Web interface for online ROOT and DAQ applications

Jörn Adamczewski-Musch, Bertrand Bellenot, and Sergey Linev

***Abstract* – *A* specialized web server, based on embeddable Civetweb http server, has been implemented in ROOT and DABC frameworks. This server can deliver data directly from running applications to a web browser where JavaScript-based code is used for interactive web graphics. Without modifications arbitrary ROOT-based code can be monitored remotely. Through a flexible plug-in mechanism data from different systems (like EPICS, FESA, MBS, and others) can be easily integrated and displayed together. As a result, a unified user interface for distributed heterogeneous systems can be build.**

I. Introduction

In many experiments online tools are required to control and monitor all stages of data taking and online analysis. Usually many different software components are involved in such set up. Each of them provides own tools and methods, which is often hard to integrate with each other.

Many online monitoring tasks can be solved with web technologies: one could use HTTP protocols for data exchange; different methods for user authentication and access control; HTML and JavaScript for interactive graphics in web browsers.

II. JSRootIO project

The main goal of the JSRootIO [1] project is to display ROOT objects like histograms or graphs in a web browser. The core functionality of the code is revealed by the name – JavaScript ROOT Input/Output. This provides the functionality to decode data from binary ROOT files and create JavaScript objects. Using interactive JavaScript web graphics, such objects can be displayed in most modern web browsers like Microsoft IE, Mozilla Firefox, Google Chrome, Opera.

Many improvements in the I/O part have been implemented to make JavaScript code compatible with original ROOT I/O: A *JSROOTIO.TBuffer* class has been introduced with very similar functionality as *TBuffer* class in ROOT. All custom streamers are treated in a central place, which makes it much easier to maintain and extend functionality for the future. Major ROOT classes with custom streamers (like *TCanvas*, *TList*, *TStreamerElement*) are supported.

Manuscript received May 22, 2014.

J. Adamczewski-Musch and S. Linev are with the GSI Helmholtzzentrum für Schwerionenforschung, Planckstr. 1, 64291 Darmstadt, Germany. Corresponding author is S. Linev (telephone +49-6159-711338, e-mail: S.Linev@gsi.de).

B. Bellenot is with CERN, Geneva, Switzerland (e-mail: Bertrand.Bellenot@cern.ch).

Also the graphical part of JSRootIO has been improved. The main focus was put on flexibility – now JSRootIO graphics can be inserted on any web page – it is not restricted to the default HTML page design as originally provided by JSRootIO. New important features were implemented – a context menu and statistic box update. The context menu provides an easy way to activate different commands associated with the object by the right mouse button: switch draw options, toggle linear/logarithmic scale, enable/disable histogram statistic. Also comfort zooming with update of histogram statistics box was implemented.

Current JavaScript graphics of JSRoootIO to a large extend reproduces the look-and-feel of original ROOT graphics. Fig.1 shows a complex *TCanvas* with several overlaid graphs, displayed both in native ROOT (left) and in a web browser with JSRootIO code (right).

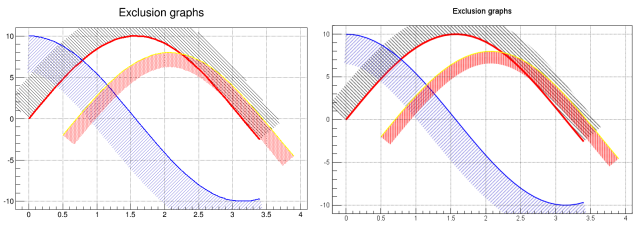

Fig.1.: Same canvas, displayed with native ROOT graphics (left) and JSRootIO code (right).

The main advantage of original JSRootIO approach – one could use such code together with any web server like Apache or IIS without any special requirements to the server. It is sufficient to put JSRootIO scripts and ROOT files on the web server and to provide a correct HTTP address on the main HTML page – the web server is used as simple file server. But this technique may be difficult for online tasks, where many objects should be accessible at any time and monitoring frequency could be relatively high. Under such conditions a pure file-based approach can not perform efficiently, as the data provider application is permanently writing hundreds of objects to ROOT files that are simultaneously read back by the web server.

