EFDA–JET–RE(02)04

C. Hogben and S. Griph

Interfacing to JET Plant Equipment using the HTTP Protocol

Interfacing to JET Plant Equipment using the HTTP Protocol

C. Hogben and S. Griph

JDN/H(02)011
Issue 1 : 8th April 2002

UNCONTROLLED WHEN PRINTED

Interfacing to JET Plant Equipment Using the HTTP Protocol

Author: Colin Hogben and Sverker Griph

Abstract
This document describes a proposal to use the HTTP protocol for interfacing to a piece of plant equipment, e.g. a diagnostic. The plant system is considered to be a 'blackbox' from CODAS point of view, i.e. it is not built from standard CODAS hardware nor does it use standard CODAS software for its implementation.

Keywords
blackbox

R.O.: Sverker Griph	Reviewer: Martin Wheatley	Approver: Jonathan Farthing

Distribution
CODAS Document Centre, web page http://w3.jet.efda.org/CODAS/Document_Library .
CODAS via E-mail

Chapter 1 Introduction

This document describes a proposal to use the HTTP protocol for interfacing to a piece of plant equipment, e.g. a diagnostic. The plant system is considered to be a ”blackbox” from the CODAS point of view, i.e. it is not built from standard CODAS hardware nor does it use standard CODAS software for its implementation.

The assumption is that the ”blackbox” is connected to the DATAnet section of the CODAS computer network and the TCP/IP protocol is used to connect to an HTTP server in the ”blackbox” equipment.

From CODAS point of view the HTTP ”blackbox” interface will support the following optional functions:

- Setting and reading back of plant equipment parameters,
- Reading of plant equipment state,
- Monitoring of plant equipment status,
- Setting up of plant equipment data channels for pulse data collection before a JET pulse and reading of the collected data after the pulse.
- Central logging or plant equipment error and warning messages.

Typically a service has an advertised base URL, such as http://blackbox/path/to/object and GET and POST methods on this URL and/or subsidiary URLs correspond to method calls on a remote plant object.

There are many implementations of both client and server sides of the HTTP protocol, in most popular programming languages. The server could for example be a full-featured web server such as Apache; or a fairly minimal implementation in Python extending the HTTP Server classes which come with the Python distribution.

For background on the HTTP protocol, see Appendix A.

The results of transactions described in this document are in general not intended to be viewed with a standard web browser. However, a browser may be of use during testing, for example to view results, or to generate test POST requests from HTML forms (application/x-www-form-urlencoded format). Furthermore, the implementer of the server side may optionally provide HTML versions of the various resources to allow the system to be browsed by a human.

Chapter 2 Use of the HTTP Protocol

2.1 JET Plant Security Considerations

This HTTP interface proposal has been specified in such a way that all HTTP client interaction with the JET plant HTTP server can be exercised using HTML forms from a standard browser HTTP client. This was done on purpose to simplify testing and commissioning. However, this means that it will be very easy to interfere with a piece of JET equipment, which is using this interface.

It is therefore of paramount importance that the equipment that is implementing the HTTP server takes steps to prevent unauthorised access. The minimal requirement is that any HTTP server using this interface standard should restrict client requests to those comming from one host or when there are special reasons possibly a list of authorised hosts.

This host authorisation should ideally be configurable. However, normally only the host subsystem computer will be authorised.

2.2 JET Network Load Considerations

The HTTP protocol allows great variation in how to use TCP/IP connections. Since opening a new TCP/IP connection is associated with considerable overhead, in particular network overhead, implementors of the HTTP server should take great care to keep TCP/IP connections open as long as possible.

For example, when a HTTP reply is associated with a document in the HTTP body, the header should always directly or indirectly specify the 'Content-Length' of the document in the HTTP header. It should not rely on connection closure to specify the length of the reply body.

The CODAS HTTP client will monitor and log any deviations from this recommendation, and implementors of the HTTP server will be reminded, if the recommendation is found to be violated.

Chapter 3 Setting and Reading Back of Plant Equipment Parameters

All plant equipment parameters are grouped into one set where each parameter is identified by a name that is unique within the set. An example parameter name could be ”wave-length”.

The set of plant parameters should be agreed in advance. The black-box plant server should initialise all parameters in the set with safe and sensible defaults at start time. When receiving parameter values the HTTP plant server should ignore parameters with unknown names. In particular unknown parameters must not be returned in the response to a parameter GET request since this would prevent the HTTP client from checking the server parameter handling.

