Web-Enabled Remote Scientific Environments

A new approach to developing Web-enabled environments for remote diagnostics, maintenance, and experimentation in engineering is based on a middleware layer that uses a Java-Internet-Labview server to provide communication between Java programs and Labview virtual instruments. The authors illustrate their technique by applying it to the development of a complete Web-enabled application for remote control of a thermal process.

Web-enabled technologies and applications for control engineering are important research topics for many institutions and companies. This interest in exploring new methods for remote experimentation, diagnostics, and maintenance in engineering stems from two fundamental breakthroughs: the introduction of Internet-enabled supervision mechanisms for industrial and didactical processes to offer competitive access to distant resources,[1–3] and innovative new technologies in curricula and renovated training programs.[4,5]

Web-based control environment deployment benefits both end users and developers (process control engineers, teaching staff, and so on). However, developers face extra work when transforming an existing local system into a Web-based environment—they know how to manage hardware and software in a local control system, but new problems arise when making the system accessible via the Internet. In particular, developers must create interactive GUIs for the system that can be deployed via the Internet in the form of a Java applet—the simplest way to integrate interactive user interfaces for remote supervision into Web-based learning-management systems.

Labview's virtual instrument (VI) is a graphical programming language specifically designed for developing instrumentation, diagnostics, and data acquisition systems. Many engineering and scientific disciplines, both professional and academic, have adopted Labview, which has resulted in a broad collection of libraries and legacy code, most of them working in local control systems. Publishing a Labview VI on the Internet is a long-standing, click-and-share feature of this software.[6] However, a simple mechanism isn't yet in place to make the VI variables (controls and indicators) accessible from Java applets. This requirement poses an important setback for developers who want to transform existing Labview-based local control systems into Web-based ones.

In this article, we present a new approach for quick and simple creation of Web-enabled control environments that use Labview on the local side, Java applets on the remote client side, and TCP/IP

1521-9615/09/$25.00 © 2009 IEEE
Copublished by the IEEE CS and the AIP

Héctor Vargas, José Sánchez-Moreno, and Sebastián Dormido
Spanish National University for Distance Education
Christophe Salzmann and Denis Gillet
Ecole Polytechnique Fédérale de Lausanne
Francisco Esquembre
University of Murcia

as the communication mechanism between both elements. We've developed two software components to solve this problem: a stand-alone application, called a JIL (Java-Internet-Labview) server, which acts as middleware to publish an existing Labview VI on the Internet; and a Java library file, which Java clients can use to control and access JIL-published VI variables. The novelty of this approach is that programmers can access the VI's controls and indicators from the Java program in a fully transparent way to the Labview developer—that is, without introducing any modification to the original VI.

Communication via the Java-Internet-Labview (JIL) Server

Labview's built-in local communication facilities let it include communication facilities at design time with almost any existing software and hardware. However, once a Labview VI is developed, implementing a bidirectional data exchange with a Java applet requires editing the existing VI wiring diagram to include new TCP/IP communication blocks into the diagram. It also requires implementing the TCP/IP-based communication library calls in the Java client. This procedure is time-consuming and demands both Java knowledge and an understanding of how TCP/IP communication works.

The JIL Server

Web-enabled environments for remote diagnostics, control, or experimentation are commonly based on client-server architectures that use TCP/IP links to exchange data and commands between both sides (see Figure 1).

The command parser, sender, and control loop are on the server side. The command parser receives commands from the client, interprets them, and executes the requested actions. When no request is received, the parser just sleeps, leaving the processor free for other duties. Similarly, the sender sends to the client application the measurements the control loop acquires when a command requires them.

Figure 1 shows that we can draw a separation line between the two communication tasks and the control loop, which is the task connected to an industrial process or didactical setup. In this scenario, the control loop is a Labview VI that supervises, diagnoses, or controls the industrial process or the didactical setup, principally without TCP/IP communication capabilities. The JIL server acts as middleware to provide a TCP/IP wrapping to the control loop, acting as both the command parser and sender blocks, effectively communicating the control loop VI with a remote Java client.

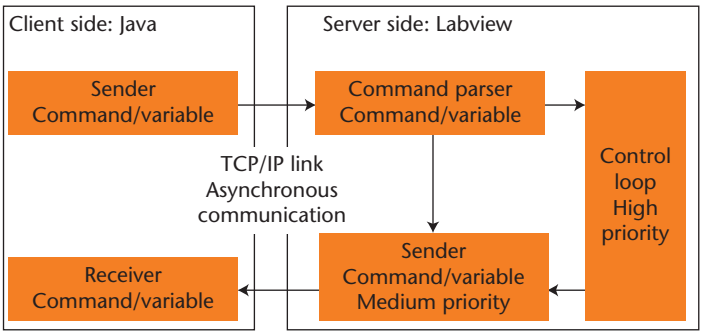

Figure 1. Command-based architecture. This design pattern known as the command-based architecture is the basis of the Java-Internet-Labview (JIL) approach.