III. HTTP server in ROOT

An HTTP server running in the ROOT application provides direct access to its data objects without the need of any intermediate files. Any ROOT object can be streamed no sooner than an HTTP request arrives for it, and can then immediately be delivered to the browser.

In the following sections the key components of the newly developed web server in ROOT are treated in detail.

A. Sniffer of ROOT objects

Implementing a web server for ROOT objects requires an API with a unified interface to different ROOT structures. This was realized as *TRootSniffer* class. It offers methods to browse and access ('sniff') objects in folders, files, trees and different ROOT collections. Any object (or object element) can be identified by a path string that can be used in an HTTP request to uniquely address the object.

Each ROOT object can be "streamed" (serialized) into binary zipped data by means of standard ROOT methods. Such binary data can be decoded by the I/O part of JSRootIO code. *TRootSniffer* also provides a list of "streamer info" objects that are required to reconstruct (deserialize) the original object from the binary data.

TRootSniffer also can generate an XML representation of any ROOT object, using *TBufferXML* class functionality. For some ROOT classes like canvas (*TCanvas*) or histograms (*TH1*, *TH2*, *TH3*), a PNG/BMP/GIF image can be produced as well.

By default *TRootSniffer* could access all objects reachable via *gROOT* pointer: open files, trees, canvases and histograms. If necessary, a user could explicitly register any object in the folder structure. *TRootSniffer* provides access not only to objects, but also to all class members by means of ROOT dictionaries.

B. JSON representation of ROOT objects

In ROOT a binary "streamed" representation is used to store objects in the files. JSRooIO makes it possible to decode such information at the web browser in JavaScript, but this does not always work, especially in case of custom streamers. The new *TBufferJSON* class solves this problem, performing all necessary I/O operations directly on the application side. It converts ROOT objects into JSON (JavaScript Object Notation) format, which can be parsed with standard JavaScript methods. With such approach, I/O code remains completely in the ROOT-based web server, and the clients will receive ready-to-use objects. An example of the JSON representation for a *TNamed* object is shown in Fig.2.

```
{
  "_typename" : "JSROOTIO.TNamed",
  "fUniqueID" : 0,
  "fBits" : 50331648,
  "fName" : "name",
  "fTitle" : "title"
}
```

Fig.2. Example of JSON representation of a *TNamed* object in ROOT. Text produced with command TBufferJSON::ConvertToJSON(new TNamed("name","title"));

TBufferJSON class performs a special treatment for classes like *TArray* or *TCollection* to generate a representation that is closer to corresponding JavaScript classes. Also user classes with custom streamers can be equipped with special function calls, providing a meaningful representation for objects data in JSON format. *TBufferJSON* can stream not only whole objects, but also specified class members. This allows to access any member of an object in a text form.

C. Civetweb-based HTTP server

Civetweb [2] embeddable web server was used to implement HTTP protocol in ROOT. Civetweb has very compact code and provides necessary functionality like multithreaded HTTP requests processing, user authentication, secured HTTPS protocol support and so on.

The *THttpServer* class in ROOT is a gateway between HTTP engine (implemented in *TCivetweb* class) and the *TRootSniffer* functionality. *THttpServer* class takes care about threads safety: any access to ROOT objects is performed from the main thread only, preventing conflicts with application code. The URL syntax of HTTP requests is used to code objects name and to provide additional arguments. For instance, canvas "c1" created in the application will get address **http://hostname:port/Canvases/c1/** in the HTTP server.

Via the HTTP protocol, following object representation can be requested:

- binary data, produced with *TBuffer*
- JSON string, produced with *TBufferJSON*
- XML string, produced with *TBufferXML*
- PNG/GIF/JPEG image, produced with *TASImage*

For instance, to get the JSON representation of the example canvas "c1", one should submit the following request: **http://hostname:port/Canvases/c1/root.json**.

