Simple Applications Messaging Protocol

Applications Working Group
IVOA Interop Meeting, Trieste, May 2008

`$Id: appsamp.tex,v 1.15 2008/05/16 22:52:17 mbt Exp $`

Plan For Sessions

Introduction

- History
- Summary of existing state
- Outline of SAMP

Outstanding Items

- Remaining open/contentious issues
- Work still to do

Future Plans

- Summarise work still required
- Commitments from document contributors
- Predictions from implementors
- Produce Roadmap

Demo

History

PLASTIC working and stable. . .

- A few working hub implementations
- Many compliant applications
- Popular with developers and users (including outside VO)

. . . but various things needed fixing

- Java-RMI dependency meant hubs could only be in Java
- Not generalisable for use in un-PLASTIC-like environments
- Various issues with the API discovered during use

SAMP intended to address these deficiencies

- Input from both PLASTIC and non-PLASTIC teams from IVOA
- Required to be "PLASTIC-like" in initial version, to build on existing base of developers and users
- Future versions (TBD) may generalise further, but underlying similarity will facilitate interoperability
  - ▷ different operating requirements, transport layers, application coupling models. . .
  - ▷ to some extent can address this by defining different *Profiles*

Current Status

SAMP document is fairly complete, "inWG"

- Lead authors (Boch, Fitzpatrick, M Taylor) worked together on initial draft (input from J Taylor, Tody)
- Circulated on `apps-samp` list since 30 April 2008
- Some items resolved by discussion on-list

Implementation

- We have two interoperating implementations!
  - ▷ Perl: hub implementation with test clients (Allan)
  - ▷ Java: SAMP functionality in SAMP (Boch)
- Different languages, different authors, few hitches, quick completion
  - ▷ Demonstrates that standard is close to complete and comprehensible
- Implementors note that standard is still in flux, so changes may be required

Some issues still to decide/resolve/complete

SAMP Document Overview

. . . over to Thomas

Next Steps

Plan for next six months:

1. Discuss outstanding issues here
2. Publish Working Draft <u>shortly</u> after this meeting (May/June)
3. Hub and client implementations
4. Revise draft in light of developer experiences
5. Produce Proposed Recommendation

Goals for this meeting:

- Reach consensus on open issues where possible . . .
- . . . but WD doesn't need to be final, so if necessary we can identify provisional/deferred decisions
- Criteria for published WD:
  - ▷ must be sufficient for people to use for write interoperable applications
  - ▷ *preferably* later changes will not invalidate or require major (any?) changes to software based on it

Outstanding Items

Several ISSUE and TODO items have been identified

- some flagged with initial draft (from earlier discussions between authors),
- some arose during discussion on list so far
- some only just introduced by me — (new)
  - ▷ apologies for short notice of these

Fall into several categories:

- Resolved ISSUEs
- Minor TODOs
- ISSUEs representing significant disagreement/uncertainty
- TODOs representing significant amounts of work

Discuss, resolve, assign responsibilities as appropriate

Resolved ISSUEs

Some items have been resolved by discussion on list already

- ISSUE: Message-id management
  Q: How are message identfiers assigned by clients and hubs?
  A: Client and hub can both choose their own free-form IDs.
- ISSUE: Lockfile in MS Windows
  Q: Where to write hub-discovery file on Windows OS?
  A: Use `%USERPROFILE%` environment variable.
- ISSUE: Difficulty of implementing synchronous call/response in hub
  Q: Implementing synchronous call in hub requires non-trivial IPC or threading — does this impose too heavy a burden on hub implementors?
  A: No.
- ISSUE: Call argument order
  Q: Arguments of some API methods look inconsistent.
  A: Rearrange them.

Minor TODOs

Small or uncontroversial items not yet addressed:

- mostly not done yet due to lack of time
- should be addressed before we issue a Working Draft
- can be handled by document authors
- noted here to make sure they get done

Items are:

- SAMP/PLASTIC comparison
  - ▷ appendix explaining the differences
- More examples *(is this required?)*
  - ▷ appendix with further examples of API use and/or XML-RPC communications
- Formal requirements for IVOA Recommendation Track document
  - ▷ "Document Status" section
  - ▷ Does LaTeX need fixing up? e.g. bibliography, `pdflatex` processing only?
- Proofreading etc. . .