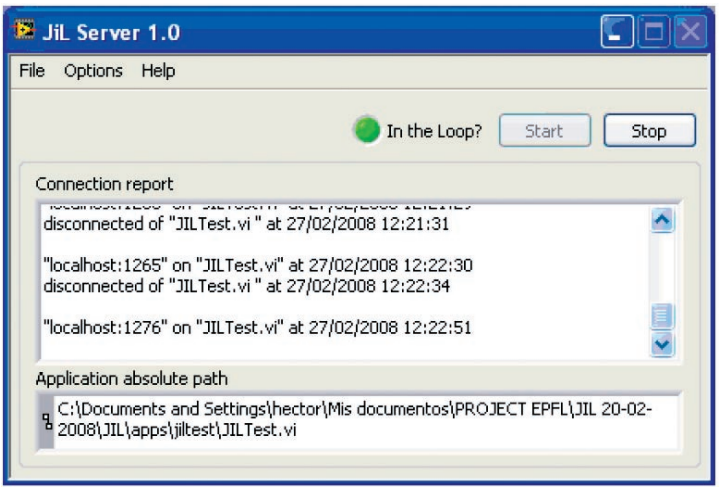

Figure 2. The Java-Internet-Labview server's front panel. Via the Options drop-down menu, it's possible to configure the server's TCP listening port and the rate at which the JIL server will send data packets to clients, change the size of the data packets, have the JIL server open the virtual instrument target's front panel, and so on. At the bottom of the GUI, a connection log is displayed. Users can save this report in a file and send it to the server's administrator.

To publish a VI using the JIL server, the author uses the server's control panel to select the local VI he or she wants to publish and presses the Start button to finish the publishing task (see Figure 2). Every control and indicator of the VI becomes immediately accessible to any applet that uses the provided Java library file, described later. The server performs an automatic scan of all the VI controls and indicators, initializes the network input port, and waits for an incoming connection from a Java applet. When the connection is estab-

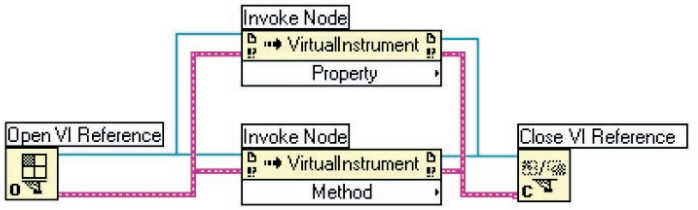

Figure 3. Property modification and method calling. Invoke-node blocks can modify a property or call a method in a virtual instrument.

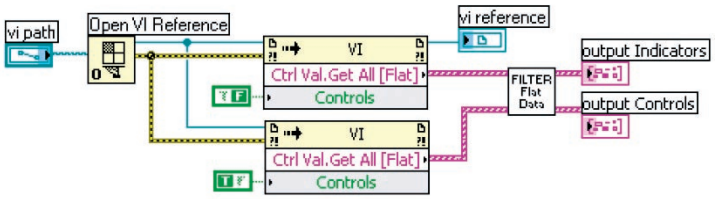

Figure 4. A portion of the Java-Internet-Labview (JIL) server wiring diagram. This part helps obtain the list of the virtual instrument's controls and indicators by using two invoke-nodes blocks.

lished, the server then listens for incoming commands, which it parses and serves as requested.

JIL Server Implementation

We have developed the JIL server entirely in Labview, making use of the VI server feature that it provides. The VI server is a collection of blocks that allows programmatic access to the VI's objects and functionalities. Any VI exposes properties—characteristics you can read, write, or both, depending on the property—and methods—actions you can perform on a VI. Authors can access these properties and invoke the methods using blocks. An example of a VI class property is `Execution:State`, which indicates a VI's execution state (bad, idle, or running). An example of method is the `run-VI` method that programmatically runs a VI, much like manually pressing the run button in the VI's user interface. Figure 3 presents a simple template, in which we've used two Labview *invoke-node blocks* of the VI server feature to execute methods and set or get property values.

Figure 4 shows a portion of the JIL server's structure developed with the invoke-node blocks. When a developer publishes a Labview VI via the JIL server, the server's first action is a programmatic reading of all VI control and indicator names. It does this by calling the `control-value-get-all` method in two invoke-node blocks—one for getting the indicators' names and the other for the controls' names. Once the JIL server receives this information, it waits for an incoming connection from a Java applet. Once the connection is established, the server continues to listen for incoming commands from the applet.

Anytime the JIL server receives a command, it's parsed and processed according to its type. If the command is a request for the available controls and indicators, the JIL server returns the name list (because it was extracted when the two invoke blocks obtained it). When the user issues a control command, he or she can start, stop, or reinitialize the VI via invoke nodes with `run-VI`, `abort-VI`, or `reinitialize-all-to-default` methods. The server can receive a state command, which is either for reading an indicator or for writing a control. Either way, the corresponding method performs this action in an invoke-node block. The rest of the time, when information isn't received, the JIL server waits.