Access to the HTTP server can be restricted using digest access authentication method [3], supported by most browsers.

D. FastCGI support

FastCGI [4] is a protocol for interfacing interactive programs with web servers like Apache, lighttpd, Microsoft ISS and many others. Contrary to widely used CGI (Common Gateway Interface), FastCGI provides a way to handle many HTTP requests in a persistently running application. From technical point of view, FastCGI creates a TCP server socket used by the web server to deliver HTTP requests and receive response from a local application.

The new *TFastCgi* class of ROOT implements FastCGI protocol as another HTTP engine for *THttpServer* class. In fact, many HTTP engines can run with *THttpServer* simultaneously allowing access to the same data via different protocols.

The compact embeddable Civetweb server in ROOT provides a standalone HTTP server (with all its pros and cons); FastCGI allows integration of arbitrary ROOT application into an existing web infrastructure. The advantages of FastCGI: a common user account management (reuse accounts from web server), a central access configuration (define users who have access to the ROOT application), a central firewall configuration (web server normally situated before firewall), and the use of all features provided by a mature web server.

E. Usage example

In many practical cases no any modification of application code is required to use the web server functionality in ROOT. It is enough to create an instance of *THttpServer* at any time, specifying the HTTP port:

root [0] serv = new THttpServer("http:8080");

root [1] .x $ROOTSYS/tutorials/hsimple.C

If necessary, an application developer could register any additional objects to the server if it is not already contained in the regular ROOT collections:

root [2] gr = new TGraph(10);

root [3] serv->Register("subfolder/graphs", gr);

After *THttpServer* has been started, in a local web browser one can just open page **http://localhost:8080**, where all objects can be browsed and displayed. An example of a web page with several histograms drawn is shown in Fig.3

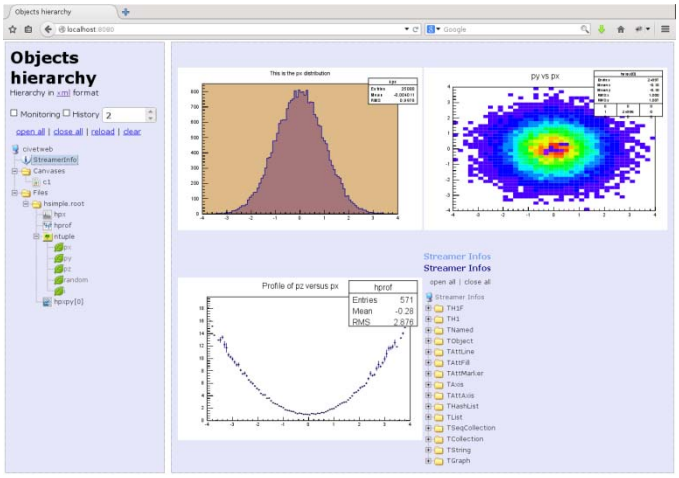

Fig.3: Browser with objects available from hsimple.C macro. The objects' hierarchy is on the left side, and several displayed histograms are on the right. Also a list of streamer-infos is displayed.

Another advantage of the HTTP protocol is that it is supported not only in web browsers, but also in many other tools and applications. In shell scripts it is possible to use tools like *wget* and *curl* to access ROOT objects and their data members. For example:

wget http://localhost:8080/Canvases/c1/root.png

wget http://localhost:8080/Canvases/c1/fTitle/root.json

IV. HTTP SERVER IN DABC

DABC [5, 6] is a data acquisition (DAQ) framework, developed and used in GSI since 2007. DABC offers elaborate mechanisms for multithreading, buffer management, and dataflow organization. The different functionalities are implemented as virtual methods in separate "worker" instances; several workers may share the same thread or run in different threads.