Setting of parameter values will be done by sending a request to the agreed URL representing the service for the complete parameter set. The request to update the set is a HTTP POST request that has the same format as if it was the result of an HTML form, where the form input fields correspond to the parameters. (All parameters in the form are considered to be set at the same time and no assumption can be made about the actual order.)

For example a HTTP POST request to http://blackbox/params with posting

```
blackbox-factor=42&wave-length=0.5432E-8
```

could have been generated from an HTML form such as

```
<form action="http://blackbox/params" method="POST">
Blackbox-factor: <input type=text name="blackbox-factor">
Wave-length:     <input type=text name="wave-length">
<input type=submit value="Go">
</form>
```

When receiving this posting the plant server should set the two plant parameters blackbox-factor and wave-length to 42 and 0.5432E-8, respectively.

The HTTP POST request should return the full set of recognised parameter names and values in text/plain format. Each line of this reply document contains a parameter name, an “=” sign, and an ASCII representation of the parameter value; lines are terminated by either a linefeed character or a carriage-return/linefeed pair. Note that only numeric parameters are catered for currently; should string parameters (e.g.) be required in the future, some as yet unspecified form of encoding of the parameter values will be required.

Similarly, a GET on the parameter set URL should return the same document.

Chapter 4 Monitoring the Plant Equipment State

4.1 Reading of Plant Equipment State Variables

This service is functionally identical to the above GET of a parameter set in all aspects but the URL address. Thus each set of plant state variables is represented by an agreed URL such as http://blackbox/state-variables. An HTTP GET request should return an agreed full set of "*name=value*" pairs in text/plain format.

This service will be used by the HTTP client to regularly poll the plant equipment state in order to display it in the control room e.g. on plant equipment mimics.

In a second implementation phase (as agreed by those providing the specific "blackbox" HTTP servers) the HTTP server could use the standard "server push" mechanism to send state monitoring updates back to the client (see Appendix A). The connection remains open, and whenever the state changes the full "*name=value*" pair set is sent again. When the client does not want to monitor the state any longer it closes the connection.

4.2 Monitoring of Plant Equipment Status

To monitor the status, the client polls the "blackbox", typically by issuing HTTP GET requests to *baseurl*/monitor. The server responds with a document containing either 1 (if the status is bad) or 0 (if good). This is analagous to the operational/non-operational bit on a subsystem component.

Optionally, the returned document for the bad state may be a Reason document in text/xml format (see Appendix B).

In a second implementation phase (as agreed by those providing the specific "blackbox" HTTP servers) the HTTP server could use the standard "server push" mechanism to send monitoring updates back to the client (see Appendix A). The connection remains open, and whenever the status changes between good and bad a further update is sent. When the client does not want to monitor the status any longer it closes the connection.

Chapter 5 JET Pulse Data Collection

This section describes how to use the HTTP protocol when setting up of plant equipment data channels for pulse data collection before a JET pulse and how to read the collected data after the pulse.

The ”blackbox” data collection service has an advertised “baseurl” which forms the base for the various methods. For example, http://blackbox/cgi-bin/pulse.cgi. The different services that are needed to set up and collect data are addressed using this “baseurl”.

5.1 Pulse Control

This is modelled on the existing GAP csl8 protocol. A ”blackbox” may be implemented as one or more GAP modules (c.f. GAVMOD GAP tree nodes) each optionally controlling one or more data collection channels.

In the context of a GAP module with multiple channels, a single “baseurl” would represent the whole module+channels complex, with subsidiary URLs representing the module and channels individually. Pulse control events would be sent as HTTP POST requests (see Appendix A) to the module URL “baseurl”/control, with query parameters to indicate the particular event and pulse type / number.

command	Meaning
init	Prepare for pulse
abort	Abort pulse - tidy up all pulse-related data
end-init	All channel nodes now initialised
end-of-pulse	Plasma pulse has finished, data collection may follow
data-archived	All data collection complete, tidy up

In addition to the command field, the other fields passed are

- type (IPF, QPF, JPF, LPF or DPF: Pulse type)
- pulse indicating the pulse number.

Note! The abort command may be issued at any pulse phase, except after the “end-of-pulse” command has been given.