JIL-Enabled Java Clients

A JIL-enabled Java client must implement two (sender and receiver) communication tasks to interact with a published VI. The sender task sends control commands (to set VI control values) or state commands (to modify the VI's state) whenever the user interacts with the client's user interface. The receiver task periodically reads the VI indicator values that the JIL server sends—for example, in a Web-enabled control application, values that the Java client sends can be controller parameters, and the data stream it receives back will contain data relative to the process state (the tank level, the temperature in a heat exchanger, a motor's angular position, and so on.) The `JIL.class` utility—part of the JIL package—provides methods that make it easy to implement these communication tasks in a Java program. Table 1 lists the methods that the JIL class provides to open a connection, control the VI, and obtain information about its controls and indicators. Table 2 enumerates the methods to read and write the indicators' values and VI's controls published by the JIL server.

The first method to invoke on a JIL object is the `connect()` method, which establishes the connection with the server and reads the remote VI's list of controls and indicators and stores them internally in the object. This internal list acts as a buffer to improve the communication performance, meaning that reading the value of one or more VI indicators requires a call to the `refresh()` method prior to calling the appropriate `getValue()` methods. The `refresh()` method

Table 1. Java-Internet-Labview class methods used to exchange information with the virtual instrument.

Method	Action
`JIL(String hostname, int port)`	Constructor used to create a JIL object
`void connect()`	Creates and initializes the TCP/IP connection with the JIL server; a `java.io.IOException` is thrown if an I/O error occurs; this method must be called prior to further operation
`void disconnect()`	Closes the TCP link with the JIL server; a `java.io.IOException` is thrown if an I/O error occurs
`String[] getIndicators()`	Returns an array containing the VI's indicator names; indicators work like read-only variables
`String[] getControls()`	Returns an array containing the VI's control names; controls work like write-only variables
`void run()`	Starts the VI; a `java.io.IOException` is thrown if an I/O error occurs
`void stop()`	Stops the VI; a `java.io.IOException` is thrown if an I/O error occurs
`boolean isRunning()`	Returns a Boolean indicating whether the VI is running; a `java.io.IOException` is thrown if an I/O error occurs

Table 2. Java-Internet-Labview class methods to set the indicator values and read the control values.

Method	Action
`void setValue(String name, Object value)`	Sets the VI control value with the given name; the value must be an object that wraps the desired primitive value; if the primitive value type doesn't correspond to the Labview control type or the name doesn't correspond to a control, a `java.lang.IllegalArgumentException` is thrown
`Object getValue(String name)`	Returns an object that wraps the primitive value (as defined in the VI) of the indicator `name`; if the name doesn't correspond to an indicator, a `java.lang.IllegalArgumentException` is thrown
`void refresh()`	Reads and saves all the VI indicator values into a private class list at once; a `java.io.IOException` is thrown if an I/O error occurs
`void flush()`	Writes the private class list with all the controls' values to the VI at once; a `java.io.IOException` is thrown if an I/O error occurs

reads all the values from the JIL server into the private list, from which the `getValue()` method retrieves any indicator's value. Similarly, to modify the value of one or more VI controls, the programmer must first call the appropriate `setValue()` methods to set the new values in the private list. When all values have been set, a single call to the `flush()` method transmits the new values to the JIL server. The JIL class implements the minimal features that communicate with the JIL server. For example, Java programmers can easily extend this class using Java inheritance—without changes to the existing architecture—to implement additional features such as advanced communication management or alarm management.

A Simple Example

Figure 5 shows a complete, albeit rather naïve, example of the Java library. The diagram consists of a synchronous `while` loop with four controls connected to four indicators. This has the simple (and rather useless) effect of showing any control modification in the corresponding indicator.

Once the VI is published using the JIL server, the Java code in Figure 6 communicates with this VI—an example of this is as follows:

The first lines of this applet's `init()` method instantiate a JIL object and try to connect the object to the VI that the JIL server published in the *example.uned.es* host at the 8080 port. If the connection is successfully established, the JIL server then starts the VI execution. The next `try/catch` block sets the values of the four controls and flushes the buffer, and the third block refreshes the input buffer and reads and prints the VI indicator values. The VI must have placed the previously sent control values in the indicators, according to the VI wiring diagram. The final `try/catch` block stops the VI and closes the connection.

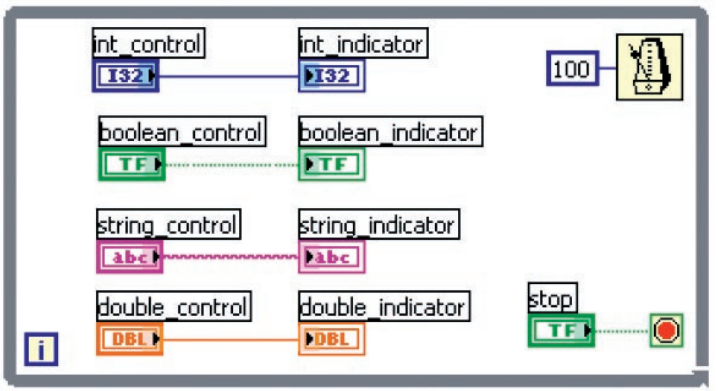

Figure 5. Wiring diagram of the `JiLTest.vi`. The diagram corresponds to a `while` loop running ad infinitum until the user pushes the Stop button. Inside the loop, four controls are directly wired to four indicators of similar data types. Every 100-msec cycle, the controls' values are transmitted to the indicators, which visualize any control modification done by the user.

