history and purpose see the IVOA Note *PLASTIC — a protocol for desktop application interoperability* [5] and the PLASTIC SourceForge site [6].

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 [7] 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. 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 producers and consumers 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 application 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 generate 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. 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:

Hub: A broker service for routing SAMP Messages.

Client: An application that talks to a Hub using SAMP. May be a Sender, Recipient, or both.

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. Contains an MType and zero or more named parameters. 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 contain returned values and/or error information. In the terminology of this document, a Response is not itself a Message. A Response is also known as a Reply in this document.

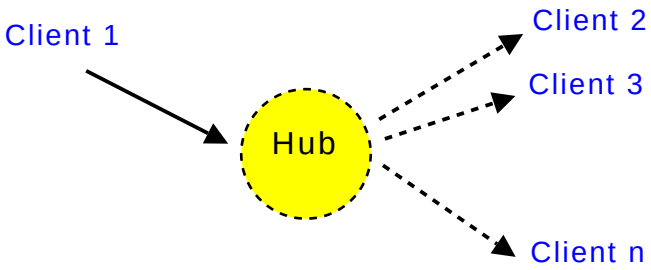

Figure 1: The SAMP hub architecture

MType: A string 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.

Subscription: 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.

Callable Client: A Client to which the Hub is capable of performing callbacks. Clients are not obliged to be Callable, but only Callable Clients are able to receive Messages or asynchronous Responses.

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 I/O 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. The former means that SAMP may not be suitable for extremely high throughput requirements; 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.

2.3 The Lifecycle 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 the 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.

By subscribing to the MTypes described in Section 5.4.1 a client may, if it wishes, keep track of the details of other clients' registrations, metadata and subscriptions.

2.4 The Lifecycle 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 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 and make the subscription information available

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.

The MTypes broadcast by the hub to inform clients of changes in its state are given in Section 5.4.1.

Readers should note that, given this scheme, race conditions may occur. A client might for instance try to register with a hub which has just shut down, or attempt to send to a recipient which has already unregistered. Specific profiles MAY define best-practice rules in order to best manage these conditions, but in general clients should be aware that SAMP's lack of guaranteed message delivery and timing means that unexpected conditions are possible.

2.5 Message Delivery Patterns

Messages can be sent according to three patterns, differing in whether and how 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, 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 situations such 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 Extensible Vocabularies

At several places in this document structured information is conveyed by use of a controlled but extensible vocabulary. Some examples are the application metadata keys (Section 3.6), message encoding keys (Section 3.8) and Standard Profile lockfile tokens (Section 4.3).

Wherever this pattern is used, the following rules apply. This document defines certain well-known keys with defined meanings. These may be optional or required as documented, but if present MUST be used by clients and hubs in the way defined here. All such well-known keys start with the string "`samp.`".

Clients and hubs are however free to introduce and use non-well-known keys as they see fit. Any string may be used for such a non-standard key, with the restriction that it MUST NOT start with the string "`samp.`".

The general rule is that hubs and clients which encounter keys which they do not understand should ignore them, propagating them to downstream consumers if appropriate. As far as possible, where new keys are introduced they should be such that applications which ignore them will continue to behave in a sensible way.

Hubs and clients are therefore able to communicate information additional to that defined in the current version of this document without disruption to those which do not understand it. This extensibility may be of use to applications which have mutual private requirements outside the scope of this specification, or to enable experimentation with new features. If the SAMP community finds such experiments useful, future versions of this document may bring such functionality within the SAMP specification itself by

defining new keys in the "`samp.`" namespace. The ways in which these vocabularies are used means that such extensions should be possible with minimal upheaval to the existing specification and implementations.

2.7 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 correspond 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 the question of "unsupported" languages does not arise.

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. A specific prescription will be described in Section 4.

3.3 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 indicate an empty value.

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 Remote Procedure Call (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. The practical requirement to transmit these types is addressed however by the next section.

3.4 Scalar Type Encoding Conventions

Although the protocol itself defines `string` as the only scalar type, some MTypes will wish to define parameters or return values which have non-string semantics, so conventions for encoding these as `string`s are 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 this section defines some commonly-used type encoding conventions.

We define the following BNF productions:

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

With reference to the above we define the following type encoding conventions:

- `<SAMP int> ::= [ <sign> ] <digits>`
  An integer value is encoded using its decimal representation with an optional 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> ] <float-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. There may be extensions to this list in future versions of this document.