A full command for starting a new JET pulse would be sent as a HTTP POST request with e.g. the following contents:

```
command=init&type=JPF&pulse=54321
```

The ”blackbox” HTTP server should reply with an empty document to each the above commands. However, the end-init command may return failure in the form of a text/xml document specifing the reason why the initialisation failed (see Appendix B).

N.B. Information as to whether this is an on-line or off-line pulse is missing. This omission is a reflection of a similar omission from the existing csl8 protocol. It is only an issue if on-line and off-line pulse numbers in the same range are being used, which in practice is unlikely to occur.

5.2 Node Initialisation

Channel nodes, which take part in the pulse data collection scheme (both data-producing channels and ancilliary channels such as clock nodes) are accessed by HTTP requests to "*baseurl*/node/*nodename*, where "*nodename*" is the name assigned by CODAS. Note that the nodename is a mapping of the canonical CODAS nodename format, lowercase, with the special characters '`<`' and '`>`' encoded as '`_in_`' and '`_out_`', respectively. E.g. the URL used for the node `KX2<DAT:001` would be encoded

```
http://blackbox/cgi-bin/pulse.cgi/node/kx2_in_dat:001.
```

The channel node initialisations take place in between the "init" and "end-init" command of the "Pulse Control" above.

Initialisation is performed by issuing a POST request to this URL, with query fields "command=init", "type=JPF" (or IPF, QPF, DPF, LPF), "pulse=*pulsno*", "nodetype=*nodetype*" and "retbyt=*retbyt*" followed by other node type dependent fields as described below.

In response, the server should send either an empty document indicating success, or a 'text/xml' document giving the reason why the initialisation could not be completed (see Appendix B).

The following nodetypes are supported:

```
GAVMOD
GACLOK
GAANA
GADIG
GASPEC
```

5.2.1 GAVMOD node type

See the gavmod manual page.

This node type can be used to send application specific initialisation data to the "blackbox" plant control system. If such data is included the init command will be sent as an HTML POST request with "multipart/form-data" format, with two parts. The initial part will be of "application/x-www-form-urlencoded" format followed by the specific initialisation data in "text/plain" format. If there is no application specific data to send the HTTP POST request will be in a single part of format "application/x-www-form-urlencoded"

The initial/single part will be in addition to the standard init fields also specify the "quota=*quota*" as given in the GAP tree node.

5.2.2 GAVCLOK node type

See the gaclok manual page.

This node type can be used to send timer specific initialisation data to the "blackbox" plant control system. The command will be sent as an HTML POST request with "application/x-www-form-urlencoded" format followed by the timer initialisation data.

```
num_of_seq=xxx&
```

```
nip_0=xxx&
nip_1=xxx&
nip_2=xxx&
nip_3=xxx&
nip_4=xxx&
nip_5=xxx&
nip_6=xxx&
tik_0=xxx&
tik_1=xxx&
tik_2=xxx&
tik_3=xxx&
tik_4=xxx&
tik_5=xxx&
tik_6=xxx&
tin_0=xxx&
tin_1=xxx&
tin_2=xxx&
tin_3=xxx&
tin_4=xxx&
tin_5=xxx&
tin_6=xxx
```

On the return server will reply either with empty text document which indicates the success or document in "text/xml" format decsribing the reason of failure.

5.2.3 GAANA node type

See the gaana manual page.

This node type is used to request data from a task which generate analogue data. The command will be sent as an HTML POST request with "application/x-www-form-urlencoded" format followed by the requested data specific information.

```
samplesize=xxx
```

On the return server replies either with the data in binary format or the document in "text/xml" format indicating the reason of failure.

5.2.4 GADIG node type

See the gadig manual page.

This node type is used to request data from a task which generate digital data. The command will be sent as an HTML POST request with "application/x-www-form-urlencoded" format followed by requested data specific information.

```
samplesize=xxx
```

On the return server replies either with data in binary format or the document in text/xml format indicating the reason of failure.

5.2.5 GASPEC node type

See the gaspec manual page.

This node type is generally used for spectrometer data such as collected by the PCs controlling Princeton and Jonathan Wright's instruments.

The init command will be sent as an HTML POST request with "multipart/form-data" format. One of the parts will be the contents of the initialisation file, as indicated by the `$FILE` field of the GAP node, and be given the name "init-file". This is equivalent to the results of a file upload from an HTML form (see Appendix A). Other parts of the multipart document will contain the standard init fields and also the "dattyp" as given in the GAP tree node.

