ISBN: 978-3-95450-200-4 doi:10.18429/JACoW-PCaPAC2018-WEP08

A UNIVERSAL SYSTEM BASED ON WEBSOCKET AND JSON FOR THE EMPLOYMENT OF LabVIEW EXTERNAL DRIVERS

Alessandro Stecchi, Claudio Bisegni, Paolo Ciuffetti, Giampiero Di Pirro, Alessandro D'Uffizi, Francesco Galletti, Andrea Michelotti (INFN/LNF, Frascati (Roma))

Abstract

One of the heaviest workloads when installing a Control System on a plant is the development of a large number of device drivers. This is even more true in the case of scientific facilities for which you typically deal with many custom devices and legacy code. In these cases, it is useful to consider the Rapid Application Development (RAD) approach that consists in lessen the planning phase and give more emphasis on an adaptive process, so that software prototypes can be successfully used in addition to or in place of design specifications. LabVIEW [1] is a typical RAD oriented development tool and is widely used in technical laboratories where many standalone programs are developed to manage devices under construction or evaluation. An original system that allows software clients to use external LabVIEW drivers is presented. This system, originally created for the !CHAOS Control System [2], is entirely written in LabVIEW and is based on JSON messages transmitted on a WebSocket communication driving LabVIEW VIs through dynamic calls. This system is completely decoupled from the client and is therefore suitable for any Control System.

DESCRIPTION OF THE ELF SYSTEM

The project called ELF (External LabVIEW Functions executor) stemmed from the need to reuse as much as possible the huge amount of software — especially device drivers — already written in LabVIEW, to speed up the implementation of new controls currently being implemented with !CHAOS, a control framework developed at the National Laboratories of Frascati of the INFN.

As the project evolved, its usefulness was also evident not only for the re-use of existing LabVIEW Virtual Instruments (VIs) but also as a proficient method for the collaborative development of complex LabVIEW programs. In fact ELF provides for the VIs acting as drivers to be called by reference and not be wired into other G code, which greatly facilitates the team development.

Ultimately, this work consists in the realization of an environment able to have LabVIEW VIs execute in a well managed and standardized manner, upon calls coming from a client application written in any language.

The project requirements where:

- to adopt widely used communication protocol and data-interchange format for the communication between the client and ELF;
- to write the whole ELF code in G: the LabVIEW graphical programming language;
- to be able to deal with simultaneous calls from multiple clients;
- to adopt a unique template for the VIs to be executed and a well defined syntax for their call;
- to realize a modular architecture, allowing its usage both from non-LabVIEW and LabVIEW clients;
- to get such performance that it could be used in a wide range of applications.

It is worth to point out that the ELF employment is non restricted to the call of device drivers but extends to any VIs complying with the adopted template and syntax.

COMMUNICATION BETWEEN CLIENT APPLICATIONS AND ELF

First we must clarify what is meant — in this context — for client application (*client* from now on). We therefore define as client any program that accesses the ELF to perform a set of predefined functions implemented in LabVIEW and obtain results back, if any.

As an example, in our case the typical client is a !CHAOS Control Unit (CU): the control node that continuously acquire data from a device and operate it. To access a physical device, the CU performs a RPC by using a C++ *skeleton* that relays the function *opcode* along with its arguments to a *stub* that eventually runs the actual driver. In !CHAOS, drivers are usually implemented as C++ programs but, if they are available as LabVIEW VIs, the *skeleton* can direct the calls towards the ELF system which acts as *stub*.

The communication between a client and ELF meets the client/server model, with ELF playing the role of server, and follows the WebSocket [3] protocol.

WebSocket provides full-duplex communication between nodes over a single TCP connection with data minimally framed, to the benefit of real-time data transfer.

It provides a standardized way for the server to send content to the client without being first requested, and allowes messages to be passed back and forth while keeping the connection open. The WebSocket handshake starts with an HTTP request/response then — once the connection is established — the communication switches to a bidirectional binary protocol which doesn't longer conform to HTTP. This method is advantageous since HTTP *flows* through proxies and therefore it is possible to open connections even from remote clients laying within a NAT, which is very convenient for distributed system where remote control nodes can be spread anywhere (as in the case of !CHAOS).

As data-interchange format it has been adopted JSON (JavaScript Object Notation) because it is lightweight and also because JSON documents are ultimately strings, which facilitates the passage of different data formats to LabVIEW VIs.