Integrating the JIL Approach

At this point, the tools to create the server-side components are clear: Labview for developing the VI (the control loop task) and the JIL server for providing the TCP/IP wrapping to the VIs. However, for the client side, we must provide a tool to help nonexpert Java developers produce JIL-flavored applets. This tool should conceal the implementation details of the JIL approach and provide advanced interactive user interfaces. To simplify the programming task for nonexpert Java developers, we've augmented the fast-prototyping program Easy Java Simulations (EJS) with the capability to create JIL-enabled applets using the JIL library in a transparent way.

EJS is a freeware, open source tool developed in Java and specifically designed to create interactive dynamic simulations.[7–9] Although we originally designed EJS for developing interactive simulations in physics, we've recently augmented it to help create Web-accessible laboratories in control engineering education. For this reason, recent releases of EJS support connections with external applications, such as Matlab/Simulink, SciLab, and SysQuake. EJS now provides point-and-click mechanisms to connect Java variables with Labview controls and indicators; it also hides JIL class implementation details from programmers. For more information, visit www.um.es/fem/EjsWiki.

From a practical viewpoint, developers can create advanced interactive applets using EJS by working in two main sections: the model and the view (see Figure 7). The model section must provide a mathematical description of the industrial process, didactical setup, or the physical phenomenon being studied. The model includes a list of its state, parameters, input, and output variables together with their initial values and equations that state how these variables relate to each other, evolve with time, or change under user interaction. The view must provide a graphical representation of the program output and an interface for user interaction. EJS provides a simplified program structure, custom model tools (such as an advanced differential equation editor), and drag-and-drop view elements that let developers work at a high level of abstraction, thus speeding up the creation process. Developers input the qualified information on the simulation that only a human can provide—such as math equations, the initial model state, and the graphical interface's design—and the program takes care of all the computer-related aspects of creating a finished, independent Java applet or application.

When using EJS to create the remote interface for an existing VI, the actual equipment determines the process's behavior, thus programmers don't need to specify the model. However, they must declare the variables to help the EJS communicate with the VI controls and indicators. We've edited the panel for variable declaration in EJS so that the developers can enter the VI's URL published by JIL. EJS then automatically connects to the VI and retrieves the list of controls and indicators and offers it to developers so that he or she can link them to the variables declared in the EJS's variables table.

Figure 8 shows the actions required to connect the controls and indicators with an applet created in EJS. The first step is to fill out the text field, *external file*, with the JIL server and VI's locations. The syntax is

```
<labview:IP_address:port>
   VI_relative_path
```

The `labview` keyword identifies the file as a JIL-enabled remote VI, the `IP_address` and `port` values indicate the URL at which the JIL server is listening, and the `VI_relative_path` value indicates the VI's location in the computer on which the JIL server runs. Once the developer sets the data source, the server sends EJS the target VI's input and output variables—that is, the names of all controls and indicators. Next, the developer must declare the local variables and connect them with their remote data source. Figure

```
public class SimpleJILTest extends javax.swing.JApplet {

public void init() { // start connection and run the VI
jil.JiL vi = new jil.JiL("example.uned.es",8080);
try {
vi.connect();
vi.run();
} catch (Exception e) {
System.out.println("Error when connecting to the VI");
}

try { // setting the value of the VI controls
vi.setValue("boolean_control", new Boolean(true));
vi.setValue("int_control", new Integer(1));
vi.setValue("double_control", new Double(0.5));
vi.setValue("string_control", "Hello world");
vi.flush();
} catch (Exception e) {
System.out.println("Error when setting VI controls");
}
try { // reading the value of the VI indicators
vi.refresh();
System.out.println("boolean = "+ vi.getValue("boolean_indicator"));
System.out.println("int = "+ vi.getValue("int_indicator"));
System.out.println("double = "+ vi.getValue("double_indicator"));
System.out.println("String = "+ vi.getValue("string_indicator"));
} catch (Exception e) {
System.out.println("Error when readingVI controls");
}
try { // stop the VI and close the connection
vi.stop();
vi.disconnect();
} catch (Exception e) {
System.out.println("Error when closing the connection");
}
} // end of the init method
} // end of class
```

Figure 6. Java code example. Even though all the code is located in just one method, to facilitate understanding, we can distinguish four sectors corresponding to the four `try/catch` blocks: opening connection, write control values, read indicators, and closing connection.

