Extending LabVIEW Aptitude for Distributed Controls and Data Acquisition

Luciano Catani
Istituto Nazionale di Fisica Nucleare (INFN) - Sezione Roma Tor Vergata
Italy

1. Introduction

LabVIEW is probably the most comprehensive environment for setting up a control/data acquisition system (CS) for a scientific/laboratory experiment. It provides ready to use solutions for both control and data acquisition for a large number of equipments and for the analysis of various types of data.
In the scientific environment, especially, these features are particularly useful because CS are frequently developed and managed by scientists willing to spend more time in operating their experimental apparatuses than in maintaining software for controlling components and reading instruments outputs.
In a scientific laboratory, or a medium/small experimental apparatus, typical selection of equipments is unavoidably heterogeneous: scopes, digital I/Os, motors, digital cameras and, quite often, a mixture of new and relatively old technologies for connecting devices to the computer managing the system.
Moreover, CS for scientific experiments are quite often far from being developed once forever; instead they are continuously updated by replacing and/or introducing new components in order to follow the evolution of the apparatus.
LabVIEW easy to learn graphic programming and its large number of instrument drivers and libraries for data analysis and graphical display, successfully fulfill the above requirements.
When the scientific apparatus became larger and more complex a single computer may not be sufficient for the management of all the components. Additionally, in some environment the equipments need to be operated from remote or it might be preferable to separate data acquisition from on-line analysis in order to optimize the performances of both.
In other word the single computer CS needs to be upgraded to implement a distributed control system (DCS).
LabVIEW provides quite a number of solutions also for the development of DCS by offering tools for remote control of Virtual Instruments (VI) and sharing of data across the network by means of dedicated LabVIEW components that allow communicating with remote computers and devices.
Developers can easily find ready to use solutions for their needs among these resources although, in some cases, they might either lack in flexibility or cannot offer the required compatibility with all the software components of the DCS.
In these situations a communication solution for the DCS should be necessarily based on the widely accepted standard protocols ensuring highest compatibility.

There exist numerous possible alternatives. One such model is Microsoft's component object model COM(mscom, 2011) and its associated distributed component object model DCOM which allows COM objects to run and communicate in a distributed manner. Also from Microsoft is the .NET environment, supporting Internet-based connectivity between components.
Another option is offered by the CORBA(corba, 2008) proposed by the Object Management Group (OMG). CORBA is a software standard for component-based development that has been quite successful among developers of DCS.
Yet another model is the Java-based proposal by Sun Microsystems, which encompasses basic infrastructure such as Java Beans and Enterprise Java Beans and remote method invocation (RMI) but also more ambitious solutions for interoperation of distributed intelligent systems such as Jini(jini, 2006).
The aim of this paper is to present the development of a communication framework for distributed control and data acquisition systems, optimized for its application to LabVIEW distributed controls, but also open and compatible with other programming languages because it is based on standard communication protocols and standard data serialization methods. In the next Paragraph the LabVIEW tools for Distributed Systems and their field of application will be briefly presented. In Paragraph 3 the general purpose communication framework will be discussed, with particular attention to the problem of data serialization and the definition of the communication protocol. Paragraph 4 and subsequents will discuss in details the implementation of the communication framework in LabVIEW, focusing on the development strategies and the solutions for achieving the performances required for this field of application.

2. LabVIEW tools for distributed systems

LabVIEW offers a number of tools for transferring, via network, data between the components of a distributed control/data acquisition system.
The Table 1 shows some solutions provided built-in by LabVIEW, a subset of the networking features suggested by National Instruments for developing distributed control systems (Lima et al., 2004). Common to all these solutions is the ease of implementation because they have been designed to require very limited programming effort.
Shared Network Variable, DataSocket, VI Server, VI reference, TCP/IP and UDP communication libraries and also interfaces to .NET and ActiveX are the main communication tools offered by LabVIEW. They are powerful and well suited for many applications but, with the exception of TCP/IP and UDP, are not flexible enough to allow the implementation of a real communication protocol.
Shared Variable and Data Socket, for instance, basically provide data sharing across the network, others (VI Server, VI reference) allow access to remote VI, with some relevant restrictions in some cases, and their use is limited to LabVIEW environment.
In addition the LabVIEV Internet Toolkit includes a HTTP server and the possibility to operate the VIs (Virtual Instruments, i.e. the LabVIEW applications or subroutines) as CGIs that a client can invoke using the HTTP protocol to execute particular procedure on the server side. This is a relatively flexible solution but offers low performances and limited features.
In conclusion it's hard to find the best candidate for developing an open, general purpose communication framework because the before mentioned solutions are either offering limited features or performance, or they are proprietary so that, for instance, integration with the control system of a nearby experiment or with a bigger apparatus, that could have been

Networking Feature	Use Case	Programming Required	Multiple Writers/ Readers	Transmission Delay	Transfer Rate
Shared Network Variable	Share live data with other VIs on a remote computer, or deployed to a target.	No	Many-to-Many	Low	High
DataSocket	Share live data with other VIs on a remote computer, or deployed to a target.	Yes	Many-to-Many	Low	High
Application Control VIs and Functions	Programmatically control VIs and LabVIEW applications across a network by way of the TCP protocol and VI Server.	Yes	One-to-one	Medium	Medium
Remote Front Panels on the LabVIEW Web Server	View and control a VI front panel remotely using LabVIEW or a Web browser.	No	One-to-Many	Medium	Low
Web Services on the Application Web Server	Deploy LabVIEW applications as Web services.	No	Many-to-Many	Medium	Low
HTTP Client VIs	Build a Web client that interacts with servers, Web pages, and Web services.	Yes	One-to-Many	Medium	Low
TCP VIs and Functions	Communicate with an instrument that uses a protocol based on TCP.	Yes	One-to-One	Medium	High
UDP VIs and Functions	Communicate with a software package that uses a protocol based on UDP.	Yes	One-to-Many	Low	High

Table 1. LabVIEW most relevant communication features (excertp from Communication Features table from LabVIEW on-line manual).

developed with different software solutions, would be hard to manage or even impossible.
On the other side, an example of requirements for the communication framework is given by the list of communication modes it should allow, like for instance the four cases shown in Fig.1.

The *case 1* is the typical client/server model: a client application asks for data to a remote controller and receives from it the required information that could be, for instance, the most recent value of an I/O channel, the history of that value, a bundle of data of different types.
The *case 2* is similar to the remote procedure call (RPC) model: a client application needs to execute a subroutine or procedure on a device attached to another controller without the programmer explicitly coding the details for this remote interaction. It could be the change in the set point of a remote equipment, the execution of a measurement. etc.
For the above mentioned modes of communication a programmer can relay on the predictability of type of data handled and on the limited set of action to be taken on the server and client side. It that would have been the case, DataSocket or VI Server will probably solve

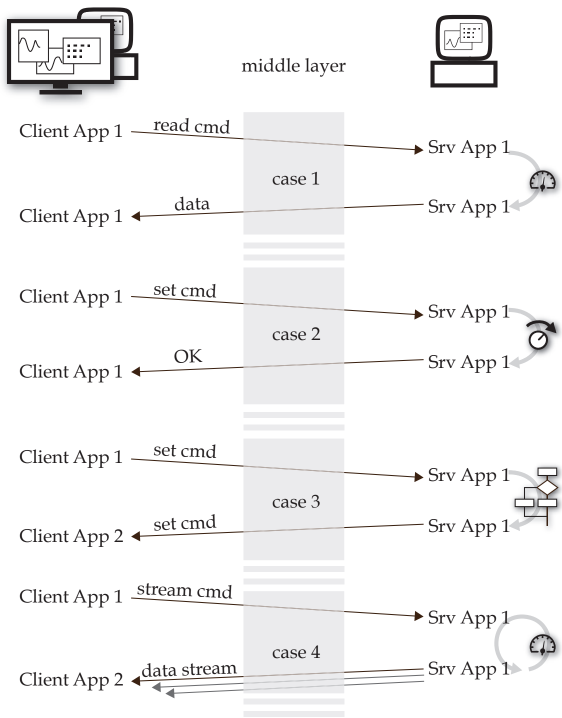


Fig. 1. Modalities of interaction between components of DCS to be supported by the communication framework.

