PLASTIC – a protocol for desktop application interoperability

Version 1.0

IVOA Note 1st June 2006

Previous version(s):
None

Authors:
Thomas Boch
Marco Comparato
John Taylor
Mark Taylor
Noel Winstanley

Editor:
John Taylor

Abstract

PLASTIC (Platform for Astronomy Tool InterConnection) is a protocol for communication between client-side astronomy applications. It is intended to be simple for application developers to adopt and is easily extended. Through PLASTIC applications can do tasks such as instruct each other to load VOTables, highlight a subset of rows or load an image of a particular area of sky. Although such operations are quite simple, they can significantly enhance an application's power and usability. The philosophy is that the astronomer should
have a suite of interoperating tools at his disposal, each of which does one thing well and which can be composed according to his particular needs.

This Note documents the PLASTIC protocol version 0.5 and does not preclude changes to the specification in the future.

**Status of This Document**

This is an IVOA Note. The first release of this document was 1st June 2006.

*This is an IVOA Note expressing suggestions from and opinions of the authors. It is intended to share best practices, possible approaches, or other perspectives on interoperability with the Virtual Observatory. The document describes a standard created by the VOTech Consortium and its partners and is not a formal IVOA standard.*

A list of current IVOA Recommendations and other technical documents can be found at http://www.ivoa.net/Documents/.

**Acknowledgments**

The authors would like to thank Sebastien Derriere for inventing the acronym “PLASTIC”, thereby keeping the VOTech mailing lists sufficiently busy that we were able to quietly work on the protocol.

**Contents**

**Table of Contents**

1 Introduction .......................................................................................................................... 3
  1.1 History ......................................................................................................................... 3
  1.2 Requirements .............................................................................................................. 4
  1.3 Example use cases ...................................................................................................... 4
2 Architecture Overview ...................................................................................................... 6
  2.1 Plastic Messages ......................................................................................................... 6
    2.1.1 Example Messages ............................................................................................ 7
Appendix A - The Plastic Protocol version 0.5 ..................................................................... 9
  Definitions ......................................................................................................................... 9
  Summary of how messages are routed ............................................................................ 9
  Choice of communication protocols .............................................................................. 10
1 Introduction

The capabilities of key astronomy desktop tools are being continually enhanced with new features. While this is of great benefit of astronomers, there are some limitations to this “bigger is better” approach. Inevitably there will be some overlap between the tools’ feature sets as users demand capabilities from other applications in their own favourite tool, leading to a duplication of effort within the astronomical software community. As the tools become more powerful, their resource footprint usually expands and their increased complexity may make them more difficult to maintain and use.

The alternative and complementary approach is to encourage collaboration between tools, each a specialist in a particular task. This approach can be compared to the Unix philosophy of “Write programs that do one thing and do it well” and enables the astronomer to assemble a suite of tools according to his own requirements.

This paper discusses work that has taken place in the VOTech consortium [votech] to address interoperability between desktop applications through a communication protocol known as PLASTIC. This protocol not only allows applications to share data, but also enables cross-application linked views.

1.1 History

Data exploration using different linked views of the data is not a new idea. For example, the Mirage [Mirage] and xmdv [xmdv] applications support several visualization methods allowing the user to explore data simultaneously in
different ways. While Mirage and xmdv have the different views built in, CDS has taken a slightly different approach with Aladin [Bonnarel et al 2000]. Aladin exposes a publicly documented interface (“VOApp”) that makes it easy for third-parties to write plug-ins that expand its capabilities. This capability has been exploited by VOIndia’s VOPlot [VOPlot] and CADC’s Octet [Octet] applications. The symmetry of the interface means that such plug-ins are peers of the Aladin application itself, being able to send as well as receive events. However, plug-ins are restricted to being Java applications that can run in the same JVM. The Aladin and VisIVO [VisIVO] teams overcame this constraint by enabling VisIVO, a C++ application, to control Aladin through the latter’s scripting interface, but the architecture is no longer symmetrical and Aladin is unable to control VisIVO in return.

At the 3rd SC4DEVO workshop [Mann] it was decided to generalise the VisIVO/Aladin interoperability work to arbitrary applications and the technology developed for the Astro Runtime [Winstanley et al 2006] offered a means to do it. The PLASTIC protocol that has grown out of this meeting has, at the time of writing, been incorporated into more than a dozen applications written in Java, C++, Python, Perl, Tcl and JavaScript/html including Topcat (AstroGrid/Starlink), Aladin (CDS), VisIVO (INAF), VOSpec (ESA), AstroWeka (VOTech) and Reflex (SAMPO).