8 shows the model section's variables page, where the local variables must be declared in EJS. In this page, the developer must specify the names of the variables, initial values, types, dimensions, and whether they're connected to some VI target control or indicator. To ease the connection process, anytime a user clicks on a cell in the "connected to" column, EJS pops up a dialog box (see Figure 8b) that contains the controls and indicators available in the VI target. To establish the link between the VI controls, the indicators, and the EJS variables, the developer just clicks on a control or indicator name. From this point, the connected EJS variables act as normal Java variables, taking into account that

- if a variable is connected to a control, it must be considered as a write-only variable (it allows putting a new value to the linked control), and
- if it's connected to an indicator, it must be seen as a read-only variable (it allows getting the current value of the associated indicator).

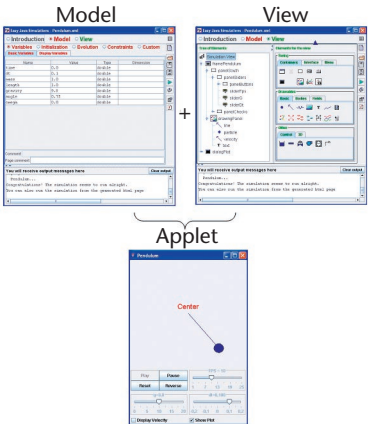

Figure 7. Creating an applet in Easy Java Simulations. The developer first describes the phenomenon to simulate by its state variables (Variable pages), its mathematical model (Evolution page) and the limitations in its possible states (Constraints page); second, the user interface is developed with the graphical library's elements to visually represent the phenomenon under study. Once this information is established, the applet is generated.

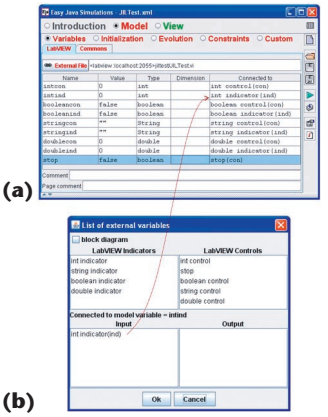

Figure 8. Easy Java Simulations (EJS). (a) The EJS Variables page, with the local variables to connect with the VI specified in the text field external file. (b) A dialog box, with the VI controls and indicators that can be connected to the EJS local variables.

The VI specified in the external file text field (see Figure 8a) corresponds to the example shown in Figure 5, the Variables page (see Figure 8b) contains the same number of indicator variables but adds a "stop" control to the number of control variables. For example, if four indicator controls are connected to four indicators, you would have a total of nine variables. By clicking on any variable's "connected to" cell, the "list of external variables" dialog box pops up to let you select the desired external variable (see Figure 8b).

Establishing this simple connection between model variables, VI controls, and indicators instructs EJS to include the necessary Java calls to the library `JiL.class` in the generated simulation. The variables' values are then passed back and forth as the program requires. Authors can now use the EJS model variables when constructing their own view, either for visualization or interaction purposes, just as in any other EJS-generated view, and obtain direct access to the VI controls and indicators. The result is a fully functional Java applet with the particularity that its variables have their data source or target in a remote VI.

Thermal Processes

Using our approach, we developed a Web-enabled environment as part of the current infrastructure that the Department of Computer Science and Automatic Control at the Spanish National University for Distance Education (UNED) provides to students for remote experimentation. The purpose of this Web-enabled environment is to be able to work either with a thermal process's computer model or with the actual equipment via the Internet. In this example, the steps to create the Web-enabled environment are to

- use EJS to design and implement a Java applet with a thermal process computer model and an appropriate user interface,
- create a Labview VI to control the actual thermal process laboratory equipment,
- publish the VI using the JIL server, and
- modify the applet to allow connection with the JIL-published VI.

In the rest of this section, we describe the subsequent development processes to create a complete Web-based environment. For the sake of brevity, we've excluded some of the finer details; however, we left enough information for readers to get a clear idea of how to create such an environment by using our approaches.

Computer Model

Quanser Consulting designed the heat-flow system (see Figure 9) we chose for the laboratory. This system consists of a duct with a heating element, a blower, and three sensors—S1, S2, and S3—along the duct. Users can control the power delivered to the heater and the fan speed using

analog signals V_h and V_b, and they can measure the fan's speed using a tachometer that produces the analog signal V_t.

We've used system identification techniques to derive the model required to simulate the system with EJS. The model has the following form:

$$G(s) = \frac{T_n(s)}{V_b(s)} = \frac{K_p(1+\tau_3)e^{-\tau_d s}}{(1+\tau_1 s)(1+\tau_2 s)},$$

where the gain K_p (degrees C/volt), the lags, and the delays—both in seconds—depend on which of the three sensors the user selects for closing the temperature control loop. We use the built-in differential equation editor to implement this model in EJS.

As mentioned earlier, we created a view in EJS using a collection of ready-to-use view elements as building blocks to construct the graphical interface. These view elements encapsulate lower-level graphical elements programmed in Java Swing classes and two- and three-dimensional widgets of the Open Source Physics project.[10] Figure 10 shows the system's graphical view.