Assign responsibilities and <u>firm</u> deadlines

ISSUE: Synchronous call timeout?

Should the synchronous call method incorporate a user-set timeout?

- Existing method is

  ```
      map response = callAndWait(string recipient-id, map message)
  ```

  could be

  ```
      map response = callAndWait(string recipient-id, map message,
                                 string timeout)
  ```

- `timeout` represents integer value in seconds; $<= 0$ means wait forever
- `timeout` should be advisory:
  - ▷ time out might occur later if hub is busy
  - ▷ time out might occur earlier if underlying protocol connection times out
- For:
  - ▷ Convenient for (e.g. script) applications which want a result but don't want to risk hanging
- Against:
  - ▷ Complicates hub implementation
  - ▷ Complicates hub API slightly
  - ▷ If you want more clever/flexible/robust invocation you can always use aysnchronous call/response

ISSUE: Rename setMetadata? (new)

Should hub method `setMetadata()` be renamed? **(my fault!)**

- Existing methods are

```
setMetadata(map metadata)
map metadata = getMetadata(string client-id)
```

- `setMetadata()` is not really the opposite of `getMetadata()`
- Rename instead:
  - ▷ `setSelfMetadata()`?
  - ▷ `declareMetadata()`? (which it was before I changed it)

Same applies to `setMTypes()` (but see ISSUE: Annotations)

ISSUE: getHubID/getSelfID

There are special client IDs which a client may want to know

(a) client's own public ID
  - ▷ needed only if client wants to send a message to itself?

(b) the ID used by the Hub (e.g. for sending hub stopping event messages)
  - ▷ needed to send a messsage to the hub as application (e.g. to get hub metadata like implementation name)
  - ▷ needed to identify if a given message comes from hub (why?)

Should it be possible for client to obtain these?

If so, how?

- Currently hub API has method `getHubID()` but not `getSelfID()`
- Could add `getSelfID()`
- Could remove `getHubID()` and require hub ID equal to fixed value (e.g. "0")
- Could have both returned at registration time:
  - ▷ register() call currently returns nothing (abstract API) or `private-key` (Standard Profile)
  - ▷ could return a map with keys `self-id`, `hub-id` (abstract API) and additionally `private-key` (Standard Profile)
  - ▷ allows extensibility to return other registration info too, if we think of other things
  - ▷ presumably remove hub `getHubID()` method in this case
- Or some combination?

ISSUE: Is a Response a Message?

A Message has an MType and some parameters

There are opposing views of what a Response to a Message is.

(1) It's like the return value of a function
- ▷ Form and meaning is as determined by original message MType
- ▷ Can only represent Success (plus return values as determined by message) or Failure (plus error information)
- ▷ The sender knows what it's expecting to get back, which makes it easy to handle
- ▷ (This is the model assumed by the current draft of the document)

(2) It is itself a new message with its own MType
- ▷ Response MType may be, e.g., `status.ok` or `status.error` or something else, with whatever additional params are appropriate
- ▷ More flexible possibilities for communications
- ▷ For instance `file.load` message might trigger a response `file.event.load` or `image.event.load` or `table.event.load` or . . .

ISSUE: Is a Response a Message? — *continued*

(1) It's like the return value of a function

```
send file.load message
get response
if (response is success)
   inform user of success
   get returned values of known form
         from response
   do something with returned values
else
   get error text etc from response
   inform user of error
```

May choose to send additional new messages (e.g. `table.event.load`) back to sender as a result of receiving this one.