1.2 Requirements
There were three main requirements for the PLASTIC protocol: it should be platform and language-neutral, extensible, and above all simple to learn and use. Language neutrality was assured by using XML-RPC [xmlrpc] as the underlying protocol, for which there are libraries for all major languages. In contrast to the Aladin scripting interface, which has a well-defined set of operations, PLASTIC is based on messaging, allowing new messages to be defined and the protocol extended as previously unforeseen uses arise. Finally, the protocol has been kept simple as it was important for application authors to adopt it quickly so that work could concentrate on the more difficult task of defining precisely how applications should interoperate and what messages were required. PLASTIC does not pretend to be a general-purpose messaging system, and does not include features such as guaranteed delivery or encryption.

1.3 Example use cases
There are many Use Cases that simply rely on the ability of one application to instruct another to load data from a URL:

1. User searches for catalogue data using AstroScope. Once VOTables have been located, User wishes to show the data in Topcat.

---

1 For an comparison of the Aladin plug-in interface and the PLASTIC protocol described here see [Fernique 2006].
2. User runs an Anomaly Detection algorithm that returns a VOTable ordered according to how unusual the objects are. User wishes to visualize the VOTable in VisIVO, with the objects coloured according to probability.

3. User is browsing VOSpace, locates a FITS file and wishes to view it in Aladin.

These Use Cases can be satisfied through the use of each application’s load/save functionality, but by using PLASTIC messaging, the operation becomes seamless.

Other Use Cases require applications to be able to define subsets of rows or columns:

4. User is viewing catalogue data in Aladin and wishes to locate the 10 reddest objects. Aladin sends the table to Topcat [Topcat], where the User adds a synthetic B-R column, sorts the table, and selects the 10 reddest objects. Topcat instructs Aladin to highlight the objects.

5. User is working on data in Topcat but wants to see a "4D" plot. Topcat sends the data to VisIVO and the user maps 3 attributes onto (x,y,z) and the fourth onto colour. Exploring the data in VisIVO finds a group of outliers. These outliers are selected in VisIVO, and VisIVO instructs Topcat to create a row subset of these outliers for further analysis.

6. User opens a VOTable in Eirik [Eirik] and uses Eirik to select the 4 most “interesting” attributes. Eirik instructs AstroWeka [AstroWeka] to open the VOTable and perform a clustering algorithm on those 4 attributes. AstroWeka then instructs VisIVO to display the clusters in different colours.

Appendix E contains a complete example of Plastic Messaging between Aladin and Topcat and includes screenshots that illustrate the powerful interaction that can be achieved.
2 Architecture Overview

The following paragraphs give an informal description of the PLASTIC protocol. Normative text can be found in Appendices A, B, C.

The two key concepts in PLASTIC are the Plastic Hub and Plastic Messages. PLASTIC is a specification for a simple messaging system: applications send each other messages requesting certain actions via the Plastic Hub, which routes them to their destinations. The advantage of this publish-subscribe architecture is that individual applications need only understand one of the Hub's communication protocols, and the Hub is responsible for any translation required. Furthermore, applications can discover other applications and their capabilities by interrogating the Hub.

The messages themselves consist of strings agreed by the application developers - some of them can be seen in table 1. Each message may be accompanied by several arguments, and may return a result. Again, the arguments and return values are determined by the application developers defining the message, they are not described by the PLASTIC specification, and (with a small number of exceptions) the Plastic Hub has no understanding of their meaning.

The PLASTIC messaging system is intended to be platform neutral and as such is available via a number of different communication protocols. As of the current version, XML-RPC and Java-RMILite are supported, but others such as HTTP and CORBA could be added if there is demand.

XML-RPC libraries are available for most major languages and PLASTIC functionality has been implemented within applications [PlasticApps] written in Java, C++, Python, Tcl and Perl, amongst others.

Java clients will usually communicate with the hub using Java-RMILite since this makes PLASTIC particularly straightforward to use, but XML-RPC can be used instead if preferred.

PLASTIC is a specification, not an application. There are Plastic Hubs embedded in Topcat and the Astro Runtime [AstroRuntime], the latter being the reference implementation. Work is currently under way to embed Plastic Hubs in Aladin and Octet.