ISBN: 978-3-95450-200-4 doi:10.18429/JACoW-PCaPAC2018-WEP08

We built the ELF as a modular system made of two LabVIEW top-level VIs working jointly. The first module (named as EPROS: Elf Proxy Server) is a proxy server that connects in WebSocket to the client and interfaces it to the second module (named as SHIELD: Simple High-level Interface for Enhanced LabVIEW Drivers), where the various functions are executed. EPROS and SHIELD talk to each other by mean of JSON documents passed through LabVIEW queues.

This architecture, shown in Fig. 1, allows two different modes to access the ELF system:

- Indirect Mode: a client not written in G or even written in G but running in a LabVIEW session different from the ELF one's, will utilize WebSocket/JSON and both the EPROS and SHIELD modules;
- Direct Mode: should the client be a LabVIEW program running in the very same session of the ELF, it would be useless to go through WebSocket, being much more convenient to utilize native LabVIEW queues. In this case, the client will use just the SHIELD module and exchange the JSON messages directly through its queues.

VIs PREPARATION AND INSTALLATION

When dealing with legacy software or even software written by more than one person, it is essential to define strict guidelines that allow for its integration.

In ELF, the integration of drivers for a given device, provides for the establishment of a list of functions (namely the different actions that have to be performed on the device) and for each of these, the set of arguments to be passed and the returned results. Each function is identified by an *opcode*.

Once the opcode list has been defined, the LabVIEW developer has to prepare for each opcode a dedicated VI able to perform its function. All these opcode VIs must adhere to a template and have the same connector pane, with:

- a string input for the JSON document containing the parameters;
- a cluster containing all the variables (named as *service bus*) that the developer considers to be meaningful to describe the status of the device;
- a string output for the JSON document containing the results (or errors) to be returned to the client.

If the developer writes the opcode VIs starting from scratch, then he will edit them directly according to the template, otherwise he will have to readjust the code he already owns, for example by encapsulating it in *wrapper* VIs compliant with the template.

Once the opcodes have been prepared, they must be *installed* in the ELF system. As already mentioned, ELF is designed to completely decouple the opcode VIs both from the client and from itself. Therefore, the opcode VIs installation comes down to their copy in a predefined directory tree, so that they can be located and launched *by reference* by the SHIELD in accordance with the JSON directives coming from the client (see Fig. 1).

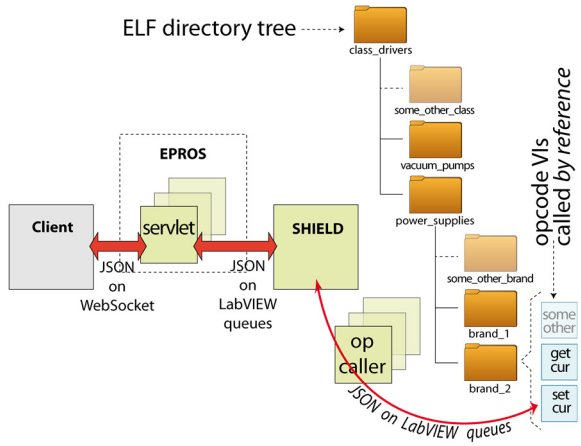

Figure 1: The ELF system. The installation of drivers (compliant with the ELF guideline) comes down to their copy in a directory of the ELF tree.

The ELF directory tree mirrors the grouping of devices into abstract classes.

A class of devices is set out by a virtualization process that — starting from a physical object — concludes with the definition of a set of meaningful variables (*dataset*) that fully describe it from the vantage point of the control and a set of actions (*commands*) you want to perform on it. In that way, we can bring together — for instance — different brands/models/types of power supplies by identifying them with the same dataset and commands. Therefore, the directory tree shall consist of a first level of directories corresponding to the abstract classes of devices, each of which contains other directories that are specific to the different brands/models/types of real power supplies.

Ultimately, the developer of opcode VIs has to prepare them in accordance with the specifications and drop them in the proper folder. It is important to point out that, although the directory structure is defined, its content is not. So, if no branch of directories is available for the class of elements to be installed, the developer is free to create it and populate it with its opcode VIs.

At the end, the new set of opcode VIs can be located by mean of a *uri* in the form:

<base_path>/class_drivers/power_supplies/brand_1/

CLIENT-ELF WORKFLOW

There is a slight difference in the sequence of actions a client must perform, according to whether he accesses the ELF system in direct or indirect mode. Direct access is in fact reserved for clients running in the same LabVIEW session of ELF (more precisely, of SHIELD). This means that both client and SHIELD are running on the very same machine, which makes the connection and authentication steps unnecessary. The workflow for indirect access is therefore described, since it also covers the direct mode.