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*".

3.5 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
3. obtain registration information

The registration information is in the form of a `map` containing data items which the client may wish to use during the SAMP session. The hub MUST fill in values for the following keys in the returned `map`:

`samp.hub-id` — The client ID which is used by the hub when it sends messages itself (rather than forwarding them from other senders). For instance, this ID will be used when the hub sends the `samp.hub.event.shutdown` message.

`samp.self-id` — The client ID which identifies the registering client.

These keys form part of an extensible vocabulary as explained in Section 2.6. In most cases a client will not require either of the above IDs for normal SAMP operation, but they are there for clients which do wish to know them. Particular Profiles may require additional entries in this map.

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

- supply the hub with metadata about itself, using the `declareMetadata()` call
- 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)

- inform the hub which MTypes it wishes to subscribe to, using the `declareSubscriptions()` call

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 might be the human-readable name of the application, a description and a URL for 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. These keys form the basis of an extensible vocabulary as explained in Section 2.6.

3.7 MType Subscriptions

As outlined above, an MType is a string which defines the semantics of a message. MTypes have a hierarchical form. Their syntax is given by the following BNF:

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

Examples might be "`samp.hub.event.shutdown`" or "`file.load`".

A client may *subscribe* to one or more MTypes to indicate which messages it is willing to receive. A client will only ever receive messages with MTypes to which it has subscribed. In order to do this it passes a subscriptions `map` to the hub. Each key of this map is an MType string to which the client wishes to subscribe, and the corresponding value is a map which may contain additional information about that subscription. Currently, no keys are defined for these per-MType maps, so typically they will be empty (have no entries). The use of a map here is to permit experimentation and perhaps future extension of the SAMP standard.

As a special case, simple wildcarding is permitted in subscriptions. The keys of the subscription map may actually be of the form `<msub>`, where

```
<msub>   ::= "*" | <mtype> "." "*"
```

Thus a subscription key "`file.event.*`" means that a client wishes to receive any messages with MType which begin "`file.event.`". This does not include "`file.event`". A subscription key "`*`" subscribes to all MTypes. Note that the wildcard "*" character may only appear at the end of a subscription key, and that this indicates subscription to the entire subtree.

More discussion of MTypes, including their semantics, is given in Section 5.

3.8 Message Encoding

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 and message identifiers) required to ensure proper delivery or handling.

The message can be encoded as a `map` with the following REQUIRED keys:

`samp.mtype` — A `string` giving the MType which defines the meaning of the message. The MType also, via external documentation, defines the names, types and meanings of any parameters and return values. MTypes are discussed in more detail in Section 5.

`samp.params` — A `map` containing the values for the message's named parameters. These give the data required for the receiver to act on the message, for instance the URL of a given file. The names, types and semantics of these parameters are determined by the MType. Each key in this map is the name of a parameter, and the corresponding value is that parameter's value.

These keys form the basis of an extensible vocabulary as explained in Section 2.6.

3.9 Response Encoding

A response is what may be returned from a recipient to a sender giving the result of processing a message (though in the case of the Notification delivery pattern, no such response is generated or returned). It may contain MType-specific return values, or error information, or both.

The response can be encoded as a `map` with the following keys:

`samp.status` (REQUIRED) — A `string` summarising the result of the processing. It may take one of the following defined values:

- `samp.ok`: Processing successful. The `samp.result`, but not the `samp.error` entry SHOULD be present.
- `samp.warning`: Processing partially successful. Both `samp.result` and `samp.error` entries SHOULD be present.
- `samp.error`: Processing failed. The `samp.error`, but not the `samp.result` entry SHOULD be present.

These values form the basis of an extensible vocabulary as explained in Section 2.6.

`samp.result` (REQUIRED in case of full or partial success) — A `map` containing the values for the message's named return values. The names, types and semantics of these returns are determined by the MType. Each key in this map is the name of a return value, and the corresponding value is the actual value. Note that for MTypes which define no return values, the value of this entry should be an empty `map`.

`samp.error` (REQUIRED in case of full or partial error) — A `map` containing error information. The following keys are defined for this map:

- `samp.errortxt` (REQUIRED) — A short string describing what went wrong. This will typically be delivered to the user of the sender application.
- `samp.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 `samp.errortxt` string, however it is undefined how this string should be handled when received.
- `samp.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.
- `samp.code` (OPTIONAL) — A string containing a numeric or textual code identifying the error, perhaps intended to be parsable by software.