2.1 Plastic Messages

Requests are sent to other applications by means of messages, which may include a number of arguments and may also return a value. The ability to return a value means that PLASTIC is not a pure messaging system, but it simplifies short information requests such as “send me your name”, and also allows clients such as scripting environments that cannot receive messages to receive information from other applications.

Application developers are free to extend PLASTIC by defining new messages, but a set of core messages has been defined (see below).
Plastic message strings are arbitrary, opaque URIs. So far, Plastic application developers have elected to use URIs that are IVORNS as this allows their registration in IVOA-standard registries, bringing the following benefits:

- Guaranteed uniqueness
- A central reference where the meaning of a message can be stored
- Facilitating the automatic discovery of applications with particular capabilities

### 2.1.1 Example Messages

This document describes the Plastic Messaging protocol, rather than the semantics of the messages, which are the responsibility of application developers. Nevertheless, some example Plastic Messages are listed below to illustrate the protocol. A full description of the Plastic Messages currently in use is the subject of a separate IVOA note [messages].

All Plastic messages are optional, and applications can specify which messages they can understand and want to receive. However, many applications will support the following core messages:

<table>
<thead>
<tr>
<th>Message ID</th>
<th>Args</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ivo://votech.org/test/echo</td>
<td>text: String</td>
<td>response: String</td>
<td>Echos text (used for troubleshooting and pinging an application).</td>
</tr>
<tr>
<td>ivo://votech.org/info/getIVORN</td>
<td>void</td>
<td>id: String</td>
<td>Returns the tool's registerable ID e.g. ivo://org.astrogrid/ar</td>
</tr>
<tr>
<td>ivo://votech.org/info/getName</td>
<td>void</td>
<td>name: String</td>
<td>Get a human-readable name e.g. Topcat</td>
</tr>
<tr>
<td>ivo://votech.org/info/getDescription</td>
<td>void</td>
<td>name: String, version: String</td>
<td>Get a human-readable description of the application.</td>
</tr>
<tr>
<td>ivo://votech.org/info/getVersion</td>
<td>void</td>
<td>version: String</td>
<td>Get the version of PLASTIC supported.</td>
</tr>
<tr>
<td>ivo://votech.org/info/getIconURL</td>
<td>void</td>
<td>url: String</td>
<td>Get a URL to an icon for the application.</td>
</tr>
<tr>
<td>ivo://votech.org/hub/event/HubStopping</td>
<td>void</td>
<td>void</td>
<td>Sent by the hub before it shuts down</td>
</tr>
<tr>
<td>ivo://votech.org/hub/event/Application Registered</td>
<td>id: String</td>
<td>void</td>
<td>Sent by the Hub when an application registers</td>
</tr>
<tr>
<td>ivo://votech.org/hub/event/Application Unregistered</td>
<td>id: String</td>
<td>void</td>
<td>Sent by the Hub when an application registers</td>
</tr>
</tbody>
</table>

### Table 2 – Example Messages

Some other messages currently in use are:

<table>
<thead>
<tr>
<th>Message ID</th>
<th>Args</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ivo://votech.org/votable/loadFromURL</td>
<td>url: String, id: String</td>
<td>success: Boolean</td>
<td>Load a VOTable from the supplied URL.</td>
</tr>
<tr>
<td>ivo://votech.org/votable/showObjects</td>
<td>id: String, rows: String</td>
<td>success: Boolean</td>
<td>Highlight/select the</td>
</tr>
</tbody>
</table>
The messages in table 2 will be of use to many astronomical applications, but developers are free to create more specialized messages. For example, the VOEvent community [estar] is defining messages specific to the notification of events. In general, application developers are advised to discuss their requirements on the mailing list [mailinglist] before defining new messages as this will help avoid duplication.
Appendix A - The Plastic Protocol version 0.5

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.

This section should be read in conjunction with the Plastic API defined in Appendix B.

Definitions

In this appendix, the following definitions are used:

- **Broadcast** – send a *Plastic Message* to all candidate clients.
- **Plastic Message** – a request sent from one client to another via a Plastic Hub. Abbreviated to Message.
- **Client** – an application that understands the PLASTIC protocol, excluding the Hub. Could be a Sender, Recipient, or both.
- **Sender** – a Client that can send Plastic Messages to Recipients via the Hub.
- **Recipient** – a Client that can receive Plastic Messages from the Hub. These may have originated from other Clients.
- **Supported Messages** – Plastic Messages that a Recipient declares at registration as wishing to receive. A recipient's supported messages are used by the Hub to determine whether it is a candidate to receive a particular plastic message.

