In this paper we present Alligator, a Web system for distributed visual programming. Based on a client-server extensible modular architecture, its purpose is to allow people without any particular computer skills to build their own applications, by connecting to a Web site and visually composing data-flow programs.

1. INTRODUCTION
Nowadays, direct manipulation paradigms typical of iconic interfaces have become natural even for children, while interaction mechanisms based on the use of mouse and direct access to information have found in the World Wide Web a breeding ground for new communication opportunities. Sometimes, however, net users would like to go beyond simple information retrieval, to more actively interact with the Web, maybe to create personal applications. By providing a Web-based visual programming environment, the Alligator project goes towards such direction.

The data-flow programming language used is very simple and guides the user in program composition. When dealing with users who are not experienced at using computers, data-flow visual programming can play a very important role, as it is generally simpler and more intuitive than the “traditional” (textual) one. In a previous work [1], we demonstrated that, by composing simple data-flow visual programs through the VIPERS programming environment [2], it becomes especially easy to build simple Web applications. Alligator goes a step further, by allowing the user to work within a Web site (without the need for installing any specific program) and by providing a distributed mechanism for the execution of functional elements (“blocks”).

2. SYSTEM OVERVIEW
The Alligator visual programming environment is basically characterized by two frames, one showing the available libraries (on the left), the other displaying the programming area (on the right). Pages relative to libraries are dynamically generated by the Alligator’s main server, where all the blocks available for program composition (whether they reside on the main server itself or have corresponding functionalities provided by other machines) are stored in a database. By selecting a specific library, the corresponding blocks are displayed in a page in the left frame (Figure 1).

The source code of such page contains full information about blocks, in the form of JavaScript structures. By choosing any block in a library and clicking on a cell of the programming grid, the block is placed in it. The visual program is therefore built, which results in a special data-flow graph without explicit links. Graphically, blocks appear to be fitted one in the other, like in LEGO® building blocks (from which the name Alligator stems), and data is intended to flow from up to bottom. The use of this simple strategy eliminates the undesired “spaghetti” effect typical of data-flow graphs. Each block is characterized by an icon, enclosed within a colored frame, reminding its function. Once a program has been built, on the client computer, and checked for errors (all activities carried out through JavaScript), it is properly coded and sent to the Alligator server (the same from which the initial pages were downloaded). Here it is processed block by block, invoking, when necessary, distributed execution of some parts on remote computers and managing the relative flow of data. Typically, programs implemented within the Alligator environment include output blocks which, when executed, add their data to a “result page”. When program execution is finished, the server returns to the user such page (in a new window).
3. ARCHITECTURE AND PROTOCOL

Alligator is a client-server application composed of three kinds of machines: the main server, execution servers and clients. Clients are the user machines, i.e. those on which visual programming is carried out through a Web browser. The main server takes care of managing requests from clients and invoking processings on execution servers, where the actual execution of some blocks occurs. Execution servers take also care of returning their results to the main server. The architecture just proposed is schematically represented in Figure 2.

![Alligator's basic architecture](Image)

The Alligator global interaction protocol can be summarized as follows. 1) A client connects to the main server and downloads the Alligator environment. Information about all the available library blocks is locally stored by means of JavaScript structures. 2) The user creates a visual program and each block may be thought of as an instruction whose operating code is the pair made up of two identifiers, one indicating the block itself and the other referring to the server on which its execution will occur. On the main server, the firing engine (a ColdFusion module in charge of executing program graphs) behaves as a “processor”, which, once an “instruction” has been recognized by means of its operating code, starts its execution, directly or by transferring control to the proper execution server. 3) When program construction is concluded, the program is coded in all its parts and sent to the main server. 4) On the main server, the program “description” is saved. If necessary, possible files provided as inputs are saved as well. 5) Still on the main server, the firing engine is invoked to execute the various instructions of the program. 6) Each block is executed. If it is not a standard one, managed directly by the main server, its execution requires the activation of a “program” on the corresponding execution server (to which possible input data is provided and from which possible results are received). By “program” in this context we intend a CGI program or any other “executable” code accessible through a URL (e.g. a ColdFusion module, an ASP page, etc.). Note that the execution server may coincide with the main server. 7) The firing engine goes on executing the program, sending/receiving data to/from the involved execution servers and storing possible intermediate results. When the STOP block is encountered, execution halts. 8) The result page is sent back to the client.

4. SYSTEM TESTING

We have tested the Alligator system in different application scenarios, such as Web applications development and basic image processing. For example, Figure 3 shows a program which receives a JPEG image and, after converting it into the PGM (8 bit gray levels) format, produces its histogram (indicating the distribution of gray levels over the 256 possible values).

![An example of image processing application implemented in Alligator](Image)

A second histogram is then calculated for the same image after it has been normalized, to show the difference. Referring to the figure, from top to bottom, blocks have the following purposes. After requesting to the user the path name of a JPEG image file, the first block sends it to the Alligator main server, where it is then converted into the PGM format (second block). At this point, processing follows two different paths. In the left branch of the graph, the converted image is saved (so that it can be displayed in the result page), the gray levels histogram is extracted and then it too is saved. The same processing is carried out in the right branch, but, this time, after a normalization. Block STOP, at last, halts program execution.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have described Alligator, a system for distributed Web-based data-flow visual programming. By allowing blocks’ functionalities to be defined anywhere within the Internet, the system lends itself to incremental library development, thus becoming exploitable for a large variety of potential applications (which we will explore shortly).

Some aspects of the project are still in a development stage and will be the object of future research. In particular, usability tests we will carry out on the implemented prototype will allow the human-machine interface and the interaction mechanism to be better defined.

6. REFERENCES