the communication problem.
The *case 3* and the following *case 4* introduce a more elaborated way of interaction. In the first one a request originated from a client application in turn requires an execution of a command on the client side according to the response received by the remote unit. The *case 4* extends the example in *case 1* to another modality of communication that is a continuous stream of data from a server application upon request from the client.
These are just few examples but they are sufficient to confirm and strengthen the requirement for a communication framework that should be based on a versatile and well established communication solutions, to achieve high compatibility and, at the same time, it should allow information to be transferred as different format, i.e. requests or commands, responses and data of different type.

3. A general purpose communication framework

When compatibility and flexibility is an issue, TCP/IP and UDP socket communication might be the natural choice. Network socket communication libraries are available for all main programming languages and the simplicity of the protocol ensures a wide compatibility.
Indeed, LabVIEW provides tools for interfacing with other devices on a TCP and UDP network with standard socket communication protocol by means of TCP/IP and UDP VI and functions.

The set of functions LabVIEW provides for TCP/IP and UDP communications, similarly to other implementations of network socket libraries, supports few elementary operation: *open, close, listen, read* and *write*.

The first three are used to establish connections between client and server, the last two for transferring data in the form of buffers of given length (VI in the *Networking/TCP&UDP* section of LabVIEW examples show some possible implementations of network socket communication).

If data to be transferred is not as simple as either a string of characters or an array of bytes, the communication framework should be equipped with tools for packaging and parsing data, regardless their type, size and complexity.

This process, known as serialization, converts any complex data structure into series of bits that can be easily transmitted across a network connection link and later restored in the original or equivalent form. This result is achieved by adding some kind of descriptor (meta-data) to the payload. Hopefully, the impact of meta-data on the size of serialized data structure and the coding/parsing execution time on the overall data throughput should be limited.

Different types of serialization strategies can be used to flatten object(s) into a one-dimensional stream of bits suited for their transmission by means of socket communication functions.

XML (xml, 2000) (eXtensible Markup Language) is a popular way of coding data especially when interoperability and compatibility between platforms and programming languages is an issue. Client/server communication protocols based on this coding exist, among these the more interesting are SOAP(soap, 2007) and XML-RPC(xmlrpc, 1999). The latter it's basically a remote procedure call(rpc, 1988) that uses HTTP, or other TCP/IP and UDP protocols, as the transport and XML to serialize data allowing complex, and relatively large, data structures to be transmitted and then un-marshaled at destination.

Services provided by the server are called *methods* that a client can invoke by issuing `method Call` to the server. The latter, in turn, replies sending the result in the form of `method Response`. Fig.2 shows an example of messages passed between a client and a server in the XML-RPC protocol. They include header with declarations and `methodCall` or `method Response` fields.

The `methodCall` contains, enclosed with the correspondent tags, the name of the method to be executed on the server side (`methodName`) and optional parameters (`params`). The `methodResponse`, being the reply message from the server to the client contains, enclosed by the (`params`) tag, the data produce by the execution of the *methodName*.

XML-RPC can be easily implemented in LabVIEW by using the before mentioned socket communication libraries; it will be discussed in details in the following Paragraphs.

Socket communication sessions are defined upon the couple IP address and port number that are chosen, on the local and remote computer, for that particular session. That means in a distributed control system with several computers the local application willing to send a command or receive data to/from another application running on a remote computer should be informed on the IP address and the port number that is used by the remote application for listening incoming connections, or equipped with instruments permitting to obtain this information from some kind of repository.

Actually, from the client application point of view, i.e. display consoles, measurement application etc. the DCS should better be seen as a distributed set of components: actuators, diagnostic components, equipments etc. and client applications could be unaware of their physical location or the details of the communication protocol.

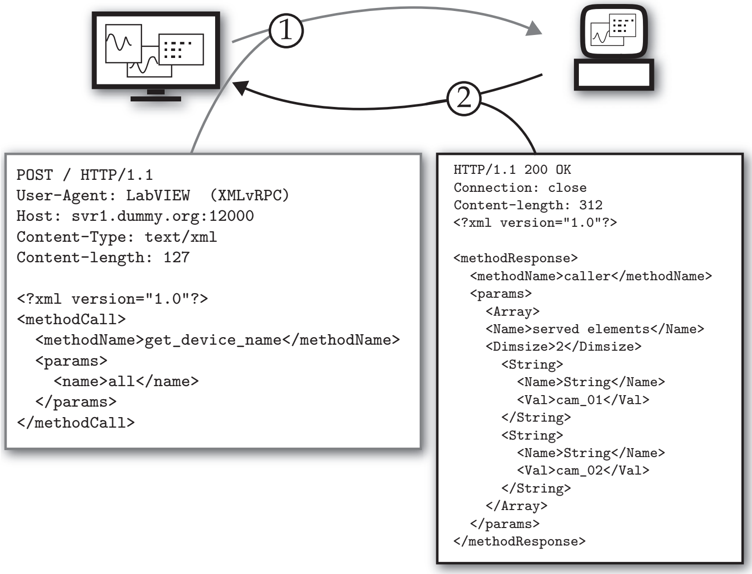


Fig. 2. `methodCall` (left) and `methodResponse` (right) in the XMLvRPC protocol.

In order to be addressed uniformly components need to be integrated in a standardized way such that the communication framework can be developed on top of a generic component model.
It means the network socket communication libraries should be the basis for a general purpose communication framework, a middle-layer between the top level with display or client programs and the front-end layer with device controllers, providing a simplified access to data and commands transfer across the network by hiding the transport layer implementation details.
In the next paragraphs definition of the middle layer and data serialization will be discussed in details.

4. Distributed controls with XMLvRPC

Although it has been basically developed for web services, XML-RPC provides a number of features that fit with the requirements of a simple and flexible communication framework for the distributed control system under development. In particular XML-RPC:

- uses a well established human readable data serialization
- can be easily implemented in LabVIEW by using the standard TCP/IP and UDP libraries
- allows a good flexibility in defining the communication between client and server
- offers high compatibility having a large number of implementations with different programming languages

A communication protocol named XMLvRPC, based on XML-RPC and optimized for LabVIEW distributed controls, was introduced by the author in a previous paper (Catani, 2008).