WebSocket Connection

The client requires a WebSocket connection to the EPROS by sending a WebSocket header (this is a pure HTTP request) with the *uri* that identifies the drivers for the desired class. The EPROS checks the header format and

ISBN: 978-3-95450-200-4 doi:10.18429/JACoW-PCaPAC2018-WEP08

asks the SHIELD to confirm that the required VIs exist. If no errors occur, the EPROS accepts and establishes the connection.

After receiving the HTTP header, all the subsequent phases of the workflow are carried on by exchanging proprietary JSON messages.

Authentication

The client sends an authentication request containing a shared key; the EPROS checks the key and if no errors occur, grants access to the client. In order to serve concurrently multiple clients, the EPROS starts *by reference* an instance of servlet dedicated to that very client. From this moment on, the EPROS servlet behaves as a PROXY, since it limits itself to relay back and forth the messages between the client and the SHIELD and to manage any communication error.

Initialization

When a client transmits an initialization request with the *uri*, the SHIELD opens an instance of a VI (named as *op-caller*) that will manage all the opcode requests for the given device. It also creates a UUID associated to the instance of the op-caller and returns it back, appended to the uri. The concatenation of the uri with the UUID is essential to be able to later discriminate equal instances but related to different devices of the same class. At this point SHIELD use the op-caller to launch *by reference* the opcode VI that performs the initialization of the physical device and — where appropriate — of the channel it communicates through.

Full Duplex Data Flow

This is the operational phase, where the client has the ELF performing the desired actions by sending JSON request messages in the form:

```
{
  "req_id":[int32],
  "msg":{
          "uri":[string],
          "opc":[string],
          "par":{...json document...}
  }
}
```

where:

- "req_id" is a transaction identifier;
- "uri" identifies the specific op-caller instance;
- "opc" specifies the action the external driver has to do;
- "ele" identifies the element (e.g. its address);
- "par" is a JSON document containing the parameters associated to the opcode and the specific element identifier (e.g. its address).

For instance, the message:

{"req_id":123456,"msg":{"uri":"/LabVIEW_external_drivers/power_supplies/brand_1/FFA0C0D8","opc":"set_cur","par":{"ele":"01", "value":10.5}}}

calls the opcode "set_cur" with the parameter "value" equal to 10.5, which cause to set the power supply current to 10.5 [A].

At each request message, SHIELD handles the client call by relaying it to the op-caller that calls by reference the appropriate opcode VI.

All client requests are always answered with frames in the form:

```
{
  "req_id":[int32],
  "msg":{
          "err":[int32],
          "err_msg":[string],
          "err_dmn":[string],
          "result":{...json document...}
  }
}
```

where:

- "req_id" is a transaction identifier that has to be equal to the transmitted one;
- "err" is the error code (0 = no error);
- "err_msg" describe the error (present only if err ≠0);
- "err_dmn" is the error domain (present only if err ≠0);
- "result" is a JSON document containing the result returned by the opcode VI. This field is present only if the opcode VI provides for some values to be returned.

Deinitialization

The client asks for the deinitialization of the physical device (if applicable) and the release of associated resource. Consequently, the SHIEL destroys the op-caller instance as well as all the related opcode VIs hanging in memory and remove the UUID from its internal list.

WebSocket Disconnection

The client sends a standard WebSocket "close connection" message which is answered by the EPROS that eventually will also drop the TCP/IP connection.

EPROS AND SHIELD

The EPROS Module

As said above, a WebSocket connection starts with an HTTP request and then switches to WebSocket over the same underlying TCP/IP connection. This means that on the server side there must be an always running TCP listener, to accept requests from various potential clients, as well as various WebSocket servers running, to service those already connected clients. The EPROS is therefore made of a top level VI that keeps listening and, when a TCP connection request arrives from a client, launches by reference an instance of a WebSocket server that take charge of that client. This instance (named as *servlet*) is a replica of a VI Template (VIT) and has a dataspace completely independent from the other servlets.

The servlet is made of four parallel loops data-independent form one another that can concurrently execute the following tasks:

- check and serve WebSocket connection and authentication requests;
- receive WebSocket request messages;
- communicate with the SHIELD through LabVIEW queues;
- send WebSocket answer messages to the client.