(2) It is itself a new message with its own MType

```
send file.load message
get response
switch (response.MType)
   case status.ok:
      inform user of success
   case status.error:
      get error text etc from response
      inform user of error
   case file.event.load:
      get file.event.load-specific values from response
      inform user file was loaded
   case image.event.load:
      get image.event.load-specific values from response
      inform user image was loaded
   case table.event.load:
      get table.event.load-specific values from response
      inform user table was loaded
   default:
      inform user something happened .. but what?
```

- Response MType may not be one I have subscribed to
- What if response MType itself has a defined response?

ISSUE: MType Wildcarding

Should you be able to subscribe to multiple MTypes using wildcards?

- You can subscribe to `spectrum.load.votable` and `spectrum.load.fitstable`
- How about subscribing to `spectrum.load.*` which lets you receive the above as well as `spectrum.load...` messages not yet thought of (e.g. `spectrum.load.fitsimage`)
- Should `*` match multiple levels, e.g. does `spectrum.load.*` cover `spectrum.load.fitstable.extnum`?

Against:

- If you receive messages with MTypes you don't know about (haven't seen documentation for), how are you supposed to know how to process them?
  - ▷ You won't know what semantics the MType is supposed to represent
  - ▷ You won't know what parameters they have, or what return values you should send back
  - ▷ If you understand `spectrum.load.fitstable` you *might* be able to guess about `spectrum.load.votable` — but what about `spectrum.load.echelle`?
  - ▷ Even logging apps (which take no action) won't be able to return correct replies — would have to signal error for unknown MTypes.
- More work in hub

For:

- Useful for logging/monitor type applications
- . . . more?

ISSUE: Message Send Terminology

Delivery pattern and message type terminology needs to be clarified

- We have two apparently similar but orthogonal sets of concepts:
  - ▷ *Delivery Pattern*
    - Whether (and how) a sender wishes to receive a response from a given message sent
    - Decided by the sender when it sends the message
  - ▷ *MType Category*
    - Whether a message is the kind which means "I want you to do X" or "X has just happened"
    - Determined by the MType and how it is documented
- Confusion has arisen because
  typically you *will* want some response from "I want you do to X" and
  typically you *will not* want some response from "X has just happened".
- However, the rules of SAMP do not enforce these habits — either category of MType can be sent using any delivery pattern
- There is no genuine technical problem here, but the use of language (especially in API method names) has repeatedly caused confusion
- We need to decide once and for all how to label these things and adjust the API method names accordingly

ISSUE: Message Send Terminology — *continued*

Current usage in the draft document is as follows

- The terms used are:
  - ▷ Delivery Pattern:
    - ○ *Call/Response*: send mode where a response is required
    - ○ *Notification*: send mode where a response is not required
  - ▷ MType Category:
    - ○ *Request*: Mtype with semantics indicating "I want you to do X"
    - ○ *Event*: MType with semantics indicating "X has just happened"
- These appear in the normative parts of the document as:
  - ▷ Hub API methods `notify*()`, `call*()` and client API methods `receiveNotification()`, `receiveCall()`
  - ▷ MTypes `*.event.*`

  as well as in the descriptive text

Although internally consistent some people still believe this too confusing:

- the term "notify" suggests something which cannot have a response
- the term "call" sounds inappropriate for informing "X has happened"

ISSUE: Message Send Terminology — *continued*

Replace "Notify" and "Call" by "Send"?

- The term "send" has been proposed to be used for all delivery patterns
- Would require modifications of hub/client APIs (`notify()`, `call()` etc) to distinguish between *want response* and *do not want response*:
  - ▷ replace existing method names by variants of "send"
    - ○ a bit unwieldy:

      ```
      notify[All]() → sendVoid[All]() (or sendNotify[All]()?)
      call[All]()   → sendAsynch[All]()
      callAndWait() → sendSynch()
      ```
  - ▷ overload single `send()` method with different signatures
    - ○ not good for use with wire protocols or languages which do not support overloading
  - ▷ use single `send()` method with delivery pattern information in arguments
    - ○ Existing `notify()` and `call()` methods have different signatures, so can't just amalgamate by adding a new `wantResponse` argument
    - ○ Could do it by moving the `wantResponse` argument *inside* the `message` map argument. Less explicit what's going on?