4.1 Client/server communications

The main components of the XMLvRPC protocol are the *XMLvRPC_Server.vi and XMLvRPC _Client.vi*. Fig.3 schematically shows an example of a data request from a client application to a XMLvRPC server running on a remote controller.
The client application calls the *XMLvRPC_Client.vi* providing the TCP socket information that identify the server side, the remote method to be invoked and optional parameters. The *XMLvRPC_Client.vi* encode the information as a standard XML-RPC call and send it to the server specified (1).
The *XMLvRPC_Server.vi* extract the `methodName` and call the correspondent VI providing "as it is" the information enclosed by the `params` tags in the `methodCall` (2). The VI that implements `methodName` is instructed to parse the data in `params`; it executes its task accordingly and replies to *XMLvRPC_Server.vi* that encode the information in a `methodResponse` that is finally returned to XMLvRPC_Client.vi (3).
As final step the *XMLvRPC_Client.vi* outputs to the calling application the content of `params` enclosed in the `methodResponse`.
At this point XMLvRPC shows a first difference respect to standard XML-RPC.
While the latter always assumes, at least so far, that `params` returned from the server will be directly used by the calling application, XMLvRPC allows the *XMLvRPC_Client.vi* to dynamically call another application, different from the one that issued the `methodCall`, for handling the data received from the remote server.
This is possible because, similarly to `methodCall`, also `methodResponse` includes a `methodName` field for specifying the application (a LabVIEW VI, in this case) that must be invoked to handle the enclosed data.
For this purpose, a number of different solutions can be implemented according to user's needs: all *method.vi* can be either pre-loaded at start-up to optimize execution time or loaded when called and released after execution or optionally cached in memory. Since *method.vi* are programmatically loaded and run, this also means that when a new method is added to a server (similarly on a client) the server source-code doesn't need to be modified to include the call to this new VI. It will be sufficient to copy the VI that serves this new method to the directory where the server *XMLvRPC_server.vi* searches for the *method.vi* implementing the particular `methodCall` requested from the client.
This feature simplifies implementation of new methods: once the client and the server side routines (i.e. LabVIEW VIs) of the method have been developed, they just need to be copied into the specified directories to be immediately available to the control system.
Fig.4 provides more information about the *XMLvRPC_Server.vi* by showing a portion of its block diagram where the main steps of execution are presented.
Let assume that during phase (0) the server has been listening for connection requests. When it finally established a connection after a client request, in (1) the *XMLvRPC_Read_request.vi*

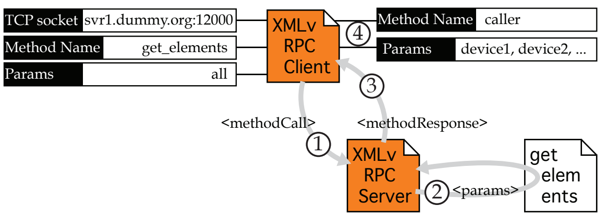


Fig. 3. Main components in a XMLvRPC client/server communication.

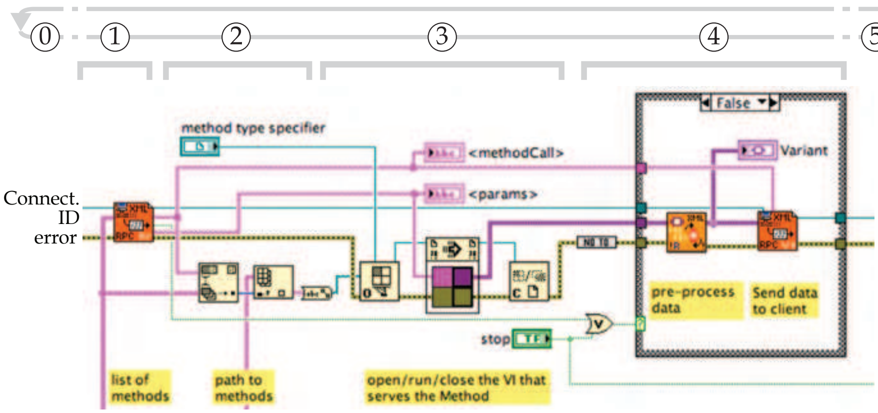


Fig. 4. Portion of block diagram of the the XMLvRPC_Server.vi showing the execution of the server loop.

receives the `methodCall` from the client and parses it, looking for `methodName` and `params`. The `methodName` is then used (2) to find out the full path of the VI that serves that particular method, that is expected to be stored in a dedicated directory with the other *methods* available for that particular server. The full path allows to dynamically load and run the target VI with `Call by Reference Node`.
This solution simplifies very much the server's structure. Methods don't need to be placed directly on the block diagram provided they all have the same connector pane because the `Call By Reference Node` requires a strictly typed VI refnum.
Fortunately this isn't a severe limitation. In fact, since data provided as input for the method execution are serialized onto the `params` string of the XML coding, for VI implementing any of XMLvRPC methods, basically, only one input connector (a String control) is sufficient. Similarly, data produced by the methods, either a single value or a complex data structure, are returned from a single output connector.
In the following paragraph it will be explained why a Variant indicator is used as output instead of a String data type. At this point it is sufficient to mention that Variant data do not conform to a specific data type allowing a program to pass it from one VI to another without specifying what kind of data type it is at compile time.
In LabVIEW, variant data type differs from other data types because it stores the control or indicator name, the information about the data type from which was converted, and the data itself, allowing to correctly convert the variant data type back to the original or to another one.
As for the input connector, the Variant allows methods VIs to output any type of data, or combination of thereof, after the `Call By Reference Node`.
For that reason the *XMLvRPC_Server.vi* is a generic server for requests issued by clients and it doesn't need to be specialized for a particular controller, i.e. for a particular set of tasks to be executed or components to be controlled, because the methods are not statically linked subVI calls. The VIs implementing the methods only need to be available at run time, ready to be loaded and executed upon request of the remote client.
Block diagram of *XMLvRPC_Client.vi* is even simpler, as shown in Fig.5 (next page).
In conclusion XMLvRPC client and server are based on four symmetric functions.

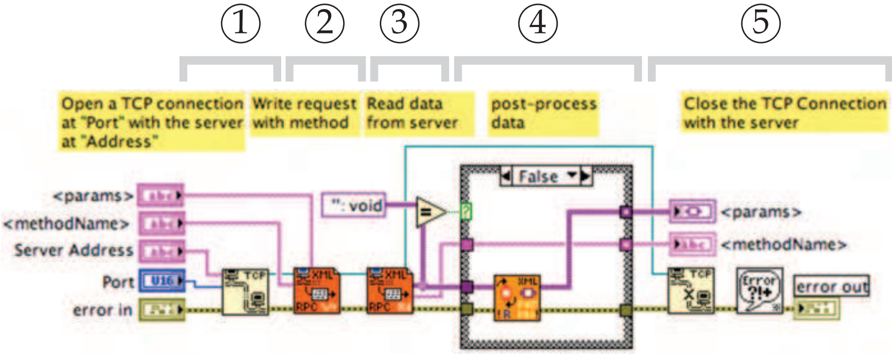


Fig. 5. Block diagram of the the XMLvRPC_Client.vi.

The VIs implementing these functions are, on the server side, *XMLvRPC_ReadRequest.vi* and *XMLvRPC_WriteResponse.vi*, on the client side, *XMLvRPC_WriteRequest.vi* and *XMLvRPC_ReadResponse.vi*.
These VIs, depending to their specific function, perform coding or parsing of XML request or response or network socket read or write operations. The block diagram of *XMLvRPC_WriteRequest.vi* is shown as example in (Fig.6 next page).