To facilitate visualization of the heat-flow dynamics, the interface's left panel displays a 3D representation of the apparatus in which the inner air's color changes according to its temperature. We consider this feature to be a kind of *augmented reality* technique that can give the user the feeling of physical presence in the lab. The bottom-left part of this panel shows several tabs that let users modify different experimentation parameters.

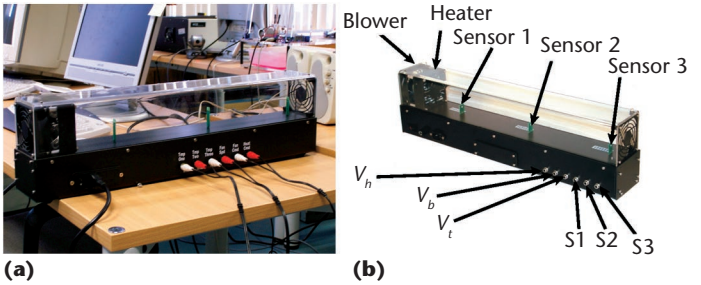

Figure 9. Heat-flow system. (a) Installation in our control laboratory and (b) a scheme showing the distribution of the three temperature sensors along the duct. Depending on the temperature sensor used to close the control loop, we obtain different process dynamics that let instructors offer students different versions of the same experiment.

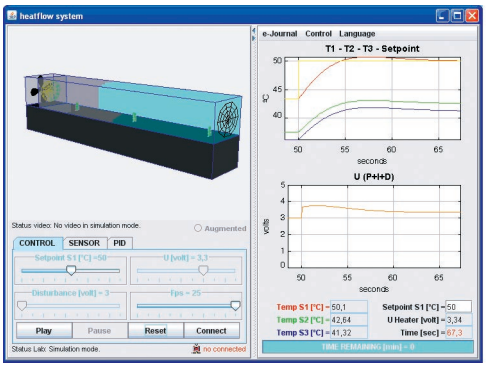

Figure 10. A heat-flow system GUI. The computer model implemented in Easy Java Simulations (EJS) produces the displayed data. The heat flow's color changes according to the inner air's temperature.

Labview VI's Design

Figure 11 shows (for illustration purposes only) Labview VI's wiring diagram, called `heatflow.vi`, which we created for the Web-enabled environment's server side. When creating the local control application, it's very important to consider how to stop the VI in a safely controlled way. This is a critical point because users can remotely manipulate expensive equipment, and therefore we should design safety actions to prevent them from damaging the hardware. In this sense, these actions should follow certain requirements as mentioned in Christophe Salzmann and Denis Gillet's work,[11] such as

- physical equipment should be identifiable to define what kind of equipment is connected;
- equipment should be fully controllable and diagnosable by the controlling computer;
- physical equipment's full controllability can't be exposed to the outside world, for security reasons (controllability also implies that it's always possible to place the equipment in a known state); and
- other requirements such as reliability and maintainability should also be considered.

In this context, every Web-enabled environment currently developed at UNED follows a common pattern to fulfill these requirements. In particular, the heat-flow VI has a Boolean control, labeled Stop, to finish the local control loop and to execute a resetting code before closing the application (see `RESET HW` in Figure 11). The reset code zeroes out the heating element and resets the blower, which closes the communication channels and finally returns the VI to its initial state (see `INIT HW` in Figure 11).

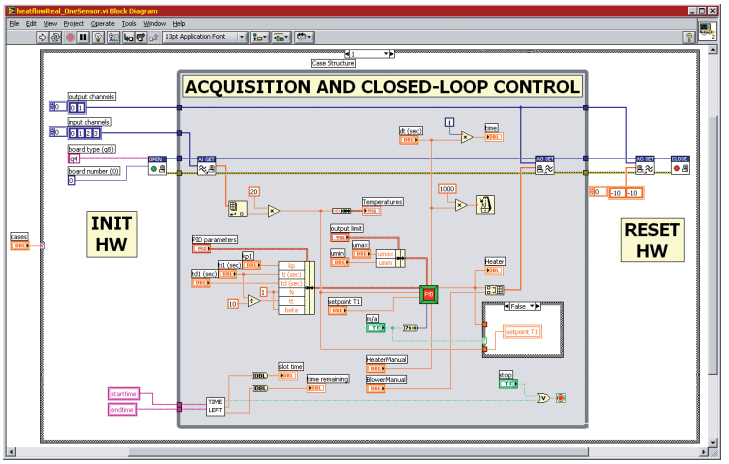

Figure 11. Local control of the heat-flow laboratory equipment. For simplification, consider the wiring diagram as being divided into three logical sections executed sequentially: `INIT HW`, `ACQUISITION AND CLOSED-LOOP CONTROL`, and `RESET HW`. The `INIT HW` section contains the blocks needed for hardware initialization purposes. The section in the middle constitutes the closed control loop—that is, the reading of the temperature sensor, the PID controller, and the sending of the control action to the heater. The `RESET HW` is the section in which the user places the blocks to reset the hardware at the completion of the heat-flow experiment.