ISSUE: Message Send Terminology — *continued*

Summary of possibilities:

1. Do nothing
   - ▷ Leave `notify()`/`call()` methods as they are
   - ▷ "Event" and "Request" are terms only used in discussion of MTypes
   - ▷ Perhaps work harder to clarify the issues in the text
2. Avoid discussion of MType categories altogether
   - ▷ Leave `notify()`/`call()` methods as they are
   - ▷ Remove general discussion of distinct "Event"/"Request" MType semantics (though `*.event.*` MTypes still exist)
3. Use overloaded `send()`
   - ▷ Replace `notify()`/`call()` methods by overloaded `send()` method
   - ▷ "Event", "Request", "Notify" and "Call" may be used in discussion of MTypes
4. Use `send()` with delivery pattern flag inside message
   - ▷ Replace `notify()`/`call()` methods by single `send()` method with `wantReply` flag encoded within `message` argument envelope
   - ▷ "Event", "Request", "Notify" and "Call" may be used in discussion of MTypes
5. Use `sendSomething()`
   - ▷ Rename methods `notify()`/`call()` as `sendVoid()`/`sendAsync()` (or something)
   - ▷ "Event", "Request", "Notify" and "Call" may be used in discussion of MTypes

ISSUE: Envelope/Body Distinction

Do we need to make a distinction between the Envelope and Body of a Message?

- It has been suggested that Body/Envelope would be a useful distinction
- As the API is currently set up, it is not used:
  - ▷ Hub send methods look like, e.g.:
    ```
    call(string recipient-id, string msg-id, map message)
    ```
    where `message` contains MType and parameters.
  - ▷ So addressing and payload information is split up as:
    - `recipient-id`: address of recipient — argument of send method
    - `msg-id`: tag to identify message — argument of send method
    - `message.mtype`: message semantic label — encoded in `message` argument
    - `message.params`: message parameters — encoded in `message` argument
    - `message.*`: other semantic information (currently undefined) — encoded in `message` argument
  - ▷ Information giving the semantics of the message is all passed in the `message` argument
  - ▷ Information required for routing/delivering the message is explicit in send method arguments
  - ▷ (MType is a bit of both however, since it also influences delivery according to whether a recipient is subscribed)
  - ▷ So the message object itself is effectively all body, no envelope
  - ▷ The motivation for this split is partly so that if SAMP changes to permit different ways of delivering messages, the `message` object which arrives at the client will still look the same
- Is this satisfactory, or should it be arranged differently?

ISSUE: Response Encoding (new)

- Currently processing success/failure flag is passed separately from response object, response object contains *either* result *or* error info
  - ▷ Asynchronous Call/Response:
    - `receiveResponse(string responder-id, string msg-id, string success, map response)`
    - for successful processing, `success="1"`), `response` contains data as defined by MType
    - in case of error, `success="0"`, `response` contains error information in a standard form
  - ▷ Synchronous Call/Response:
    - `map response = callAndWait(string recipient-id, map message)`
    - for successful processing, `response` contains data as defined by MType
    - in case of error, the invocation itself results should fail in a protocol-dependent way

- Would it be better for response object to contain success flag?
  - ▷ Asynchronous Call/Response:
    - `receiveResponse(string responder-id, string msg-id, map response)`
  - ▷ Synchronous Call/Response:
    - `map response = callAndWait(string recipient id, map message)`
  - ▷ In all cases (synch/asynch and success/error) `response` map has same form, with keys:
    - `success`: "1" for success, "0" for error
    - `result`: return values as defined by MType; SHOULD be absent in case of error
    - `error`: error information in standard form; SHOULD be absent in case of success

ISSUE: Response Encoding — *continued*

- Advantages of success flag contained in response object:
  - ▷ More consistent handling between synchronous and asynchronous cases
  - ▷ All semantic information from result of message is in one place (more consistent with the way messages are sent)
  - ▷ Possibility of extending response map to contain more information in future
  - ▷ It's possible to return partially filled in results even in case of error

- Advantages of success flag transmitted separately from response object:
  - ▷ Success flag is guaranteed to be present.
  - ▷ Easier for clients to determine success/failure.
  - ▷ It's impossible to return partially filled in results even in case of error.