These keys form the basis of an extensible vocabulary as explained in Section 2.6.

These keys form the basis of an extensible vocabulary as explained in Section 2.6.

3.10 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; any MType may legally be sent using any delivery 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.

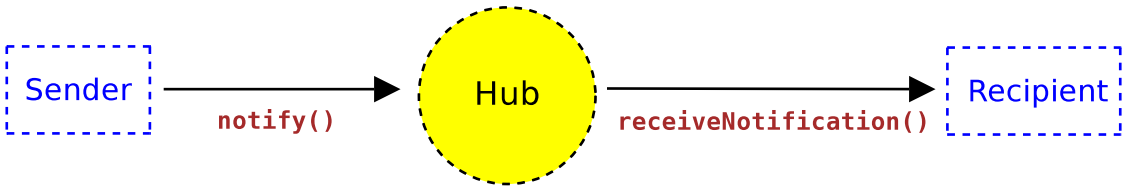

Figure 2: Notification pattern

Asynchronous Call/Response: In the asynchronous call pattern, *message tags* and *message identifiers* 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 a tag string of its own choice, *msg-tag*. In return it receives a unique identifier string, *msg-id*.
2. The hub forwards the message to the appropriate recipients, supplying along with the message an identifier string, *msg-id*.
3. Each recipient processes the message, and sends its response back to the hub along with the ID string *msg-id*.

4. Using a callback, the hub passes the response back to the original sender along with the ID string *msg-tag*.

The sender is free to use any value for the *msg-tag*. There is no requirement on the form of the hub-generated *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 *msg-tag* back with the response to the sender[1]. In most cases the sender will not require the *msg-id*, since the *msg-tag* is sufficient to match calls with responses. For this reason, the sender need not retain the *msg-id* and indeed need not wait for it, avoiding a hub round trip at send time. The only case in which the sender may require the *msg-id* is if it needs to communicate later with the recipient about the message that was sent, for instance as part of a progress report. 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.

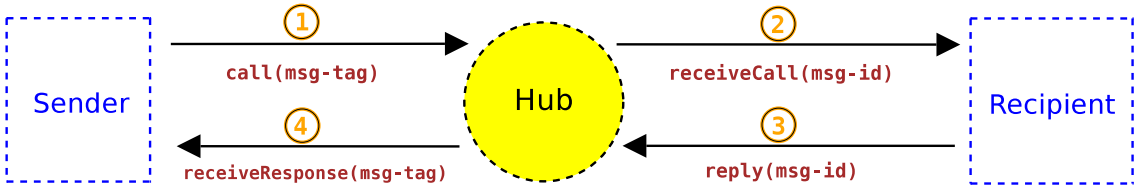

Figure 3: Asynchronous pattern

Synchronous Call/Response A synchronous utility method is provided by the hub, mainly for the convenience of 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, optionally specifying as well a maximum time it is prepared to wait. The sender's call blocks until a response is available.

[1]One way a hub might implement this is to generate *msg-id* by concatenating the sender's client id and the *msg-tag*. When any response is received the hub can then unpack the accompanying *msg-id* to find out who the original sender was and what *msg-tag* 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.

2. The hub forwards the message to the recipient, supplying along with the message an ID string, *msg-id*.
3. The recipient processes the message, and sends its response back to the hub along with the ID string *msg-id*.
4. The hub passes back the response as the return value from the original blocking call made by the sender. If no response is received within the sender's specified timeout the blocking call will terminate with an error. The hub is not guaranteed to wait indefinitely; it MAY in effect impose its own timeout.

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

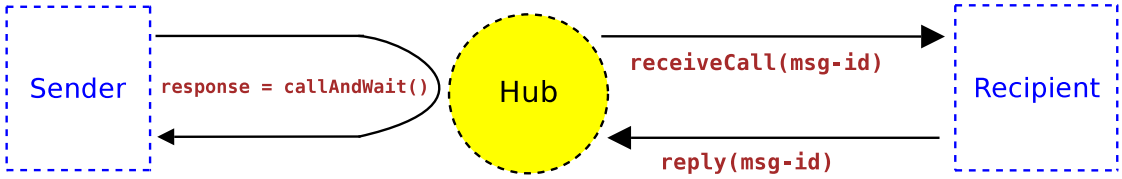

Figure 4: Synchronous pattern

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.