4.2 Data serialization

Before going into details of data serialization for XMLvRPC, it should be noted that XML is not the unique choice for providing a human-readable serialization of a given data structure. Another option for text-based serialization is, for instance, JSON(JSON, 2009) derived from Java Script syntax and, as well as XML, well supported from many programming languages. Compared to XML, JSON is more lightweight though, probably, a bit less readable.
Similarly to XML-RPC, a remote procedure call protocol based on JSON encoding has been proposed with the name of JSON-RPC(JSONRPC, 2009). The XML-RPC `methodCall` and `methodResponse` examples shown in Fig.2 would translate to JSON-RPC as the following:
Request from Client: `{"jsonrpc": "2.0", "method": "get_elements", "params":[all], "id": 1}`
Response from Server: `{"jsonrpc": "2.0", "result": ["cam_01", "cam_02"], "id": 1}`
Clearly JSON coding, being less verbose and more compact, provides a clear advantage with respect to XML when used for web services.
However, taking into account all the possible data structures that are routinely transferred across the network in the case of distributed controls for scientific applications, neither XML nor JSON can address the crucial problem for the final size of serialized data that is the coding of large binary arrays that client applications may receive from some particular devices.
Typical example could be the read out of the buffer of a digital scope, consisting of few hundreds of floating point values or, what's worse, raw images produced by a digital monochrome camera consisting of hundreds of thousand pixels, eight or more bits each.
In this case, the text-based serialization produced by either JSON or XML could be really unfavorable because of the large number of single values to code and, especially, because of the much larger size of the serialized data with respect to the original.

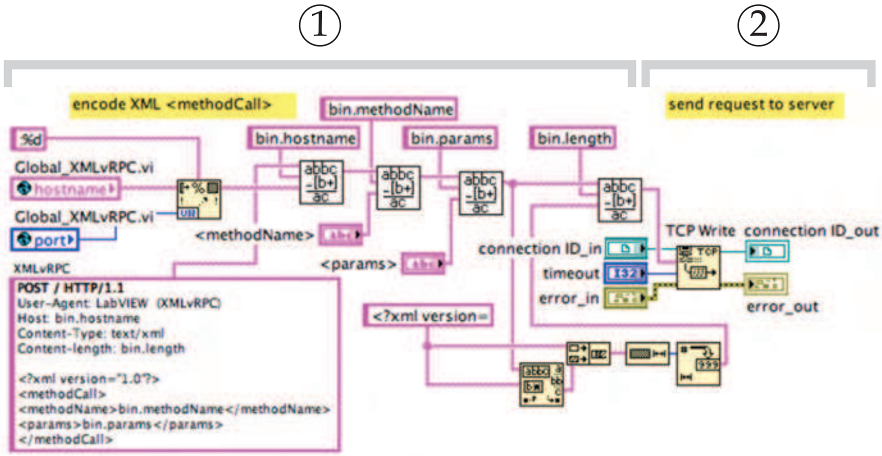


Fig. 6. Block diagram of the the XMLvRPC_WriteRequest.vi.

4.2.1 Binary arrays management

Embedding of binaries in XML format has different options. Binary data, for instance, can be enclosed with the XML `CDATA` tag, a special tag for processing data that isn't going to be parsed during XML processing.
Unfortunately, this method is not perfectly safe and might lead to messy results as, for instance, when binary data contains the `]]>` sequence, which would indicate to the XML parser the end of the non parsed data even though it's not the end of the binary data.
Another option is binary encoding, a process that changes the binary bytes into ASCII bytes using relatively simple algorithms. The two most popular binary encoding algorithms are *UUencode* and *base64*(base64, 2006) encoding. They are commonly used when binary data needs be stored and transferred over media that are designed to deal with textual data.
However, binary encoding introduces some processing overhead and, moreover, it expands 3 bytes into 4 characters, thus leading to an increase of data size by one third.
In other words, a well recognized and efficient standard for handling binary data in text-based serializations is not available, at least so far, and since this work is aimed to developing a communication framework for LabVIEW based distributed systems, it's worth trying to find a suitable solution among the LabVIEW features.
The natural approach to an efficient serialization of large binary arrays is to flatten the binary data into characters and then handle the result as any other string in XML.
In LabVIEW this data transformation is provided by one of the *flatten to string* functions that convert to string either variants or directly any kind of data type.
LabVIEW *flatten to string* transforms numeric arrays, as well as any other data type, to strings of binary digits in big-endian form. In the case of arrays, the binary sequence of the data is preceded by the record of the size, in elements, of each of the array dimensions.
Obviously, an arbitrary flattened data or data structure can be specified in an XML-RPC communication as the content of a `<String>` element, i.e. its associated `<Val>` container, as long as any special characters such as "<" are represented as entities ("<").
XML provides five pre-declared entities that can be used to escape special characters(xml, 2000) in an XML encoded document. This process is under the responsibility of the server

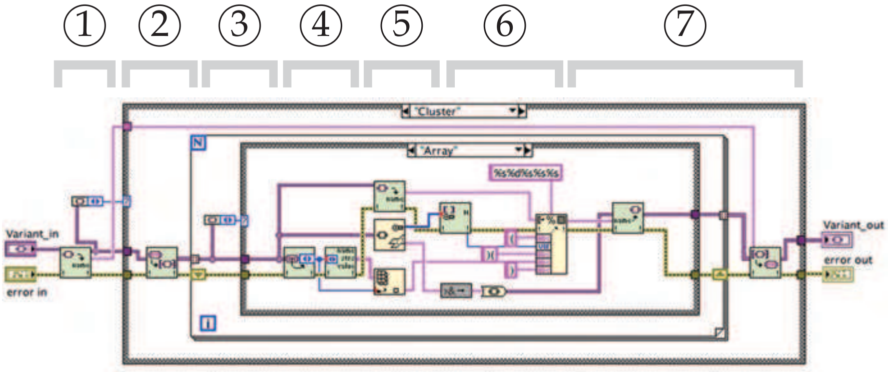


Fig. 7. Pre-processing of LabVIEW data converted to Variant to replace binary arrays with correspondent flattened strings.

side application after encoding the data it has to send, while the client receiving the flattened data will need to check the binary sequence to replace the escaped characters before it process the flattened string to recover the original binary stream.

To preserve compatibility with standard LabVIEW XML functions, instead of introducing modification in the XML coding to process binaries as previously mentioned, it is worth to pre-process the LabVIEW data before it's converted to XML by inserting a VI that inspects the input data structure and replace, when it finds it, any binary array with the equivalent as flattened to string.

Because the pre-processor, similarly to the XML coding tool, must be ready to accept any possible type of data structures as input, the latter is first converted to LabVIEW *Variants*, that sort of type-less container for any (simple or structured) data type that has been introduced in Par.4.1. The *Any-to-Variant* function converts any LabVIEW data to this particular format that can be passed between VIs and manipulated independently from the original data type.

A Variant can be unpacked, its content modified (adding, deleting or replacing data, for instance) and at the end converted back to a "standard" LabVIEW data (numeric, text, array, cluster, etc. or any combination thereof).

The pre-processor developed for XMLvRPC is a VI that recursively searches for nested binary arrays into a LabVIEW data structure, previously converted into a Variant, and replace them with the correspondent flattened strings.

Since the binary array(s) in the Variant structure is(are) flattened and coded into a XML string the reduction in size, with respect to the non pre-processed data, can be significant especially when size of the binary array is large.

In Fig.4 the *XML_preR-processor.vi* is executed just before the *XMLvRPC_WriteResponse.vi* that serialize the LabVIEW data into an XML format and then send the *MethodResponse* to the caller. In Fig.7 a portion of the block diagram of *XML_preR-processor.vi* is shown. The VI inspects the input Variant (1) looking for either a *Cluster* or a *Array* data type. When a *Cluster* is found the VI recursively inspects its inner elements (2). If, at some point, an *Array* is found (3), it's flattened to string (5), checked to escape special characters, and finally replaced to the original *Array* (7) into the *Cluster*.