Choice of communication protocols

A Hub offers different communication protocols to support language neutrality. At present the following are supported:

- XML-RPC
- Java-RMI Lite\(^2\)

Typically, a client will use the same protocol to communicate with the Hub as the Hub will use to communicate with the client, but this is not mandatory.

\(^2\) See Appendix D
Connecting to the Hub

Before a client can send or receive messages it must locate the Hub and register with it. A client makes its initial connection to the Hub by means of a file named .plastic in the user’s home directory. For the avoidance of doubt, the location of this file is given by the Java snippet

```
System.getProperty("user.home")
```

On Unix-style operating systems this will usually be equal to the value of the HOME variable, while on Windows it will usually equal the value of the HOMEPATH variable.

A Hub SHOULD write this file when it has started and is ready to receive connections, and SHOULD delete it on shutdown.

This file contains a number of properties in key=value format. The full format of this file is described in [http://java.sun.com/j2se/1.4.2/docs/api/java/util/Properties.html#load(java.io.InputStream)](http://java.sun.com/j2se/1.4.2/docs/api/java/util/Properties.html#load(java.io.InputStream)).

The file MUST contain:

- `plastic.xmlrpc.url`=(url of Hub XML-RPC web server)
- `plastic.version`=(version of protocol supported)
- `plastic.rmi.port`=(port number for RMI connections)

The file MAY contain other properties, but any keys beginning “plastic.” are reserved to the Plastic specification.

A Hub that finds that the .plastic file already exists SHOULD NOT overwrite it if it was written by another Hub, unless it believes that Hub is no longer running. This implies that a User can only run one Hub at a time.

Registering

To register with the Hub, clients call one of the `org.votech.plastic.PlasticHubListener.register*` methods, supplying a human-friendly name, a list of supported messages (if any) and, if appropriate, a means of being called back by the Hub. The method is chosen according to how the client wishes to be called back by the Hub, if at all. If a client chooses Java-RMILite then on registration it MUST pass to the Hub a reference to an object implementing the `org.votech.PlasticListener` interface described in Appendix B. This object will receive messages from the Hub, which may originate from other clients. If a client chooses XML-RPC then on registration the client MUST pass the URL of an XML-RPC server to the Hub. This server will receive messages from the Hub and other clients. The Hub MUST respond with an ID for that client, unique within the Hub, which is used in any further communications. The Hub MUST broadcast a
Sending messages

Clients send messages to other clients via the Hub using one of the `org.votech.plastic.PlasticHubListener.request*` methods.

Senders can broadcast a message to all registered recipients that support the message, or to a subset of them. Messages are not echoed back to the sender. Messages can be sent synchronously, in which case the Hub collects all responses before returning them to the sender or asynchronously, in which case any results are discarded by the Hub. The Hub MUST not forward messages to recipients that have not registered as supporting them. Hubs MAY choose not to pass on messages, for instance if the sender is not registered, or the User has configured the Hub to block certain messages.

Receiving messages

A recipient implements the `org.votech.plastic.PlasticListener.perform` method or its XML-RPC equivalent. The Hub delivers messages to a recipient by invoking this method.

Summary of how messages are routed

Messages are routed from one client to others by the following steps:

1. The sender invokes one of the `request()` methods on the Hub using one of the supported communication protocols.
2. The Hub establishes which of the currently registered clients should receive the message. Only clients which registered as supporting the message and are not the sender are candidates.
3. The Hub invokes the `perform()` method on each of the candidate recipients. The communication protocol used is determined by the `register*` method used by the recipient. The `perform` arguments are determined by the `request` method from the sender. The Hub may have to transform arguments if the sender communication protocol differs from some or all of the recipient communication protocols. The transformation is described in Appendix C. If a sender provides arguments that cannot be transformed then Hub behaviour is undefined.
4. In the case of synchronous requests, the Hub assembles the return values from the recipients into a map/struct and returns this to the sender.