For monitoring and control of framework components a web-based interface has been implemented. The *http::Server* class of DABC follows the same approach as *THttpServer* of ROOT – it provides the gateway between HTTP protocol and the hierarchical structures created by the workers.

A. Hierarchical structures

The *dabc::Hierarchy* class was introduced in DABC for creation and manipulation of complex data structures; such hierarchy is similar to the folders/subfolders/files organization on a hard disk. Every entity in such structure can have a number of named properties (fields). A property can be a simple data type (integer, float, boolean, string) or an array (vector) of simple data types. Several methods are provided to set and get properties values.

An important feature of *dabc::Hierarchy* is version control – any change in a property, or any removing and adding of elements is marked with a new version number. Using version control capability one could always recognize what was changed in the structure relative to a specified version.

Optionally *dabc::Hierarchy* can also record changes of the property values. Later the history of these changes can be used for display of trending plots.

Each worker can publish its internal hierarchy, making it available for other DABC components. *dabc::Publisher class* gathers the registered workers hierarchies into a global one. All elements of such global hierarchy can be accessed via *http::Server* and HTTP protocol. JavaScript code has been developed to organize and display hierarchical structures in a "tree view" representation. An example of such view is shown in Fig.4.

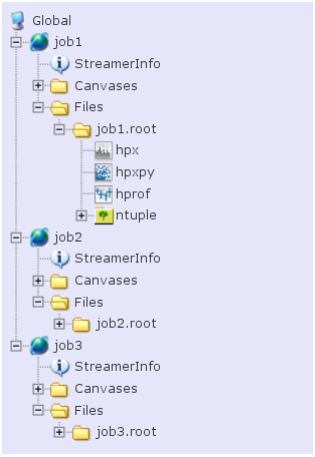

Fig.4: Tree-view display of the hierarchy elements available in a DABC application. Here several ROOT jobs are connected, files with trees can be browsed, and histograms can be displayed.

Different displays are possible for the elements registered in the hierarchy. In a simple case the element properties are printed out as text. ROOT objects are drawn with JSRootIO graphics. Hierarchy elements declared as rate meters are rendered as a gauge. If history recording has been enabled for such rate meter element in *dabc::Hierarchy*, a time trending plot could be displayed.

B. Command interface

There can be special elements in the hierarchy representing command definitions by the application developer. Commands can have an arbitrary number of arguments. The user can

configure minimum, maximum, and default values for each command argument, resp.

When a command is selected in the web browser, the user can specify all arguments and execute such command. In HTTP protocol command execution looks like access to address like:

http://localhost:8080/app/command1/execute?arg=100

On command request a dedicated virtual method will be called in the corresponding DABC worker where the command is to be executed. Afterwards the web server will return an xml string containing the results of execution. The HTTP access to commands can be restricted for authorized users only.

C. MBS support

MBS [7,8] is a well-established DAQ system at GSI that has been used in many experiments for 20 years. DABC provides many components to combine MBS with other kinds of DAQ and slow-control systems.

A worker class *mbs::Monitor* has been implemented in DABC, which can acquire and display status information from running MBS node, e.g. event rate, data rate, and file storage rate. All these values are exported to *dabc::Publisher* and can be observed in the web browser. With most recent MBS version 6.3 such worker also can acquire logging information from the MBS node and execute arbitrary commands on the MBS node.

Thus full control and monitoring of many MBS nodes simultaneously is possible now via web interface provided by DABC.

D. ROOT and Go4 support

The information from *TRootSniffer* class can be seamlessly integrated into the hierarchical DABC structures. A special DABC plugin does such integration and makes ROOT objects accessible via *http::Server* of DABC. This opens up the possibility to integrate ROOT-based applications into a larger system that is combined by means of DABC.

Go4 [9,10] is ROOT-based analysis framework. Its concept consists in the separation of an analysis process executing the Go4 data processing code, from another process that optionally provides an asynchronous graphical user interface (GUI). In addition to the default Go4 GUI that was implemented as Qt and ROOT graphics application, an alternative webserver-based GUI for inspecting analysis objects has been implemented. Now any existing Go4 analysis without any modification can be monitored via web browser.