Although it is REQUIRED by this standard that client applications provide a Response for every Call that they receive, there is no way that the hub can enforce this. Senders using the Synchronous or Asynchronous Call/Response patterns therefore should be aware that badly-behaved recipients might fail to respond, leading to calls going unanswered indefinitely. The timeout parameter in the Synchronous Call/Response pattern provides some protection from this eventuality; users of the Asynchronous Call/Response pattern may or may not wish to take their own steps.

3.11 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. 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.

- `map reg-info = register()`
  Method called by a client wishing to register with the hub. The form of `reg-info` is given in Section 3.5. 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

- `declareMetadata(map metadata)`
  Method called by a client to declare its metadata. 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. The form of the `metadata` map is given in Section 3.6.

- `map metadata = getMetadata(string client-id)`
  Returns the metadata information for the client whose public id is `client-id`. The form of the `metadata` map is given in Section 3.6.

- `declareSubscriptions(map subscriptions)`
  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. The form of the `subscriptions` map is given in Section 3.7.

- `map subscriptions = getSubscriptions(string client-id)`
  Returns the subscribed MTypes for the client whose public id is `client-id`. The form of the `subscriptions` map is given in Section 3.7.

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

- `map client-subs = getSubscribedClients(string mtype)`
  Returns a map with an entry for all other registered clients which are subscribed to the MType `mtype`. The key for each entry is a subscribed client id, and the value is a (possibly empty) `map` providing further information on its subscription to `mtype` as described in Section 3.7. An entry for the caller is not included, even if it is subscribed. `mtype` may

not include wildcards.

- `notify(string recipient-id, map message)`
  Method called by a client wishing to send a notification to a given recipient. The form of the `message` map is given in Section 3.8.

- `notifyAll(map message)`
  Method called by a client wishing to broadcast a notification to all recipients. The form of the `message` map is given in Section 3.8.

- `string msg-id = call(string recipient-id, string msg-tag, map message)`
  Method called by a callable client wishing to send an asynchronous call to a given recipient. The form of the `message` map is given in Section 3.8.

- `string msg-id = callAll(string msg-tag, map message)`
  Method called by a callable client wishing to broadcast an asynchronous call to all recipients. The form of the `message` map is given in Section 3.8.

- `map response = callAndWait(string recipient-id, map message, string timeout)`
  Method called by a client wishing to make a synchronous call to a given recipient. The forms of the `message` and `response` maps are given in Sections 3.8 and 3.9. The `timeout` parameter is interpreted as a *SAMP int* giving the maximum number of seconds the client wishes to wait. If the response takes longer than that to arrive this method SHOULD terminate anyway with an error (not return a `response` indicating error). Any response arriving from the recipient after that will be discarded. If `timeout`$<= 0$ then no artificial timeout is imposed.

- `reply(string msg-id, map response)`
  Method called by a client to send its response to a given message. The form of the `response` map is given in Section 3.9.

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

3.12 Operations a Callable Client Must Support

This section lists the operations which the hub may call on a callable client. Note that callability is optional for clients; 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 may send messages using the Notify or Synchronous Call/Response patterns, but are unable to receive messages or to use Asynchronous Call/Response, since these operations rely on client callbacks from the hub.

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. The form of the `message` map is given in Section 3.8.

- `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. The form of the `message` map is given in Section 3.8.

- `receiveResponse(string responder-id, string msg-tag, map response)`
  Method used by the hub to dispatch to the sender the response of an earlier asynchronous call. The form of the `response` map is given in Section 3.9.

All these operations should complete quickly.

3.13 Error Processing

Errors encountered by clients when processing Call/Response-pattern messages themselves (in response to a syntactically legal `receiveCall()` operation) should be signalled by returning appropriate content in the response map sent back in the matching `reply()` call, as described in Section 3.9.

In the case of bad calls of the operations defined in Sections 3.11 and 3.12, for instance syntactically invalid parameters, hubs and clients should use the usual error reporting mechanisms of the transport protocol in use.