Later, the XML encoder will handle the string corresponding to the flattened binary array as well as any other string type data, i.e. by encoding the data into a <*String*> element and by

XML-RPC


```
HTTP/1.1 200 OK
Connection: close
Content-length: 450
<?xml version="1.0"?>
<methodResponse>
  <methodName>get_scope_data</methodName>
  <params>
    <Array>
    <Name>chan1</Name>
    <Dimsize>4</Dimsize>
    <DBL>
      <Name>data</Name>
      <Val>1.76365E-1</Val>
    </DBL>
    <DBL>
      <Name>data</Name>
      <Val>2.75606E-1</Val>
    </DBL>
    <DBL>
      <Name>data</Name>
      <Val>7.96079E-1</Val>
    </DBL>
    <DBL>
      <Name>data</Name>
      <Val>1.57441E-1</Val>
    </DBL>
    </Array>
  </params>
</methodResponse>
```


XMLvRPC


```
HTTP/1.1 200 OK
Connection: close
Content-length: 215
<?xml version="1.0"?>
<methodResponse>
  <methodName>get_scope_data</methodName>
  <params>
    <String>
    <Name>chan1(1)(DBL)</Name>
    <Val>¿¿¿¿?Ô58B‡FD?Ït∏,wöE?,˜∫[tæ<?%G&amp;’Y÷û</Val>
    </String>
  </params>
</methodResponse>
```


¿ represents non-printable cahracters

Fig. 8. Binary arrays coding in XML and XMLvRPC.

enclosing the string as it is in *Val* tags (Fig.8)
As consequence, compared to standard XML, the quantity of bytes to be transferred on the network is very much reduced, overhead is almost negligible and the throughput of the communication protocol become compatible with the requirements of a control system.
If considering, for instance, XML coding of a data structure (e.g. a LabVIEW cluster) that includes a 640x480 2D-array of unsigned-bytes, a typical pixels map of a raw image produced by a CCD camera, the reduction in size obtained by applying the pre-processing just described can be a factor 100 or more with respect to standard XML.
It should be mentioned that development of the XMLvRPC's pre and post-processor has been significantly simplified by complementing the LabVIEW functions with the OpenG(Jim Kring, 2003) LabVIEW Data Tools library providing a number of useful functions for manipulating Variants.
Fig.9 presents the execution time in ms for the pre-processing and post-processing VIs as function of the size of the input array. The latter is a 2D-array of unsigned-bytes with equal sizes in both dimensions. In Fig.9 values in abscissa are the (equal) dimensions of the 2D-array. Since pre and post processing are executed separately by the two partners in the communication process (data sender does pre-processing while receiver post-processes data received), their contribution (green and light blue areas) to the total time budget (blue) has been evidenced.
Total execution time, well below 10 ms even for large binary arrays, is comparable to the typical time needed to transfer the same amount of data through the network (from few to several tens of milliseconds).
It must be noted that when a LabVIEW binary array is flattened into a string, some relevant information about the original array is lost. As consequence reconstruction of a binary

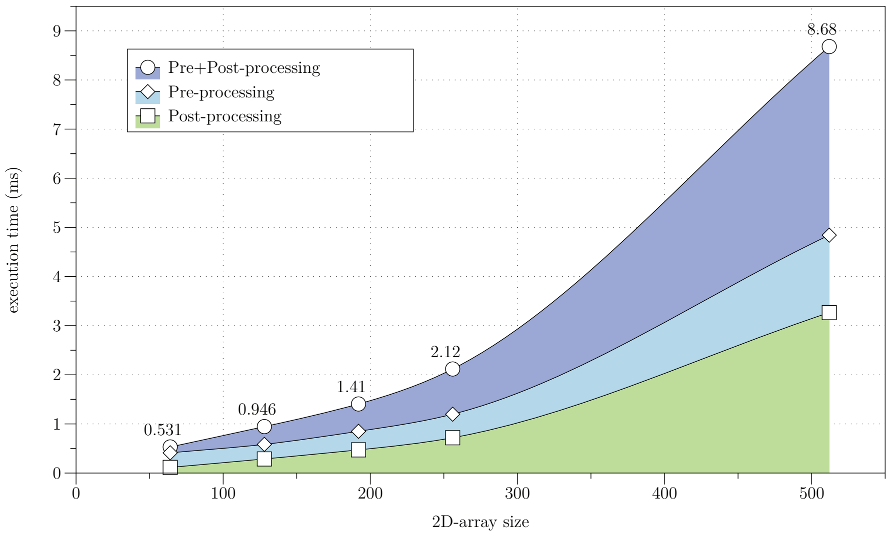


Fig. 9. Perfomances of pre-processing and post-processing routines for different sizes of 2D binary array. Values in abscissa (2D-array size) represent the two equal sizes of a 2D array, e.g. the value 128 corresponds to a 128x128 U8 binary array given as input to processing routines.

array on the receiver (client) side is not possible unless it is supplied, by other means, the dimension(s) of the array and its data type and size. It has been already mentioned that the number of elements for each dimension is included by LabVIEW in a header of the flattened string.

The solution that has been chosen for XMLvRPC serialization is straightforward: the missing information, i.e. the dimensions of the array and its data type (U8, U32, I32 etc.), properly coded and formatted is appended to the name of the variable. This part of the procedure corresponds to step (6) in the block diagram of Fig.7.

As an example, the variable `image`, being the 640x480 2D unsigned-bytes array previously mentioned, after the pre-processing procedure transforming it into a string will change its name into `image(2)(U8)`. On the receiver side a post-processor parses the LabVIEW Variant obtained converting the XML data. It selects the strings that it recognizes, by their particular names, as flattened binary array and un-flatten them into an array having the indicated dimensions (2) and data type (U8).

As alternative additional XML elements can be introduced into `<String>`, e.g. `<ArrayDim>` and `<ArrayDataType>` to specify the original array structure.

Results of some tests have been carried out to evaluate the performance of the communication protocol are shown in Fig.10.

To the overall command execution time shown in the graph contribute, beside the time needed to transfer data-in and data-out from/to client to/from server, the execution time of the *method.vi* on the server side and time needed to open/close communication sockets for the transmission of the `methodCall` and the `methodResponse` between client and server. Performance, especially when dealing with large data sets, can be improved by optimizing the network parameters (e.g. ethernet packet size) as evidenced by the two curves resulting

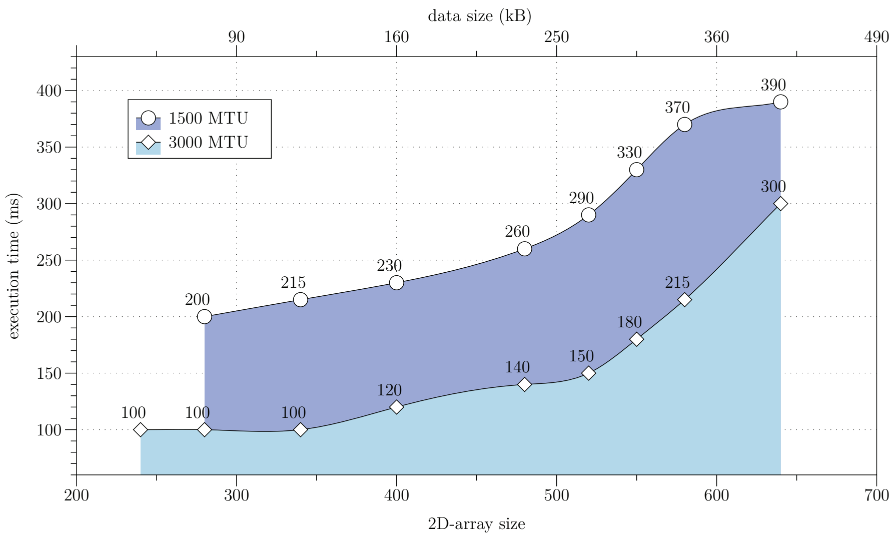


Fig. 10. Time needed to complete a client/server `methodCall` and `methodResponse` as function of different data size returned by the server. The two curves correspond to results obtained with different settings of Maximum Transmission Unit (MTU, i.e. Ethernet maximum packet size) on the network interface of the client computer in a 100 MBps switched network.

from different settings of MTU.