5.3 Data Collection

Data is collected by the client issuing a GET request on the node URL, with query parameters indicating the pulse number and type.

The response should be either an XML reason why data was not available, or the collected data in either BIG-endian binary "application/octet-stream" format or, in the case of GAANA node types, optionally a "multipart/mixed" format, where the actual data is BIG-endian binary "application/octet-stream" preceded by an "text/xml" document describing how the data has been calibrated (XML format to be decided when this feature is needed the first time).

Chapter 6 Plant Equipment Logger Interface

In order to provide the HTTP server and the JET plant equipment with a mean to log information on the CODAS subsystem, a logger interface is provided. This will use the URL *baseurl*/log. The HTTP client will be polling the HTTP server with this url (POST method) periodically, with predefined (configurable parameter) timeout. The server keeps and accumulates internally log messages and sends them back to the client as the response to this command. Each such poll should empty the server log message queue.

The returned log messages may be in one of two formats: text/xml Reason and text/plain format. Mime types "text/plain" and "text/xml" must be used.

When the client receives a response it interprets the content-type in the header and determines type of message. In the case of text/xml format, severity level and text stream are part of reason structure.

In the case of the text/plain format, these messages - ASCII strings - will be treated separately and redirected to a log stream. In CODAS we have three levels of severity for messages: Error, Warning and Information. If the ASCII text strings start with any of the specifications: "Error:" "Warning:" or "Info:" they will be logged accordingly to the provided severity level. If the specification is missing then default Error is assumed. The HTTP server side can thus provide a optional severity level for each log message.

Appendix A HTTP Protocol

This appendix briefly describes the HTTP protocol, and is intended as background information for the HttpBlackboxInterfaceProposal. Features of HTTP which are salient in this context are described. It is intended for readers who are familiar with web browsing and the concept of MIME without knowing the underlying details. See RFC 2616 (e.g at `http://RFC.net/rfc2616.html`) for the full gory details.

HTTP is carried over TCP/IP, and each transaction consists of a request from client to server, and a response from server to client. Both the request and the response have the same overall format: an introductory line, a set of header lines (like in an email message), a blank line and an optional body. The body, when present, can contain anything; its MIME type is indicated in the header.

A typical HTTP transaction to fetch a fictional web page http://w3/~fred/home.html might look like this:

```
Client to server:

  GET /~fred/home/html HTTP/1.0
  User-Agent: gethttp
  Accept: */*
  Host: w3


Server to client:

 HTTP/1.1 200 OK
 Date: Fri, 31 Aug 2001 08:57:59 GMT
 Server: Apache/1.2.4
 Last-Modified: Wed, 09 May 2001 13:56:53 GMT
 ETag: "63561-7c4-3af94ca5"
 Content-Length: 52
 Accept-Ranges: bytes
 Connection: close
 Content-Type: text/html

 <title>Fred's Home Page</title>
 Fred is not at home
```

In the first line of the request from client to server, ”GET” is the request method (”GET” and ”POST” are essentially the only two methods used when web browsing, though others are defined); next is the path, everything in the URL following the hostname; finally an indication of which version of HTTP the client is talking.

The ”Host:” header tells the server which host the client thinks it is talking to. This is useful in a virtual hosting environment where many web sites share the same IP address. The ”User-Agent:” header gives information about which client is being used; if Netscape is being used it might say “Mozilla/4.75 [en] (X11; U; SunOS 5.7 sun4u)”. The ”Accept:” header tells the server which MIME types the client understands or prefers.

Following the headers is the mandatory blank line. In the case of a "GET" request, there is no message body.

The response begins with a status line. The first item tells the client the hisghest version of HTTP the server supports. In the example it is 1.1; however, the server tailors the remainder of its response to the version the client specified. The second item is a 3-digit status code; 200 means success, other common ones are 404 (file not found), 401 (authorisation required) and 302 (page redirect). Following the status code is a short English description of the status code.

Various headers follow, many of which are just informational and not vital to the operation of the protocol. The most important headers are "Content-Type:" and "Content-Length:" which describe the reply body (in the example, an HTML document).

A.1 Web forms, Queries and POST requests

The semantics of "GET" are to fetch some data without modifying state. By comparison, "POST" can result in a change of something.