E. Control system plug-ins

The flexible plug-in architecture of DABC also allows including information delivered from different control systems.

The EPICS [11] plug-in *ezca::Monitor* can read specified IOC records and publish them in the DABC web server. Obtained records can be optionally packed into binary buffers and delivered to an analysis process together with DAQ data. This significantly simplifies implementation of analysis code, because DAQ and slow-control data will come synchronized from a single DABC data source. A similar approach (reading of preconfigured list of records) was used in *fesa::Monitor* plug-in for FESA [12], the CERN/FAIR accelerator control system.

The name server of DIM [13] system delivers a complete list of available DIM records, therefore the corresponding DABC plug-in *dim::Monitor* could provide access to all such records. Through DABC command interface one could also access and execute the available DIM commands.

Thus information from different sources can be combined together and provided to the users via unified interfaces. In Fig. 5 this is illustrated schematically, including some of the control plug-ins described above.

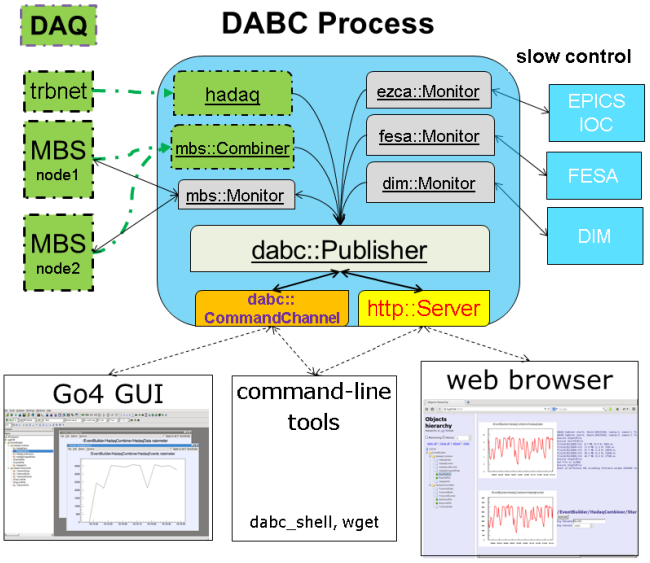

Fig.5: Schematic view of a DABC process publishing monitoring data of several plug-in sources. In addition to various slow control systems, the parameters of external MBS DAQ systems and internal DABC DAQ plug-ins may be monitored. The central *dabc::Publisher* will deliver the monitoring data both to an http server, and to a proprietary command channel socket. Here the variables of interest can be inspected with a web browser, a regular Go4 GUI, or by means of simple command-line tools, resp.

F. Distributed agents

Although DABC application can run many workers in multiple threads, not always all components of a big DAQ system can be read out from a single node. DABC provides a way to build distributed system, where information is acquired on different nodes (agents), but monitoring and control can be performed via web interface on a central "master" node. Master node automatically collects hierarchy descriptions from all agents and provides a combined global description to the clients (web browsers). Communication between master and agents is done by means of a proprietary TCP/IP-based socket protocol. When a browser sends an HTTP request to the webserver on the master node, the request is redirected to the agent, responsible for the specified element. Fig.6 illustrates schematically a typical DABC master-agent set up.