ISBN: 978-3-95450-200-4 doi:10.18429/JACoW-PCaPAC2018-WEP08

The four loops pass information to one another by using queues, which is fast and maintain the data-independency among them.

The low level WebSocket read and write functions have been written from scratch and cover the following WebSocket actions: read text frame (also multiframes), ping, pong, close connection.

The SHIELD Module

The SHIELD must be able to manage multiple clients at the same time. To do that,

when a client asks for the initialization of a given device, it launches by reference a dedicated instance of op-caller that will handle all the calls for that device. There is a one-to-one correspondence among EPROS servlets and SHIELD op-callers, being each pair the two endpoints of a channel dedicated to a specific device.

When a call with a given opcode arrives for the first time, the op-caller starts by reference the opcode VI which is in the folder specified by the uri. The opcode VI executes the action and then stays idle in memory, ready for the next call with the same opcode. To permit this, all the opcode VIs must be set as "pre-allocated clone reentrant execution" which provides for a mutual data-independency and no loading time at call.

The SHIELD provides also for the garbage collection of dandling instances.

SYSTEM PERFORMANCES

The ELF system has been tested to understand its potential fields of application.

We therefore made some measurements to determine the frequency of execution of an opcode VI upon continuous calls coming from a client.

This figure depends on the client performance, the rank of the machines running both client and ELF system, the network and obviously the intrinsic execution time of the opcode VI that can vary widely. An opcode VI could in fact run on a remote machine and having to drive a device connected through low baud-rate serial line or run on embedded processor and having to simply set a register of a board resident in its own bus.

The measurement was therefore made net of the opcode execution time, by using a dummy VI that always returned a constant value to the get_curr (get current) opcode.

The client application was a standard !CHAOS Control Unit running on a virtual machine and the ELF system was running on another virtual machine, with ping time between the two of ~500 us. The read process can be reduced to three main steps:

- the client sends a WebSocket message with the opcode "get_curr" to the servlet;
- the opcode VI executes and returns the value to the servlet;
- the servlet sends a WebSocket message with the result back to the client.

Therefore, the full query consists of two WebSocket transmissions that are performed synchronously by the Control Unit. In this condition the overhead for the query is ~1 ms. The measure has been done by varying an internal parameter of the CU (sleep-time) that allows the modulation of its running frequency. It can be seen in Fig. 2 that as the CU sleep-time decreases the frequency of calls increases up to 1 kHz, which is consistent with the overhead of 1 ms due to network latency.

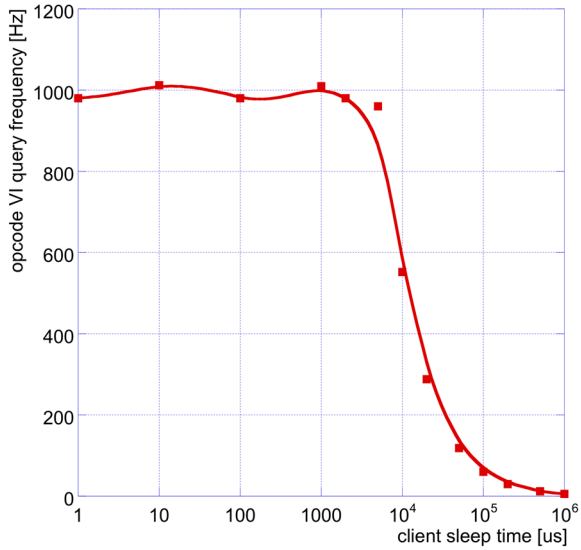

Figure 2: Frequency of calls of a LabVIEW driver.

The sharp breakthrough at 1 kHz must not be considered as a limit of the ELF system because it is due to the CU running synchronously and therefore executing the queries in a series. We are going to do more tests with asynchronous calls in order to work out the actual limits of the system.

The SHIELD module has also been tested in direct mode using a minimal LabVIEW client written on purpose. In this case the process takes 100% of the CPU and the frequency dramatically increases, according to the CPU performances.

REFERENCES

[1] LabVIEW, http://ni.com

[2] L. Catani *et al*, "Introducing a New Paradigm for Accelerators and Large Experimental Apparatus Control Systems", *Phys. Rev. ST Accel. Beams,* Nov. 2012, vol. 15, p. 112804.

[3] WebSocket, https://tools.ietf.org/html/rfc6455