Moreover, when a continuous flow of large data buffers needs to be implemented as, again, in the case of streaming the output of a digital camera, one can consider to implement a dedicated streaming-like communication between server and client(s) that can be initiated/terminated on demand by XMLvRPC commands.

4.2.2 Enhancing the client/server communication

It was mentioned before that the XMLvRPC protocol supports asymmetric `methodCall`/`methodResponse` communications. It means that on the client side the *method.vi* that is required to handle the data received from the server can be different from the one that originated the `methodCall`.

The `methodResponse` can indicate another method (i.e. another client application) to deal with the response on the client side, according to the data produced from the *method.vi* on the server.

The client's `methodRequest`, for instance, might ask for the newest data on the controller, i.e. the most recently updated value among the I/O channels read from equipment assigned to this particular controller. In this case the result will have a data format that cannot be defined a-priori and needs the appropriate client application to be displayed. Another example of asymmetric XMLvRPC communication will be given later in Par.5 when the services registration procedures will be discussed in details.

Interestingly, this feature of XMLvRPC can be employed for extending the client/server communication discussed so far by introducing another option for data transfer between a server and the client application.

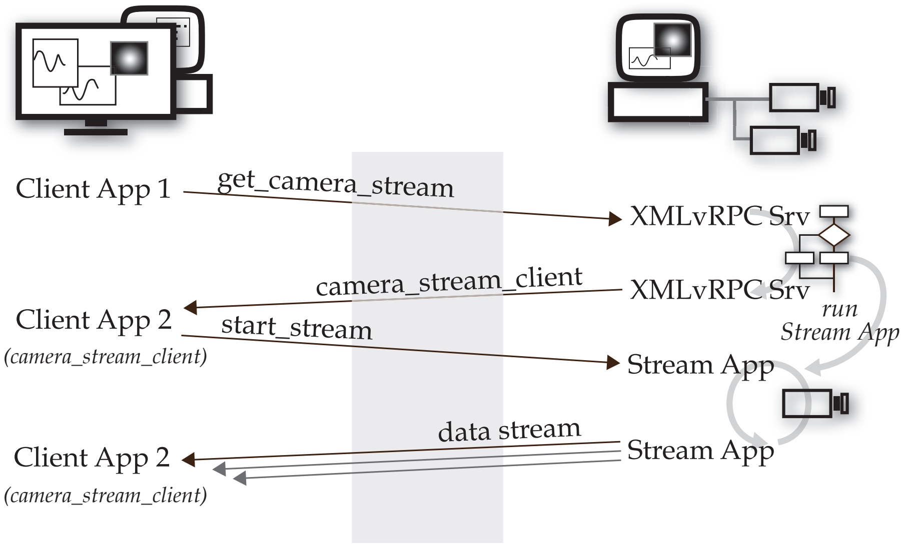


Fig. 11. An example of asymmetric communication in XMLvRPC establishing a streaming of image data from a controller of a digital camera to a client application.

When, for instance, a display or a measurement application is expecting to receive continuous updates of a value for a given time, it would be more efficient to open a socket connection between the two parts and keep it open, as long as needed, instead of forcing the client to continuously send identical *methodRequest*s to the server.
This is even more significant when the data to be transferred each iteration is large. In this case data serialization can be optimized in such a way to reduce the overhead by XML coding/parsing and, as consequence, avoiding the pre/post-processing.
Fig.11 shows an example of asymmetric communication in XMLvRPC aimed to establishing a streaming of image data from a controller of a digital camera to a client application.
As first step the client application sends a `methodCall` to the controller by issuing the method, e.g. `get_camera_stream`, that starts the image stream server. As soon as the stream server is running, the XMLvRPC server replies to the client with `camera_stream_client` as `methodName` providing, as parameters, its IP address and port number for the socket connection, and other optional information. The client, as consequence of the `methodResponse` dynamically opens and runs the display application, i.e. the client-side method, `camera_stream_client`.
Block diagrams of both the client and server side of the data stream connection are shown in Fig.12.
On the server the Stream Application starts listening for incoming requests. When connection is established the inner loop read data from the device controller and, in this particular case, push the 2D array with raw image data to the client.
Since server and client are specialized for handling a particular type of data, i.e. a 2D unsigned-bytes array, serialization and de-serialization are very much simplified compared to what has been previously shown for XML coding. The string sent to the client is obtained by simply appending the 2D array, previously flattened to a string, to the 4-bytes string being

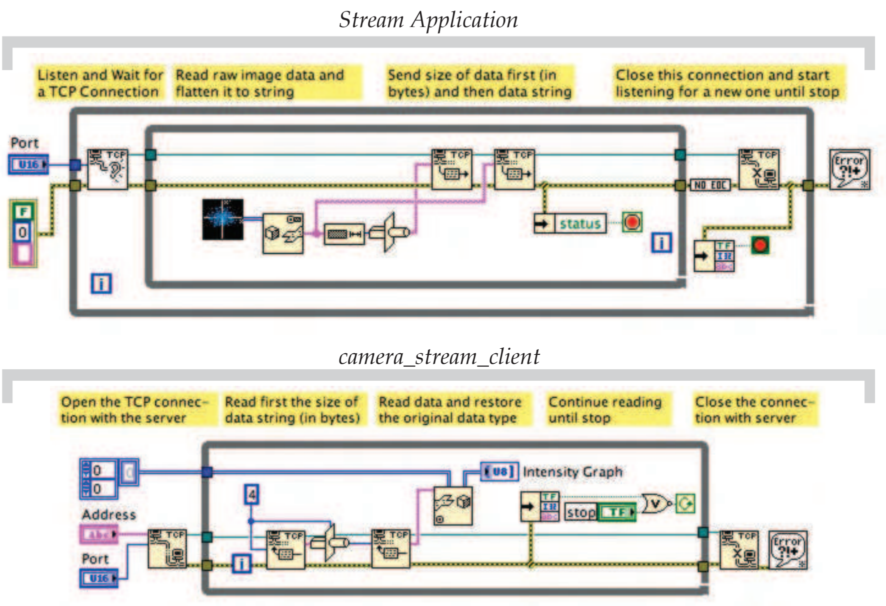


Fig. 12. LabVIEW diagrams of server (top) and client (bottom) applications in a data stream session.

the flattened U32 size of the 2D array.
This information is needed to the client application to inform the *TCP Read.vi* about the size of the buffer it's going to receive from the server.
Finally, a *Type cast* allows to restore the original 2D unsigned-bytes array.
As expected this approach allows better performances than XMLvRPC, around a factor 2 faster, and very simple programming.

5. Inizialization and registration of services

On Par.3 it was mentioned that the final goal of this development should be the realization of a *middle-layer* providing a set of functions allowing (top) client applications communicating with (bottom) hardware equipment via XMLvRPC.
Actually, what has been presented so far already provides a fairly complete solution for small or medium size CS where the number of components to be controlled, and that of controllers and client consoles, is limited. In this case it shouldn't be too difficult to organize a simple list, or a spreadsheet table, with a catalog of components managed by each controller, their I/O channels, IP addresses of network units etc. Then, each client application could relay in this catalog to search for information such as the controller in charge for a particular component and its IP address, the list of methods it provides and optional parameters for a correct formatting of a `methodCall`.
An improvement of the system configuration procedure can be achieved by implementing either a central configuration service or a sort of service location protocol allowing components of the CS to find services and components without prior configuration.