4 Standard Profile

Section 3 provides an abstract definition of the operations and data structures used for SAMP messaging. As explained in Section 2.7, 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 [7]. 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 using HTTP POST (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 3.3 map straightforwardly onto XML-RPC data types as follows:

SAMP type		XML-RPC element
`string`	—	`<string>`
`list`	—	`<array>`
`map`	—	`<struct>`

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

Note that other XML-RPC scalar types (`<i4>`, `double` etc) are not used; even where the semantic sense of a value matches one of those types it must be encoded as an XML-RPC `<string>`.

4.2 API Mappings

The operation names in the SAMP hub and client abstract APIs (Sections 3.11 and 3.12) 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 following form:
   - `map reg-info = register(string samp-secret)`

   The argument is the `samp-secret` value read from the lockfile (see Section 4.3). The returned `reg-info` map contains an additional entry with key `samp.private-key` whose value is a 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 unless and until it has invoked this method.
5. Another new method, `ping()` 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 determining 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 3.13. 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]
```

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).

These keys form the basis of an extensible vocabulary as explained in Section 2.6. Other blank, comment or assignment lines may be included as desired.

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

[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.

[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.

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 `ping()`) 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()`, `declareMetadata()`, `declareSubscriptions()`
- 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:
  - Remove lockfile (it is RECOMMENDED to first check that this is the lockfile written by self)
  - Notify candidate clients that shutdown will occur
  - 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.

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.
map reg-info = hub.xmlrpcCall("samp.hub.register", samp-secret);
string private-key = reg-info.getValue("samp.private-key");

# Store metadata in hub for use by other applications.
map metadata = ("samp.name" -> "dummy",
                "samp.description.text" -> "Test Application",
                "dummy.version" -> "0.1-3");
hub.xmlrpcCall("samp.hub.declareMetadata", 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 = ("samp.mtype" -> "file.load",
               "samp.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: 464

<?xml version="1.0"?>
<methodResponse>
  <params><param><value><struct>
    <member>
      <name>samp.private-key</name>
      <value><string>client-key:1a52fdf</string></value>
    </member>
    <member>
      <name>samp.hub-id</name>
      <value><string>client-id:0</string></value>
    </member>
    <member>
      <name>samp.self-id</name>
      <value><string>client-id:4</string></value>
    </member>
  </struct></value></param></params>
</methodResponse>
```

The client can 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: 600

<?xml version="1.0"?>
<methodCall>
  <methodName>samp.hub.declareMetadata</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.text</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>
```

The message itself is sent from the client to the hub as follows:

```
POST /xmlrpc HTTP/1.0
User-Agent: Java/1.5.0_10
Content-Type: text/xml
Content-Length: 523

<?xml version="1.0"?>
<methodCall>
  <methodName>samp.hub.notifyAll</methodName>
  <params>
    <param><value><string>app-id:1a52fdf-2</string></value></param>
    <param><value><struct>
      <member>
        <name>samp.mtype</name>
        <value>file.load</value>
      </member>
```