In a webpage context, filling in a form can be associated with either a GET or a POST request; this is specified in the <form> element. Other elements within the form allow the construction of a query, which is a set of name/value pairs, and these parameter pairs are passed to the server as part of the request.

In the case of a GET, the query becomes part of the URL, separated by a question mark, with each name=value item separated by an ampersand. For example, a search form defined thus:

```
<form action="http://w3/search.cgi" method="GET">
Area: <select name="area">
 <option value="prod">Products
 <option value="folk">People
</select>
Search for: <input type=text name="term">
<input type=submit value="Go"></form>
```

might result in a request URL

```
‘‘http://w3/search.cgi?area=prod&term=chocolate’’
```

In the case of POST, the query is passed in the body of the request instead of being tacked onto the URL. If we were to change the method in the eaxmple above to POST, the request to the server might look like this:

```
POST /search.cgi HTTP/1.0
User-Agent: gethttp
Accept: */*
Host: w3
Content-type: application/x-www-form-urlencoded
Content-length: 24

area=prod&term=chocolate
```

Note the Content-type header which is specific to this type of form submission. There is an alternative Content-type and way of sending parameter name/value pairs, which is used when sending large amounts of data, particularly when uploading files - this is "multipart/form-data". In this case, the body of the request is a sequence of MIME parts. For example, imagine a web form for uploading a picture into a photo album, together with a caption. The request might look like this.

```
POST /album-add.cgi HTTP/1.0
User-Agent: gethttp
Accept: */*
Host: w3
Content-type: multipart/form-data; boundary="foo123"
Content-length: 4628

--foo123
Content-type: text/plain
Content-disposition: form-data; name="caption"
Content-length: 21

Fred at the seaside

--foo123
Content-type: image/jpeg
Content-disposition: form-data; name="photo"
Content-Transfer-Encoding: base64
Content-length: 4358

/9j/4AAQSkZJRgABAQEAWgBaAAD/2wBDAAgGBgcGBQg...
...
p5sMEJHasKvum0JaWP/Z
--foo123--
```

The fact that HTML allows the formulation of queries in this way is not part of the HTTP protocol as such. However, using the same conventions for parameter passing is benficial: there is plenty of server-side software support; and standard browsers can potentially be used for testing parts of a blackbox implementation.

A.2 Server Push

Normally, a request results in a single document being returned by the server. There are circumstances where it is desirable for the server to 'push' updates of a document to the client at intervals. Examples are updating cricket scores, or replacing banner adverisements. A special MIME type has been allocated for this purpose: "multipart/x-mixed-replace". The server sends the sequence of document updates as parts of a multipart MIME document, only sending a new part when it is ready. The client (e.g. browser) replaces its representation of the document whenever a new part arrives.

E.g. an updating status could be sent to the client with server push by the server sending this response initially, and keeping the TCP connection open:

```
  HTTP/1.0 200 OK
  Connection: close
  Content-type: multipart/x-mixed-replace; boundary="xyz"

  --xyz
  Content-type: text/plain

  OK
  --xyz

Later when it wants to update the status, it send this:

  Content-type: text/plain

  Bad
  --xyz
```

And so on.

Appendix B Structured Reasons

Structured Reasons is a proposed XML representation to be used to transport detailed information about any type of exceptions.

Here is a Reason example:

```
<reason xmlns="http://jet.efda.org/ns/reason/">
 <source uri="http://jet.efda.org/KG1"/>
 <text>Lasers not ready for pulse</text>
 <sub>
  <reason>
   <source uri="http://jet.efda.org/KG1/DCN-laser"/>
   <text>DCN laser amplitude too low</text>
   <error domain="http://jet.efda.org/codas/errors/" number="13435003"/>
  </reason>
 </sub>
</reason>
```

The formal XML DTD that defines the vocabulary for structured Reasons is:

```
<!ELEMENT reason (source?,text,error?,severity?,sub?) >

<!ELEMENT source EMPTY >
<!ATTLIST source uri CDATA #REQUIRED >

<!ELEMENT text (#PCDATA) >

<!ELEMENT error EMPTY >
<!ATTLIST error domain CDATA #REQUIRED >
<!ATTLIST error number NUMBER #REQUIRED >

<!ELEMENT severity EMPTY >
<!ATTLIST severity domain CDATA #REQUIRED >
<!ATTLIST severity number NUMBER #REQUIRED >

<!ELEMENT sub (reason+) >
```