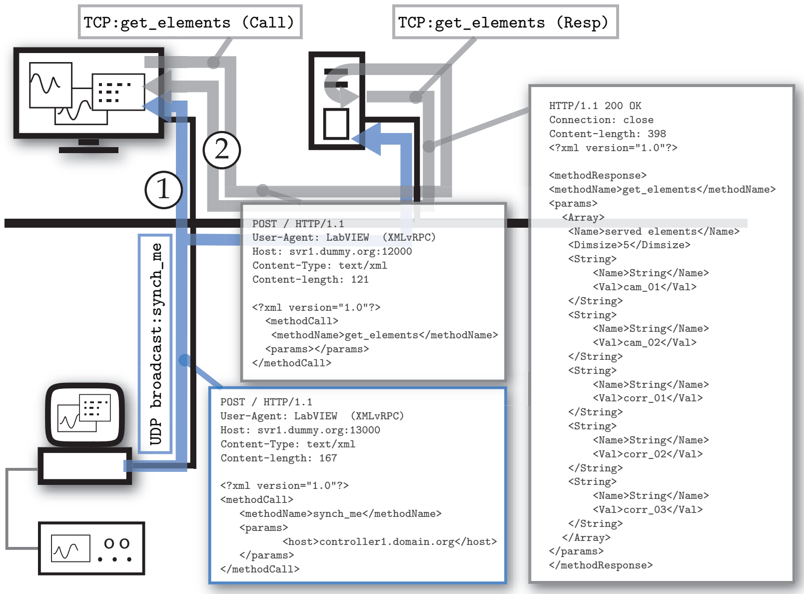


Fig. 13. Synchronization of a controller with the Configuration database either at startup by means of UDP-broadcast (1) or run time by using dedicated XMLvRPC `methodCall` (2).

Fig.13 introduces a new component, the Configuration DataBase, to the DCS depicted so far. Its role is the management of the configuration of components in the distributed control system.

At startup each controller sends a UDP-broadcast to register on the Configuration DataBase by issuing `synch_me` or `register_me` (Fig.13).

The method `register_me` is used if the controller has been configured with all its methods and elements. The method `synch_me` is used if some methods (and elements) are provided by the Configuration DataBase. If the system has more than one Configuration DataBase, for redundancy purposes, both will receive the request to register the controller in the system. The Configuration DataBase detects the UDP-broadcast and then sends to the controller a TCP/IP `get_elements` `methodCall` and then a `get_element_conf` for each element listed in the previous methodResponse received from the controller.

Practically, local services (i.e. those specific for a class of elements) are configured directly on each controller while global services (e.g. back-up, restore etc.) can be configured centrally in the Configuration Database.

Consoles and high level applications rely on the Configuration DataBase to locate the controller in charge for a particular element. They use an UDP broadcast to find the Configuration DataBase, i.e its IP address. At this point the client can either decide to receive the complete configuration of the system at once and refresh it periodically or inquire that service each time an application needs to identify the controller in charge of a particular I/O channel or service.

The Configuration DataBase can run either on a dedicated server (as shown in the picture) or any client, or controller, of the the DCS.

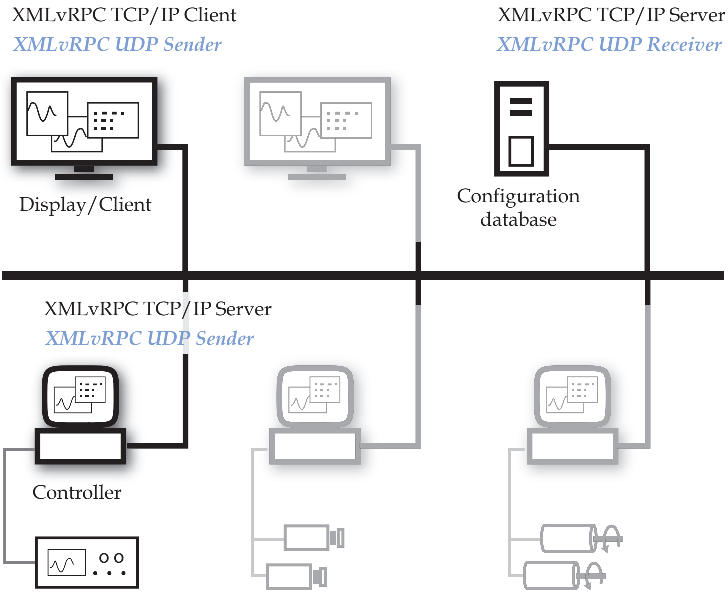


Fig. 14. Typical components in a XMLvRPC based distributed control system. Services running on each component are also shown.

6. Components in a XMLvRPC distributed control system

Fig.14 shows an example of components in a XMLvRPC distributed control system. Controllers run front-end applications: they are either the interface to equipment or provide general services.

Displays, or consoles, run user applications or analysis and measurement procedures. They directly connect to controllers to run remote procedure provided they know (the IP address of) the controller in charge for the particular I/O channel (or service) and the methods made available for it.

This information is provided by the Configuration Database on request from the Console (or any another client). The Configuration Database is thus the repository of the system configuration files collected from any controller at the time they start-up and register to the system.

To summarize, TCP/IP and UDP services in XMLvRPC are the following:

XMLvRPC TCP/IP Server: runs on each controller and on the Configuration DataBase serving XMLvRPC `methodCalls` issued by clients. For each controller, valid `method Names` correspond to VIs listed in the *XMLvRPC_ClientServer/methods_svr* directory. Elements under control are listed in *XMLvRPC_ClientServer/elements_svr* directory.

XMLvRPC TCP/IP Client: runs on each console (a client, in general); it sends XMLvRPC `methodCall` to XMLvRPC TCP/IP Server as consequence of some action on the console panels or from measurement application.

Fig. 15. Directories, VIs and configuration files in the XMLvRPC package

XMLvRPC UDP Receiver (Configuration DataBase): runs on the Configuration DataBase to serve `synch_me` or `register_me` `methodCalls` sent by controllers or `locate_cdb` sent by consoles at startup.

XMLvRPC UDP Sender: runs on controllers and consoles at startup. It sends `synch_me` or `register_me` `methodCalls` to Configuration DataBase for registering the new controller in the system. Consoles use it to locate the Configuration DataBase.

Configuration DataBase: is the repository of the configuration; it supplies clients (e.g. display/measurements applications) with information about the controller in charge for a given element.

6.1 The XMLvRPC suite of VIs

Fig.15 shows the structure (directories, VIs and configuration files) of the XMLvRPC software package.