Shutting down

Clients SHOULD unregister with the Hub before closing. If the Hub is unable to contact a client, it may assume it has terminated and unregister it. The Hub MUST broadcast a `ivo://votech.org/hub/event/ApplicationUnregistered(id)` message when a client registers (see Table 1).
message when an application unregisters (or is unregistered). An unregistered client will not receive any further messages, and thus will not receive the ivo://votech.org/hub/event/ApplicationUnregistered(id) message.

When the Hub shuts down, it SHOULD broadcast an ivo://votech.org/hub/event/HubStopping message (see Table 1) and delete the .plastic file.

**Error handling**

Both XML-RPC and Java-RMILite support Faults/Exceptions. These can be used for the Hub to signal an error to the sender, or the recipient to the Hub. Both Hubs and clients should therefore ensure that they handle exceptions robustly. However, a Hub will not pass on exceptions from client to client.

Treatment of Exceptions may be modified in future versions of the protocol.
Appendix B - The Plastic API version 0.5

With apologies to non-Java programmers, the Plastic API is defined by the Java-RMILite interface and described by Javadoc. The Javadoc is given for Java 1.5 to disambiguate Collection object types, but note that clients are not required to use Java 1.5.

Appendix C offers guidance on translating between the Java-RMILite and XML-RPC interfaces.

The PlasticListener interface implemented by Plastic Clients

```java
public interface org.votech.plastic.PlasticListener

Method Detail

java.lang.Object perform(java.net.URI sender,
                         java.net.URI message,
                         java.util.List<java.lang.Object> args)
```

Request that the application perform an action based on a message. Message arguments and return types must be valid XML-RPC types as defined in Appendix C of the Plastic IVOA Note.

**Parameters:**
- `sender` - the PLASTIC ID of the originating application.
- `message` - the URI representing the action.
- `args` - any arguments to pass. These must be valid XML-RPC types.

**Returns:**
- any return value of the action. This must be a valid XML-RPC type.
The PlasticHubListener interface implemented by Plastic Hubs

public interface org.votech.plastic.PlasticHubListener

### Method Detail

`java.util.List<java.net.URI> getRegisteredIds()`

Get the PLASTIC IDs of all the currently registered applications.

`java.net.URI getHubId()`

Get this Hub's PLASTIC ID. The Hub has an ID which it uses when it sends clients messages from itself.

**Returns:**
see above

`java.lang.String getName(java.net.URI plid)`

Get the human-readable name of the application with this PLASTIC ID, as given at registration.

**Parameters:**
plid - the PLASTIC ID

**Returns:**
The user friendy name

`java.util.List<java.net.URI> getUnderstoodMessages(java.net.URI plid)`
Get the messages understood by the application with this PLASTIC ID. Note that just because an application declares itself able to understand a message, there is no requirement for it to act on it.

**Parameters:**
- plid - the PLASTIC ID of the application

**Returns:**
A list of message URIs

```java
java.util.List<java.net.URI> getMessageRegisteredIds(java.net.URI message)
```

Get the PLASTIC IDs of all the applications that support a particular message.

**Parameters:**
- message - the message in question

**Returns:**
A list of plastic ids.

```java
java.net.URI registerXMLRPC(java.lang.String name,
java.util.List<java.net.URI> supportedMessages,
java.net.URL callBackURL)
```

Register an application with the Hub, requesting that any callbacks are made via XML-RPC.

**Parameters:**
- name - A human-readable name describing the application.
- supportedMessages - an array of messages (as URIs) the application is interested in.
- callBackURL - the application's internal xmlrpc server URL. Used by the hub to send messages to the application.

**Returns:**
A hub-assigned PLASTIC ID

```java
java.net.URI registerRMI(java.lang.String name,
java.util.List<java.net.URI> supportedMessages,
PlasticListener caller)
```
Register an application with the Hub, requesting that any callbacks are made via Java RMI.

**Parameters:**
- **name** - A human-readable name describing the application.
- **supportedMessages** - an array of messages (as URIs) the application is interested in.
- **caller** - a reference to the PlasticListener that wishes to register

**Returns:**
a hub-assigned PLASTIC ID

---

```java
java.net.URI registerNoCallBack(java.lang.String name)
```

Register an application with the Hub, requesting no callbacks from the Hub. This is to allow uncallable applications like scripting environments to register.

**Parameters:**
- **name** - A human-readable name describing the application.

**Returns:**
a hub-assigned PLASTIC ID

---

```java
void unregister(java.net.URI plid)
```

Unregister the application with this Plastic ID from the Hub.