Publishing the Local Control VI in the JIL Server

To make the VI available via the Internet, just run the JIL server and select the `heatflow.vi` file in its interface. No special communication facilities need to be included in the VI.

Applet Modification to Communicate with the VI

To modify an applet, an additional page of model variables is created in EJS that will connect to the external file

```
<Labview:62.004.199.xxx:xxxx>heatflow/
heatflow.vi
```

(We deliberately omitted IP and port information.) We declare as many variables as are required by the VI to write its controls and read its indicators. Figure 12 shows a partial view of the resulting table of variables.

Because we originally designed the applet to display a computer simulation of the thermal process, we had to modify the view to allow switching between the displays of the virtual (simulated) and the remote (real) processes. The user triggers the transition from virtual to remote by clicking a button in the user interface, labeled Connect. This button invokes the code shown in Figure 13.

The identifier `_external` in this code refers to a JIL object created automatically by EJS. Invoking this code will establish the connection, and EJS will automatically send to the VI the values of all model variables connected to the controls a number of times per second (as indicated in the EJS evolution panel) or whenever the model is updated. EJS will also read the indicators connected to the model's variables. A final `if-then` block in an EJS evolution page decides whether to use the computer model's variable values or those of the remote process to update the view, depending on the connection status. Updating the view with the correct values causes it to display an accurate description of the state of the simulated or real process.

Figure 14 shows a view of the EJS applet connected with the remote VI. We also added to the view an Internet-enabled, video-capturing element that displays the input of a webcam as background to the 3D representation. The result is a very nicely augmented, realistic effect. Because the air-heating process isn't visible, the video image is enhanced with a colored 3D image representing the system's current air temperature.

In 2008, several Spanish universities implemented the JIL approach for fast development of Web-based laboratories in control engineering. The experience shows that not having to deal with the details of the communication implementation highly improves the Web-enabled applications' maintainability and scalability. This has enabled us to quickly transform legacy Labview VIs into server-side components. Our developers were process control engineers without much programming expertise in networks or computer graphics. For this reason, we chose to program the user interfaces via EJS. Developer satisfaction is very high because they were all able to create sophisticated user interfaces with high interactivity capabilities and elaborate 2D and 3D schematic representations of the processes.

Our approach lets Labview developers concentrate their programming efforts in developing the VI core and leave the VI's communication task implementation to the JIL server. Also, `JiL.class` provides an extremely simple API for a Java applet to communicate with a VI published via the JIL server. Finally, the extension of EJS with the JIL approach provides non-Java programmers with a convenient tool for creating advanced user

interfaces and linking Java and Labview variables with just a few mouse clicks.

Further research is oriented toward improving some of the JIL server features to include augmented communication and data management, whereas the communication is currently based exclusively on TCP sockets. We plan to include alternative communication protocols such as, for example, UDP, HTTP, HTTPS, and so on. At this time, only primitive data types can be exchanged between a Java client and a Labview VI, so we plan to add the possibility to exchange Labview clusters, JPEG pictures, and videos.

Acknowledgments

This work was supported in part by the Spanish Ministry of Science and Technology under project DPI 2007-61068 and the IV Regional Plan of Scientific Research and Technological Innovation (PRICIT) of the Autonomous Region of Madrid under project S-0505/DPI/0391.

References

1. A.P. Kalogeras et al., "Vertical Integration of Enterprise Industrial Systems Utilizing Web Services," *IEEE Trans. Industrial Informatics*, vol. 2, no. 2, 2006, pp. 120–127.
2. S. Hua, C. Dai, and R.P. Knott, "Remote Maintenance of Control System Performance over the Internet," *Control Eng. Practice*, vol. 15, no. 5, 2007, pp. 533–544.
3. I. Calvo et al., "A Methodology Based on Distributed Object-Oriented Technologies for Providing Remote Access to Industrial Plants," *Control Eng. Practice*, vol. 14, no. 8, 2006, pp. 975–990.
4. D. Gillet, A. Nguyen, and Y. Rekik, "Collaborative Web-Based Experimentation in Flexible Engineering Education," *IEEE Trans. Education*, vol. 48, no. 4, 2005, pp. 696–704.
5. N. Duro et al., "An Integrated Virtual and Remote Control Lab: The Three-Tank System as a Case Study," *Computing in Science & Eng.*, vol. 10, no. 4, 2008, pp. 20–29.
6. D.L. Shirer, "Labview VI Adds Internet Features to Data Acquisition Environment," *Computing in Science & Eng.*, vol. 3, no. 4, 2001, pp. 8–11.
7. F. Esquembre, "Easy Java Simulations: A Software Tool to Create Scientific Simulations in Java," *Computer Physics Comm.*, vol. 156, no. 2, 2004, pp. 199–204.
8. J.L. Guzmán et al., "Web-Based Remote Control Laboratory Using a Greenhouse Scale Model," *Computer Applications in Eng. Education*, vol. 13, no. 2, 2005, pp. 111–124.
9. A. Visioli and F. Pasini, "A Virtual Laboratory for the Learning of Process Controllers Design," 7th *IFAC Symp. Advances in Control Education*, Elsevier IFAC, 2006.
10. W. Christian, *Open Source Physics: A User's Guide with Examples*, Addison-Wesley, 2007.
11. C. Salzmann and D. Gillet, "From Online Experiments to Smart Devices," *Int'l J. Online Eng.*, vol. 4, special issue, 2008, pp. 50–54.