The installation can be identical for any component of the XMLvRPC DCS because the role assigned to each component (i.e. server or client) and the services it will provide are configured by *XMLvRPC_startup.vi* according to the settings in the configuration file *Config_local.xml.*

If the local computer runs a Configuration Database the methods to be used for this service are

PHP script

```php
<?php
include("xml-rpc.class.php");
function get_tag($string, $tag){
    $tagstart = "<".$tag.">";
    $tagend = "</".$tag.">";
    $string = " ". $string;
    $ini = strpos($string,$tagstart);
    if ($ini == 0) return "";
    $ini += strlen($tagstart);
    $len = strpos($string, $tagend, $ini) - $ini;
    echo $tag.": ".substr($string, $ini, $len)."</br>";
    return substr($string, $ini + $len);
}
$thisHost = "client1.dummy.org";    //client hostname
$thisPort = 12000;                  //client port
$server = "svr1.dummy.org"          //remote server hostname
$serverPort = 12000;                //remote server port
$socket = Socket::singleton();      //create socket
$socket->connect($server, $serverPort); //connect to remote server
//template for methods without params, e.g. get_elements, get_methods, get_timestamp
$str1 = "POST / HTTP/1.1\nUser-Agent: LabVIEW  (XMLvRPC)\nHost: "     ;
$str2 = "\nContent-Type: text/xml\nContent-length: ";
$str3 = "<?xml version=\"1.0\"?>\n<methodCall>\n<methodName>";
$str4 = "</methodName>\n<params><name>none</name>\n</params>\n</methodCall>";
$methodName = "get_elements";                   //set method's name
$length = strlen($str3.$methodName.$str4);  //length of payload string
//format methodCall
$methodCall = $str1.$thisHost.":".$thisPort.$str2.$length."\n".$str3.$methodName.$str4;
//print methodCall
echo "<b>methodCall sent to server:</b></br>".
 str_replace("\n", "</br>", htmlspecialchars($methodCall)."</br>"."</br>");
//send method and get response
$socket->sendCmd($methodCall);
$response = $socket->getMultilinedResponse();
echo "<b>methodResponse from server:</b></br>";  //print methodResponse's name and Values
$resp = get_tag($response, "methodName");
while (strlen($resp)) $resp = get_tag($resp, "Val");
?>
```

HTML output

Fig. 16. PHP script (left) for issuing an XMLvRPC `methodCall` from a web browser and the HTML output of the values in the `methodResponse` (rigth).

in *methods_cdb_rec*. The *system_database* directory contains a directory with the configuration files of the controlled elements for each of the controllers (servers) which registered to the system.
Adding an element, for instance a new device or service, is as simple as creating, and properly editing, the corresponding configuration file in the controller directory of the *system_database* folder. Similarly, a method can be added to a server, or a client, by including the correspondent VI in the *methods_svr*, or *methods_clt*, directory respectively. In both cases there is no need to modify either source code or global configuration file since these directories are scanned at start-up to identify services and elements available to this component of the control system.
Development of a VI for a new method can start from a common template since they all present an identical interface (i.e. the connector pane) to the calling application.

7. Interoperability

The communication framework described so far, although it has been designed and optimized for a LabVIEW-based DCS, exhibits a clear attitude to interoperate with network applications developed by using different programming language and/or running on diverse hardware platforms or operating systems.
First of all it is, essentially, a fairly customized version of the XML-RPC protocol, yet compatible with all its implementations at least for what concerns the client/server communication and the basic structure of the body of the request.
That means any XML-RPC compatible client can issue a well-formed `methodCall` to an XMLvRPC server and receive a `methodResponse` that, afterwards, it will be able to parse to properly extract the XML elements.
Handling of binary arrays that have been serialized as described in Par. 4.2 will be under responsibility of the application that required the data. Even if the latter wouldn't be equipped with tools for restoring the original format of the serialized binary array, these data will be still

recognized as a valid string element, though meaningless.
A lower, hence more general, level of compatibility is the network socket communication that is at the basis of the XMLvRPC protocol.
Libraries for socket communication are available for, practically, any existing programming language and it very easy, as it was with LabVIEW, writing a piece of software for implementing a socket communication session.
An example is shown in Fig.16 allowing to request and display data received from an XMLvRPC server on a web browser.
The simple PHP script on the main frame is used for composing and issuing a `methodCall`, to read the reply of the XMLvRPC server and finally parse the `methodResponse` for printing the main information such as the value elements in the `<params>`.

8. Conclusion

The communication framework presented in this paper has been described in details to provide a ready to use solution for implementing a distributed control system with LabVIEW. Nevertheless, it could also be seen as a collection of strategies that instead of being adopted as a whole may be individually replaced by, or integrated with others if those are found to be best suited for some particular application or requirement.
It was mentioned, for instance, that XML could be exchanged with other rules for formatting data and also that binary array serialization can be based, as alternative, on the referred standard encoding algorithms.
All the development strategies that have been presented share in the intention to exploit and take advantage from the great features of LabVIEW for delivering an overall solution that still offers the expected compatibility with other programming languages and with other well-established communication solutions.

9. References

COM: Component Object Model Technologies *Microsoft Corporation*, http://www.microsoft .com/com/default.mspx

Catalog Of OMG CORBA/IIOP Specifications *Object Management Group, Inc.*, http://www. omg.org/spec/CORBAe/1.0/

Jini Architecture Specification *jini.org*, http://www.jini.org/wiki/Jini_Architecture_Specifica tion

T.Bray, et.al. (2008). Extensible Markup Language (XML) 1.0 (Fifth Edition) *W3 Consortium*, http://www.w3.org/TR/REC-xml

SOAP Version 1.2, *World Wide Web Consortium (W3C)*, http://www.w3.org/ TR/soap12-part1/

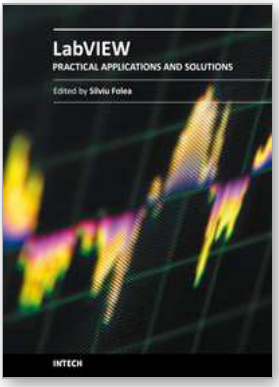
XML-RPC Specification, *Scripting News, Inc.*, http://www.xmlrpc.com/spec

Remote Procedure Call Protocol Specification v.2, *Network Working Group, Sun Microsystems, Inc.*, http://tools.ietf.org/html/rfc1057

Catani, L. (2008). An XML-based communication protocol for accelerator distributed controls, *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, Volume 586, Issue 3, 1 March 2008, Pages 444-451.

Lima et al., (2004). Choosing among LabVIEW Communication Features. *LabVIEW 2010 Help Manual*, Part Number: 371361G-01, June 2010

Jim Kring, (2003). OpenG LabVIEW Data Tools. *OpenG.org website*, http://wiki.openg.org/Oglib_lvdata
Crockford, D. (2006). JSON format specifications. *Internet Engineering Task Force*, http://www.ietf.org/rfc/rfc4627.txt?number=4627
JSON-RPC Working Group, (2009). JSON-RPC 2.0 Specification proposal. *JSON-RPC Google Group*, http://groups.google.com/group/json-rpc/web/json-rpc-1-2-proposal
Josefsson, S. (2006). The Base16, Base32, and Base64 Data Encodings. *Network Working Group*, http://tools.ietf.org/html/rfc4648



Practical Applications and Solutions Using LabVIEW™ Software

Edited by Dr. Silviu Folea

ISBN 978-953-307-650-8
Hard cover, 472 pages
Publisher InTech
Published online 01, August, 2011
Published in print edition August, 2011

The book consists of 21 chapters which present interesting applications implemented using the LabVIEW environment, belonging to several distinct fields such as engineering, fault diagnosis, medicine, remote access laboratory, internet communications, chemistry, physics, etc. The virtual instruments designed and implemented in LabVIEW provide the advantages of being more intuitive, of reducing the implementation time and of being portable. The audience for this book includes PhD students, researchers, engineers and professionals who are interested in finding out new tools developed using LabVIEW. Some chapters present interesting ideas and very detailed solutions which offer the immediate possibility of making fast innovations and of generating better products for the market. The effort made by all the scientists who contributed to editing this book was significant and as a result new and viable applications were presented.

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821