```
      <member>
        <name>samp.params</name>
        <value><struct>
          <name>filename</name>
          <value>/tmp/foo.bar</value>
        </struct></value>
      </member>
    </struct></value></param>
  </params>
</methodCall>
```

Again, there is no interesting response.

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. 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. 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.

5.1 The Form of an MType

MType syntax is formally defined in Section 3.7. 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.

5.2 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:

1. The MType string itself
2. A list of zero or more named parameters
3. A list of zero or more named returned values
4. A description of the meaning of the message

For each of the named parameters, and each of the returned values, the following information must be provided:

- name
- data type (`map`, `list` or `string` as described in 3.3) and if appropriate scalar sub-type (see 3.4)
- meaning
- whether it is OPTIONAL (considered REQUIRED unless stated otherwise)
- OPTIONAL parameters MAY specify what default will be used if the value is not supplied

Together, this is much the same information as should be given for documentation of a public interface method in a weakly-typed programming language.

The parameters and return values associated with each MType form extensible vocabularies as explained in Section 2.6, except that there is no reserved "`samp.`" namespace.

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-like return value (success or failure), since this information can be conveyed as the value of the `samp.status` entry in the call response as described in Section 3.9.

5.3 MType Vocabulary: Extensibility and Process

The set of MTypes forms an extensible vocabulary along the lines of Section 2.6. The relatively small number of MTypes in the "samp." namespace are defined in Section 5.4 of this document, but applications will need to use a wider range of MTypes to exchange useful information. Although clients are formally permitted to define and use any MTypes outside of the reserved "samp." namespace, for effective interoperability there must be public agreement between application authors on this unreserved vocabulary and its semantics. Since addition of new MTypes is expected to be ongoing, MTypes from this broader vocabulary will be documented in some kind of working list outside of this document to avoid the administrative overhead and delay associated with the IVOA Recommendation Track [8]. This working list may take the form of an IVOA Note.

Details of the procedure remain to be decided, but good practice for an application author who has the need to send or receive a message with particular semantics will be along the following lines:

1. Check the SAMP standard and the MTypes working list to see whether a suitable MType already exists
2. If not, suggest or solicit suggestions on the form of a suitable MType on the `apps-samp@ivoa.net` mailing list
3. If some agreement is reached, the new MType will be incorporated into the MTypes working list
4. In some cases, an equivalent MType in the `samp.` namespace may be defined, and be incorporated into a future version of the SAMP standard

5.4 Core MTypes

This section defines those MTypes currently in the `samp.` hierarchy. These are the "administrative"-type MTypes which are core to the SAMP architecture or widely applicable to SAMP applications.

5.4.1 Hub Administrative Messages

The following MTypes are for messages which SHOULD be broadcast by the hub in response to changes in hub state. By subscribing to these messages, clients are able to keep track of the current set of registered applications and of their metadata and subscriptions. In general, non-hub clients SHOULD NOT send these messages.

`samp.hub.event.shutdown`:

Arguments:

none

Return Values:

none

Description:

The hub SHOULD broadcast this message just before it exits. Hubs should make every effort to broadcast this message even in case of an exit due to an error condition.

`samp.hub.event.register`**:**

Arguments:

`id` (`string`) — Public ID of newly registered client

Return Values:

none

Description:

The hub SHOULD broadcast this message every time a client successfully registers.

`samp.hub.event.unregister`**:**

Arguments:

`id` (`string`) — public ID of unregistered client

Return Values:

none

Description:

The hub SHOULD broadcast this message every time a client unregisters.

`samp.hub.event.metadata`**:**

Arguments:

`id` (`string`) — public ID of client declaring metadata
`metadata` (`map`) — new metadata declared by client

Return Values:

none

Description:

The hub SHOULD broadcast this message every time a client declares its metadata. The `metadata` argument is exactly as passed using the `declareMetadata()` method.

`samp.hub.event.subscriptions`**:**

Arguments:

`id` (`string`) — public ID of subscribing client
`subscriptions` (`map`) — new subscriptions declared by client

Return Values:

none

Description:

The hub SHOULD broadcast this message every time a client declares its subscriptions. The `subscriptions` argument is exactly as passed using the `declareSubscriptions()` method, and hence may contain wildcarded MType strings.

5.4.2 Client Administrative Messages

The following messages are generic messages defined for client use.

`samp.app.ping`:

Arguments:

none

Return Values:

none

Description:

Diagnostic used to indicate whether an application is currently responding. No "status"-like return value is defined, since in general any response will indicate aliveness, and the normal `samp.status` key in the response may be used to indicate any abnormal state.

`samp.app.status`:

Arguments:

`txt` (`string`) — Textual indication of status

Return Values:

none

Description:

General purpose message to indicate application status.

`samp.app.event.shutdown`:

Arguments:

none

Return Values:

none

Description:

Indicates that the sending application is going to shut down. Note that sending this message is not a substitute for unregistering with the hub — registered clients about to shut down should always explicitly unregister.

`samp.msg.progress`:

Arguments:

`msgid` (`string`) — Message ID of a previously received message