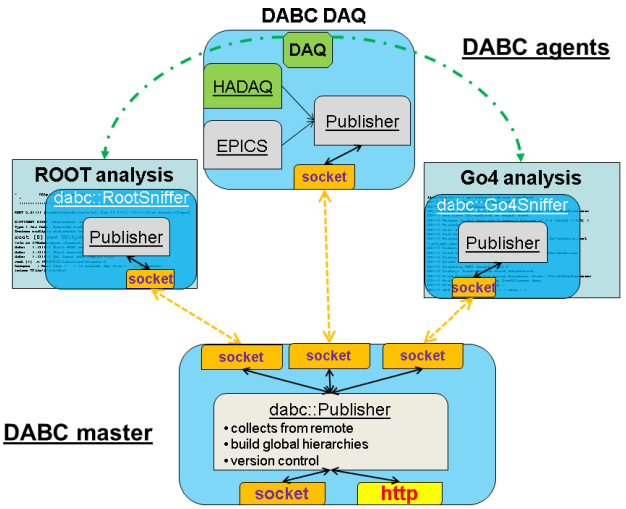

Fig.6: A DABC master process collects monitoring information from different DABC agents: A regular DABC data acquisition process may deliver requested objects of the activated plug-ins as an agent to the master. Using the *RootSniffer* and *Go4Sniffer* classes, any ROOT or Go4 analysis, respectively, may also connect to the DABC master as an agent. So any information published by the agent processes is available at the central master process via http server, or dabc command socket. Even a more complex monitoring hierarchy will be possible if a local DABC master registers via this command socket as an agent to a higher level "super master" process.

An arbitrary number of agents can be connected to the master. A flexible master/agent communication protocol allows at any time stopping/braking an agent and starting it again. Also the master application can be restarted at any time without the need to restart agents. This gives flexibility to dynamically increase/decrease the number of agents without reinitializing the complete system; a failure on a single node will not cause a system-wide error.

Such approach can also be used to monitor distributed ROOT analysis jobs via a central web server, running a master DABC application. Every new job will automatically connect to the master and provide a list of registered objects. Via the web interface one could browse all items of the running jobs and display/monitor any interesting object. Fig.4. shows a web display example with three ROOT jobs, connected to a master node.

V. Conclusion

HTTP access to different kind of online applications is provided. JavaScript code for browsing and display of different object kinds is implemented; JSRootIO is used for ROOT classes. With minimal efforts any existing ROOT application can be equipped with an HTTP server and monitored from any web browser. Using DABC software, heterogeneous distributed systems, including various components, could be steered via web interface.

The code is available in ROOT [14] and DABC [15] repositories.

References

[1] B. Bellenot and S. Linev, "ROOT I/O in JavaScript", J. Phys.: Conf. Ser., Proceedings of CHEP2013, vol 513, in press

[2] Civetweb homepage and repository, https://github.com/bel2125/civetweb

[3] RFC 2617, HTTP Authentication: Basic and Digest Access Authentication, http://tools.ietf.org/html/rfc2617

[4] FastCGI homepage, http://fastcgi.com

[5] DABC homepage, http://dabc.gsi.de

[6] J. Adamczewski-Musch, H.G. Essel, N. Kurz and S. Linev, "Dataflow Engine in DAQ Backbone DABC", IEEE Trans. Nucl. Sci. Vol.57, No.2, pp 614-617, April 2010

[7] MBS homepage, http://daq.gsi.de

[8] H.G. Essel and N. Kurz, „The general purpose data acquisition system MBS", IEEE TNS Vol.47, No.2, pp 337-339, April 2000

[9] Go4 homepage, http://go4.gsi.de

[10] J. Adamczewski-Musch, H.G. Essel, N. Kurz and S. Linev, „Online Object Monitoring With Go4 V4.4", IEEE Trans. Nucl. Sci. Vol.58, No.4, pp 1477-1481, Aug. 2011

[11] EPICS homepage, http://www.aps.anl.gov/epics/

[12] M. Arruat, J.J. Gras, A. Guerrero, S. Jackson, M. Ludwig, J.L. Nougaret, "CERN Front-End Software Architecture for Accelerator Controls", Proceedings of ICALEPS 2013

[13] DIM homepage, http://cern.ch/dim

[14] ROOTDEV git repository, http://root.cern.ch/git/rootdev.git

[15] DABC subversion repository, https://subversion.gsi.de/dabc/