**Parameters:**
- **plid** - the PLASTIC ID of the application to unregister

---

```java
java.util.Map<java.net.URI,java.lang.Object> request(java.net.URI sender,
java.net.URI message,
java.util.List<java.lang.Object> args)
```
Send a message to all registered PLASTIC applications. Message arguments and return types must be valid XML-RPC types as defined in Appendix C.

Parameters:
- **sender** - the PLASTIC ID of the originating tool - provided by the hub on registration.
- **message** - the message to send.
- **args** - any arguments to pass with the message. Arguments must be valid XML-RPC types.

Returns:
- a Map of application plastic ids to responses. Responses must be valid XML-RPC types.

```java
java.util.Map<java.net.URI,java.lang.Object>
requestToSubset(java.net.URI sender,
                java.net.URI message,
                java.util.List<java.lang.Object> args,
                java.util.List<java.net.URI> recipientIds)
```

Send a request to listed registered PLASTIC apps. Message arguments and return types must be valid XML-RPC types as defined in Appendix C.

Parameters:
- **sender** - the plastic id of the originating tool - provided by the hub on registration.
- **message** - the message to send.
- **args** - any arguments to pass with the message. Arguments must be valid XML-RPC types.
- **recipientIds** - a list of target application PLASTIC IDs

Returns:
- a Map of application PLASTIC IDs to responses. Responses must be valid XML-RPC types.

```java
void requestToSubsetAsynch(java.net.URI sender,
                           java.net.URI message,
                           java.util.List<java.lang.Object> args,
                           java.util.List<java.net.URI> recipientIds)
```

Send a request to listed registered PLASTIC apps, without waiting for a response. Message arguments and return types must be valid XML-RPC types as defined in Appendix C.
Parameters:
- **sender** - the PLASTIC ID of the originating tool - provided by the hub on registration.
- **message** - the message to send.
- **args** - any arguments to pass with the message. Arguments must be valid XML-RPC types.
- **recipientIds** - a list of target application PLASTIC IDs

```java
void requestAsynch(java.net.URI sender,
                   java.net.URI message,
                   java.util.List<java.lang.Object> args)
```

Send a message to all registered PLASTIC applications, without waiting for a response. Message arguments and return types must be valid XML-RPC types as defined in Appendix C.

Parameters:
- **sender** - the PLASTIC ID of the originating tool - provided by the hub on registration.
- **message** - the message to send.
- **args** - any arguments to pass with the message. Arguments must be valid XML-RPC types.
Appendix C – Converting from Java-RMILite to XML-RPC

The XML-RPC method names are the same as for Java-RMILite, except that:

- the client method
  `org.votech.plastic.PlasticListener.perform` maps to `perform`,
- and the hub methods beginning
  `org.votech.plastic.PlasticHubListener.*` map to `plastic.hub.*`

Since a message transmitted using the Java-RMILite communication protocol must have precisely the same meaning as an equivalent one transmitted using XML-RPC, all message arguments and return values must be losslessly translatable between Java objects and XML-RPC value elements. Therefore only those Java object types which have exact equivalents in XML-RPC may be used in message argument lists and return values. Behaviour of the Hub if it receives a message with invalid argument types is undefined. The following table lists the correspondence between the two forms; the hub will transform between these types when translating a message or response from one communication protocol to the other.

<table>
<thead>
<tr>
<th>Java Type</th>
<th>XML Tag Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td><code>i4</code>, or <code>int</code></td>
<td>A 32-bit, signed, and non-null, integer value.</td>
</tr>
<tr>
<td>Boolean</td>
<td><code>boolean</code></td>
<td>A non-null, boolean value.</td>
</tr>
<tr>
<td>String</td>
<td><code>string</code></td>
<td>A string, non-null.</td>
</tr>
<tr>
<td>Double</td>
<td><code>double</code></td>
<td>A signed, non-null, double precision, floating point number. (64 bit)</td>
</tr>
<tr>
<td><code>java.util.Date</code></td>
<td><code>dateTime.iso8601</code></td>
<td>A pseudo ISO8601 timestamp, like 19980717T14:08:55. However, compared to a true ISO8601 value, milliseconds, and time zone informations are missing. This type is deprecated.</td>
</tr>
<tr>
<td><code>byte[]</code></td>
<td><code>base64</code></td>
<td>A byte array, which is encoded as base64 in the XML-RPC representation</td>
</tr>
</tbody>
</table>
A set of key value pairs. The keys are strings. The values may be any valid data type, including another map.