`txt` (`string`) — Textual indication of progress
`percent` (`string`) — (OPTIONAL) SAMP float value giving the approximate percentage progress
`timeLeft` (`string`) — (OPTIONAL) SAMP float value giving the estimated time to completion in seconds

Return Values:

none

Description:

Reports on progress of a message previously received by the sender of this message. Such progress reports may be sent at intervals between the receipt of the message and sending a reply. Note that the `msg-id` of the earlier message must be passed to identify it — the sender of the earlier message (the recipient of this one) will have to have retained it from the return value of the relevant `call*()` method to match progress reports with requests.

A Changes between PLASTIC and SAMP

In order to facilitate the transition from PLASTIC to SAMP from an application developer's point of view, we summarize in this Appendix the main changes. In some cases the reasons for these are summarized as well.

Language Neutrality: PLASTIC contained some Java-specific ideas and details, in particular an API defined by a Java interface, use of Java RMI-Lite as a transport protocol option, and a lockfile format based on java Property serialization. No features of SAMP are specific to, or defined with reference to, Java (or to any other programming language).

Profiles: The formal notion of a SAMP Profile replaces the choices of transport protocol in PLASTIC. In practice since the Standard Profile is the only one currently defined, this means that XML-RPC is currently the only transport protocol which can be used to communicate in SAMP.

Nomenclature: Much of the terminology has changed between PLASTIC and SAMP, in some cases to provide better consistency with common usage in messaging systems. There is not in all cases a one-to-one correspondence betweeen PLASTIC and SAMP concepts, but a partial translation table is as follows:

PLASTIC	SAMP
message	MType
support a message	subscribe to an MType
synchronous request	synchronous call/response
asynchronous request	notification

MTypes: In PLASTIC message semantics were defined using opaque URIs such as `ivo://votech.org/hub/event/HubStopping`. These have now been replaced by a vocabulary of structured MTypes such as `samp.hub.event.shutdown`.

Asynchrony: Responses from messages in PLASTIC were returned synchronously, using blocking methods at both sender and recipient ends. As well as inhibiting flexibility, this risked timeouts for long processing times at the discretion of the underlying transport. The basic model in SAMP relies on asynchronous responses, though a synchronous façade hub method is also provided for convenience of the sender. Client toolkits may also wish to provide client-side synchronous façades based on fully asynchronous messaging.

Registration: In PLASTIC clients registered with a single call which acquired a hub connection and declared callback information, message subscriptions, and some metadata. In SAMP, these four operations have been decomposed into separate calls. As well as being tidier, this offers benefits such as meaning that the subscriptions and metadata can be updated during the lifetime of the connection.

Client Metadata: PLASTIC stored some application metadata (Name) in the hub and provided acess to others (Description, Icon URL, ...) using custom messages. SAMP stores it all in the hub providing better extensibility and consistency as well as improving metadata provision for non-callable applications and somewhat reducing traffic and burden on applications.

Named Parameters: The parameters for PLASTIC messages were identified by sequence (forming a list), while the parameters for SAMP MTypes are identified by name (forming a map). As well as improving documentability, this makes it much more convenient to allow for optional parameters or to introduce new ones. The same arrangement applies to return values.

Recipient Targetting: PLASTIC featured methods for sending messages to all or to an explicit list of recipients. In practice the list variants were rarely used except to send to a single recipient. SAMP has methods for sending to all or to a single recipient.

Typing: Data types in PLASTIC were based partly on Java and partly on XML-RPC types. There was not a one-to-one correspondence between types in the Java-RMI transport and the XML-RPC one, which encouraged confusion. Parameter types included integer, floating point and boolean as well as string, which proved problematic to use correctly from some weakly-typed languages. SAMP uses a more restricted set of types (namely string, list and map) at the protocol level, along with some auxiliary rules for encoding numbers and booleans as strings.

Lockfile: The lockfile in SAMP's standard profile is named `.samp`, its format is defined explicitly rather than with reference to Java documentation, and there is better provision for its location in a language-independent way on MS Windows systems. In many cases however, the same lockfile location/parsing code will work for both SAMP and PLASTIC except for the different filenames (".samp" vs. ".plastic").

Public/Private ID: In PLASTIC a single, public ID was used to label and identify applications during communications directed to the hub or to other applications. This meant that applications could easily, if they wished, impersonate other applications. SAMP's standard profile uses different IDs for public labelling and private identification, which means that such "spoofing" is no longer a danger.

Errors: SAMP has provision to return more structured error information than PLASTIC did.

Extensibility: Although PLASTIC was in some ways extensible, SAMP provides more hooks for future extension, in particular by pervasive use of the *extensible vocabulary* pattern.

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] `http://www.eurovotech.org/`.

[2] 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.

[3] 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.

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

[5] 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.

[6] `http://plastic.sourceforge.net/`.

[7] `http://www.xmlrpc.com/`.

[8] R. J. Hanisch et al. IVOA Document Standards. `http://www.ivoa.net/Documents/latest/DocStd.html`, 2003. IVOA Recommendation.