- I favour change (success flag in response), and I specified those methods in the first place
  - ▷ looks like poor initial design on my part — sorry

ISSUE: Annotations

Annotations permit dynamic (run-time) refinement of MType semantics

- Brief history
  - Annotations in PLASTIC
    - Retrofitted at slight cost to message syntax tidiness
    - Demonstrated to do what they were supposed to do
    - Not widely used
  - Annotations in SAMP
    - Present in early drafts of SAMP document
    - Removed before mailing list circulation, since concepts not well integrated into the rest of the document
  - A really neat idea, or completely unnecessary and misguided, according to who you talk to
  - Widely misunderstood
- Transparent yet complete explanation of the exact what, why and how of Annotations in ten words or less:

  omitted due to lack of space in the margin
- Possible ways forward:
  - Reinstate section from early drafts, with appropriate required modifications to API and text
  - Abandon idea altogether
  - Omit for now, but modify API in such a way that they remain a possibility

ISSUE: Annotations — *continued*

Compromise: how to leave door open for Annotations

- Change to API
  - ▷ Currently:
    - ○ A client's subscriptions are represented as a `list` of MTypes

      ```
      setMTypes(list mtypes)
      list mtypes = getMTypes(string client-id)
      ```
  - ▷ Proposed:
    - ○ A client's subscriptions are represented as a `map` in which the keys are Mtypes

      ```
      setSubscriptions(map subscriptions)
      map subscriptions = getSubscriptions(string client-id)
      ```
    - ○ The values of these keys are undefined (may be empty)
    - ○ This provides a place which annotation information could be stored, if we decide we want it
  - ▷ Notes
    - ○ The modified API is hardly any more complicated to use
    - ○ It's set up so that Annotation-aware and Annotation-unaware applications can interoperate without either needing to know the difference
    - ○ This introduces flexibility which could be used in future for other possibilities (e.g. finer-grained subscriptions based one parameter values??)
- How to proceed if this is adopted
  - ▷ Application developers can experiment if they wish (via discussions on `apps-samp` list)
  - ▷ If annotations look useful, we can reconsider introducing them to doc before PR stage

ISSUE: Rationalise Reserved Words? (new)

- Several places in the document have a vocabulary of reserved words (mostly map keys):
  - ▷ Application metadata keys (`samp.name`, `samp.icon.url`, . . .)
  - ▷ Message content encoding keys (`mtype`, `params`)
  - ▷ Response content encoding keys (`errortxt`, `usertxt`, `code`, . . .)
  - ▷ Standard profile lockfile tokens (`samp.secret`, `samp.hub.xmlrpc.url`, . . .)
  - ▷ *possibly more arising from discussions above?*

- All these vocabularies are individually documented as being extensible:
  - ▷ Undefined keys (ones not described in the SAMP document) MAY be used in these contexts
  - ▷ Applications coming across keys they don't understand should generally ignore them
  - ▷ This means that applications can experiment with new features in such a way that the API doesn't need to change and they don't break existing interoperability
  - ▷ If such features are agreed to be useful, they can be introduced into future versions

- Some use "`samp.`" prefix to mark reserved namespace, others don't (more or less at random)

- Should we rationalise so that all do or all don't?

- If so, which is better?
  - ▷ Using `samp.` prefix is safer — can be sure of avoiding accidental clashes
  - ▷ But using a flat namespace (no `samp.`) makes it easier to adopt de facto usages into the standard

ISSUE: HTTP/JSON? (new)

Should we add an HTTP-GET-based interface alongside the XML-RPC one?