An array of objects. The array elements may be any valid data type, including another array.

In addition, some Plastic method signatures include URI argument types in the Java-RMI Lite form. These are translated to and from strings for the XML-RPC form.

Equivalents of the XML-RPC types in Perl, Python etc, are dependent on the XML-RPC library used by the client.
Appendix D – A note on the Java-RMILite interface

The Java-RMI (Lite) interface used in Plastic is simplified through the use of an open source library called RMILite [RMILite] that removes the need for Remote, RemoteException and rmic.

In traditional RMI [RMI], the server instantiates an object implementing an interface that extends Remote, and each method in that interface must throw a RemoteException. This object's class must be compiled using rmic to create a stub class for the client to use, and the object must be registered with an rmiregistry.

With RMILite, this complexity is hidden. A server needing to expose an object's methods over RMI contains code similar to:

Server server = new Server(port);
server.publish(PlasticHubListener.class, new PlasticHubListenerImpl());

A client needing to access this server contains:

Client client = new Client("localhost", port);
client.exportInterface(PlasticListener.class);
hub = (PlasticHubListener)
client.lookup(PlasticHubListener.class);

Note the exportInterface() method. This instructs RMILite to pass object implementing this interface by reference, rather than by serializing a copy.

Users of RMILite do not need to compile with rmic, and do not need to deal with RemoteExceptions – any network related exceptions are thrown as RuntimeExceptions.

This simplification offered by RMILite comes at the cost that Plastic Hubs are not accessible through vanilla Java RMI.
Appendix E – Screenshots of Plastic Messaging between Aladin and Topcat

The following screenshots illustrate Plastic Messaging between two real-world applications. Many of the details are application-specific, but demonstrate how Plastic works in practice. Other demos can be found on the Plastic Demos [demos] page.

At startup, both TOPCAT and Aladin try to automatically connect to the Hub:

![Screenhots of Plastic Messaging between Aladin and Topcat](image)

This behaviour is application-specific. For some applications it will be more appropriate for the user to initiate the connection manually.

**Sending a table from Aladin to TOPCAT**

By right-clicking on a given catalogue plane, Aladin allows to broadcast it to TOPCAT (or any other PLASTIC-compatible application):
Conversely, we could also send any TOPCAT table to Aladin (or any other PLASTIC-compatible application) by clicking on Topcat's 'Interop' menu.
**Sending a density map image from TOPCAT to Aladin**

After having created a density map in TOPCAT, the result can be sent to Aladin as a FITS image:

Once in Aladin, this image can be manipulated as a regular image plane:

If the density map was plotted on the sky (which is the case in our example), TOPCAT sends WCS information in the FITS header.

**Selecting a given source highlights it in the other application**

From TOPCAT activation preferences, we can choose to 'transmit' the row when selected:
Thus, selecting a given source in TOPCAT will highlight it in Aladin:

Conversely, passing over a source in Aladin will show it in TOPCAT:
This feature allows linked views between applications, and facilitates exploration of a dataset in both physical and parameter space.

**Broadcasting a subset of sources**

After having created a subset of sources from any TOPCAT view, we can broadcast this subset in order to see where they lie in Aladin:
Conversely, the selection of a subset of sources in Aladin will create a new subset in TOPCAT:
**Sending the position of a source**

From TOPCAT activation preferences, we choose to send the position of a source when selecting it:

Thus, selecting a given source in TOPCAT will send a message asking Aladin to show its position on the sky (pointed out by the pink reticle):

**References**

[AstroRuntime] [http://www2.astrogrid.org/desktop](http://www2.astrogrid.org/desktop)

[AstroWeka] [http://astroweka.sourceforge.net](http://astroweka.sourceforge.net)

[Fernique, P.] 2006 VOAApplication Interoperability Study
http://www.ivoa.net/internal/IVOA/InterOpMay2006Applications/Victoria2006.ppt
[Mann]  http://www.roe.ac.uk/~rgm/sc4devo/
[messages]  IVOA Note on current Plastic Messages (in prep.)
[Topcat]  http://www.starlink.ac.uk/topcat/
[VisIVO]  http://visivo.cineca.it/
[votech]  http://eurovotech.org
[Winstanley, N. et al] 2006  Astro Runtime - Client-Side Middleware for the VO,