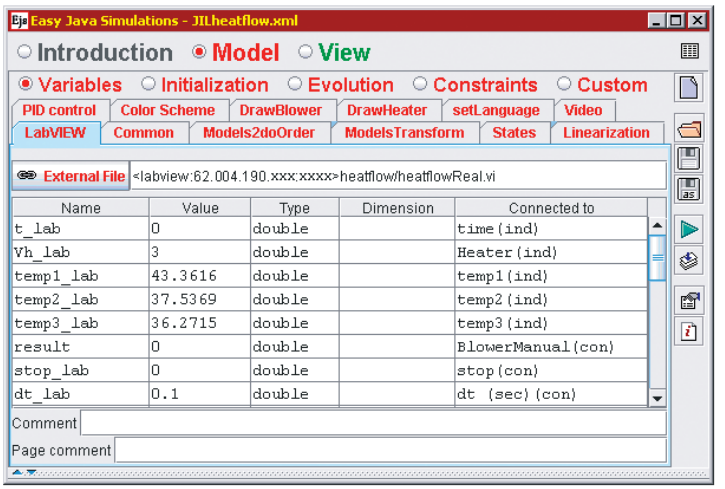

Figure 12. Table of Easy Java Simulations (EJS) variables connected to the VI controls and indicators. The list of the local EJS variables used in this example are located in the column Name. Each one of these variables is connected to a control and an indicator to send or receive information to and from the heat-flow system, respectively. The names of the controls and indicators connected to the EJS variables are in the column "Connected to" (`ind` means indicator and `con` means control).

His research interests include Web-based system design for control education. Vargas has a master's degree in electronics from the University of the Frontier, Chile. Contact him at hvargas@bec.uned.es.

***José Sánchez-Moreno** is an associate professor in the Department of Computer Science and Automatic Control at UNED. His research interests include event-based control, networked control systems, and remote and virtual laboratories in control engineering. Sánchez-Moreno has a PhD in the sciences from UNED. Contact him at jsanchez@dia.uned.es.*

***Sebastián Dormido** is a full professor of automatic control in the Department of Computer Science and Automatic Control at UNED. His research interests include automatic control and Web-based labs for distance education. Dormido has a PhD in physics from the Complutense University of Madrid. Contact him at sdormido@dia.uned.es.*

***Christophe Salzmann** is a senior research associate at the Ecole Polytechnique Fédérale de Lausanne (EPFL). His research interests include new Web technologies, real-time control, and real-time interaction over the Internet, with an emphasis on quality of service and bandwidth adaptation. Salzmann has a PhD in the sciences from EPFL. Contact him at christophe.salzmann@epfl.ch.*

```
try {
_external.connect();
_external.run();
} catch (Exception e) {
System.out.println("Error
   when connecting to the VI");
}
```

Figure 13. Code invoked when a user presses the Connect button. The first line opens the TCP connection with the JIL server; the second one runs the published VI. From this point, the data exchange is with the real system and not with the simulated model.

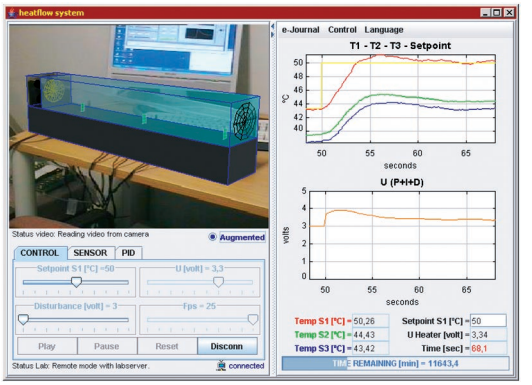

Figure 14. Experimentation console. When the real setup is used for experimentation instead of the model, the Augmented button lets users display a video image of how the three sensors' current outputs affect the duct's temperature.

Denis Gillet *is an associate professor at EPFL. His research interests include optimal and hierarchical control systems, distributed e-learning systems, sustainable interaction systems, and real-time Internet services. Gillet has a PhD in control systems from EPFL. Contact him at denis.gillet@epfl.ch.*

Francisco Esquembre *is an associate professor at the University of Murcia. His research interests include simulations for didactical purposes. Esquembre has a PhD in mathematics from the University of Murcia. Contact him at fem@um.es.*