- What
  - ▷ Standard Profile would require hubs to provide an interface based on HTTP GET and JSON as well as the existing XML-RPC one
  - ▷ JSON (http://www.json.org/, RFC 4627 — 10 pages!)
    - JavaScript Object Notation — but in no way Java/JavaScript specific!
    - Simple prescription for encoding structured data (maps, lists) in strings
  - ▷ Clients can choose whether they use XML-RPC or HTTP/JSON flavour
  - ▷ Only requirements for use are:
    - HTTP GET: very widely available, often without requiring external libraries
    - JSON parser: libraries available for many languages, but very feasible to write your own/parse by hand
  - ▷ Only certain SAMP operations would be available — no Callable clients ⇒ no asynchrnous calls or MType subscriptions

ISSUE: HTTP/JSON? — *continued*

Should we add HTTP/JSON to Standard Profile?

- For
  - Makes limited use of SAMP *really* easy
  - Makes limited use of SAMP possible from restrictive/primitive environments (e.g. shell scripts, IDL, . . .)
  - Useful for, e.g., doing something very simple like broadcasting a load table message
- Against
  - Complicates (Standard Profile part of the) specification
  - More work for hub implementors
  - More choices of wire protocol means more things to go wrong, more untested code in hubs
  - Proliferating wire protocols willy-nilly is a bad thing
  - Applications using this can't use all SAMP capabilities (e.g. asynchronous messaging)
- Only worth doing if it makes worthwhile use cases *significantly* easier
  (e.g. enables SAMP use from places it would otherwise be impractical)

ISSUE: Rename Standard Profile PLASTIC? (new)

Should we retain PLASTIC name for PLASTIC-like parts of SAMP?

- What
  - ▷ SAMP covers messaging architecture designed to be extended in future for different messaging requirements
  - ▷ "Standard Profile" describes XML-RPC bindings, hub discovery using lockfiles etc
  - ▷ SAMP + Standard Profile is by design PLASTIC-like
  - ▷ We could, e.g., label the Standard Profile the PLASTIC Profile or PLASTIC v2 or SAMP/PLASTIC.
  - ▷ SAMP itself remains the overall label for the more general/generalisable messaging system
- For:
  - ▷ The PLASTIC "brand" is quite well known and popular, among developers and even (non-VO) astronomers.
  - ▷ Starting with a new name may be hard to sell to existing users.
- Against:
  - ▷ Could result in confusion about compatibility etc
  - ▷ May risk underselling the differences/improvements represented by SAMP over PLASTIC

TODO: MType vocabulary

. . . over to Mike

Implementations

Coming along nicely!

Next steps following WD publication (May/July):

- Existing implementations updated as required
  - ▷ Perl hub — AA
  - ▷ Aladin — TB
- New hub implementations/infrastructure
  - ▷ Java hub — MT
  - ▷ Java client toolkit — MT
  - ▷ Hub test suite? — MT?
  - ▷ . . . other people's plans?
- Uptake in existing applications
  - ▷ TOPCAT — MT
  - ▷ SPLAT — MT
  - ▷ VODesktop? — MT/AG
  - ▷ GAIA? — MT
  - ▷ . . . other people's plans?

PLASTIC/SAMP migration

Hopefully existing PLASTIC tools will start to move to SAMP.

- Do we need to be proactive about this?
  - ▷ "Why should I recode my PLASTIC-speaking app to use SAMP?"
  - ▷ "What happens if I don't?"

How do we manage the transition?

- Danger of alienating existing PLASTIC users
- Would be nice if nothing/not much stopped working while applications migrate
- Do we need to take special steps?
  - ▷ Attempt to fix crossover date from PLASTIC to SAMP?
    - ○ Difficult to organise — probably not practical
  - ▷ Existing PLASTIC applications encouraged to speak both PLASTIC and SAMP?
    - ○ Temporary measure
    - ○ Not ideal for developers, but would probably work best
  - ▷ PLASTIC/SAMP bridge?
    - ○ Temporary measure
    - ○ A SAMP hub implementation could also function as PLASTIC hub, translating messages between the two
    - ○ Or separate daemon could do a similar job (hence work with any SAMP hub)
    - ○ Translation unlikely to be perfect (PLASTIC msg $\leftrightarrow$ SAMP Mtype correspondance required)
    - ○ Could probably be made to work reasonably well??