Programming Web Multimedia Applications with Hop

Manuel Serrano
Inria Sophia Antipolis
2004 route des Lucioles - BP 93 F-06902 Sophia Antipolis, Cedex, France
http://www.inria.fr/mimosa/Manuel.Serrano

Abstract

Hop is a new execution platform for running interactive and multimedia applications on the Web. It is aimed at executing applications such as Web agendas, Web galleries, Web music players, etc. Hop consists of: *i)* a new programming language specially designed for addressing the distributed aspects of Web programming, *ii)* a rich set of libraries for dealing with music files, sounds, pictures, photographs, etc., *iii)* a full-fledged Web server for executing the *server-side* components of the applications.

In this paper we illustrate Hop's skills for programming multimedia applications in two examples. We show that, with 50 lines of code, an operational photograph gallery can be implemented and we show that with approximatively 30 lines of code an operational *podcast* receiver can be built.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Applicative (functional) languages, Concurrent, distributed, and parallel languages, Design languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*

General Terms

Design, Languages

Keywords

Web programming, Functional programming

Download

The Hop's Web site, which is entirely implemented in Hop, contains the distribution of the system, its source code, the online documentation, and various demonstrations. Although Hop can be executed on the three main-stream platforms (namely, Linux, MacOS X, and Win32), it is recommended to install it on Linux, the system that suits it best.

Version:	`1.6.0`
License:	GPL
Port:	Linux, MacOS X, win32
Home:	`http://hop.inria.fr`
FTP:	`ftp://ftp-sop.inria.fr/mimosa/fp/Hop`

MM'07, September 23–28, 2007, Augsburg, Bavaria, Germany.
ACM 978-1-59593-701-8/07/0009.

1. INTRODUCTION

Thanks to its last technical breakthroughs, the Web has become a vector of choice for deploying multimedia applications. First, during the last decade, the latency of the communication was drastically reduced. That is, the speed of communication has increased while, in the same time, effective compaction algorithms have emerged. Hence, nowadays, downloading a picture may take less than a second and a whole music record or a full movie only a couple of minutes. Secondly, Web browsers now can deal with all kind of multimedia resources such as texts, images, sounds, movies, and graphical animations. They support rich, responsive, and programmable graphical user interfaces. Together, these two improvements have given birth to the most famous popular modern Web applications such as *Google Map*, *YouTube*, *Flickr*, etc. However, behind this terribly attractive face, the Web has a dark side: programming such multimedia applications on the Web is still generally awfully complex. This is the point addressed by Hop [2,3] which aims to simplify the development of Web applications.

Developing programs on the Web is complex for several reasons: *i)* A Web application is *de facto* distributed because it executes in parallel on several computers (at least two, a server and a client). *ii)* Web applications have to deal with many different kinds of resources (images, sounds, texts, streams of data, etc.). *iii)* Idiosyncrasies of Web browsers are strong. Their scripting language (namely JavaScript) is not 100% compatible. They do not exactly implement the same set of features. They do not use the same APIs. *iv)* deploying applications is difficult because, usually, it involves several servers that need to be configured and tuned. *v)* Most widely used programming languages have not been specially crafted for the Web. They are not well equipped for addressing the issues mentioned previously. *vi)* Since the Web consists in opening resources to several users and networks, by nature, it is exposed to attacks and piracy. Hence, the Web demands strong security enforcement. Hop addresses all these issues but the last.

- The Hop programming language exposes a model based on two computation levels. The first level is in charge of executing the logic of an application while the second level is in charge of executing the graphical user interface. Hop separates the logic and the graphical user interface but it packages them together and it supports strong collaboration between the two engines. The two execution flows communicate through function calls and event loops. Both ends can initiate communications.

- In contrast to most other approaches, requiring a Tower of Babel of programming languages, Hop embraces all the aspects of a Web application (ranging from server configuration to finest graphical tuning of the client) in one single unified formalism and syntax.

- Hop provides a rich set of libraries for dealing with most of the standard formats used on the Internet. For instance, it implements the RFCs dedicated to network programming (mails, HTTP, mime, distributed calendars, ...). It can deal with EXIF meta-information of photographs. It can stream audio files and handle play lists. It embeds generic parser generators and parsers for XML, HTML and RSS. Etc.

In this paper, in Section 2, the overall architecture of Hop is presented and its implementation briefly sketched. In Section 3, in order to give an intuition of the flavor of writing programs with Hop, a simple photograph gallery is presented. This section is used to present informally the main constructions of the Hop programming language. At last, in the Section 4, in order to show another facet of multimedia programming with Hop, the whole source code of an actual *podcast* client is presented.

2. HOP ARCHITECTURE

The traditional Web architecture restricts the sets of operations clients (i.e., the programs contained in the Web pages that are executed by the Web browsers, denoted as *Web pages* or even *pages* when the context is unambiguous) are allowed to perform. For security matters, all browsers enforce the "*same domain restriction*" preventing a downloaded Web page from initiating communications with other servers. Also for security reasons, the only resources a Web page is allowed to access to are resources attached to the graphical user interface built by the Web browser. In particular, the page is not able to access to the file system of the computer running the Web browser. Since they constitute the minimal security protection against malicious Web sites, these elementary rules are unlikely to be once disabled. At first glance, they seem to prevent from using the resources obviously needed for implementing multimedia applications. For instance, how would it be possible to implement a photograph gallery if the photographs stored on the disk cannot be accessed! How is it possible to implement a mashup combining resources scattered all of over the network if the page is only allowed to communicate with one unique Web server!

Augmenting the capacities of the browsers with *plugins* that offer restricted services (such a reading a dedicated directory) is one way to address this problem. This solution has been adopted by Google Gears which provides a native plugin or by Flapjax which provides a portable plugin implemented in Flash. Hop has adopted a different solution. It does not modify the browsers, but it relies on a home brewed lightweight Web server. Since it has a small memory footprint several instances of such server can be run simultaneously on a single computer. In particular, for each application, a new server can be started. Hence, one server may be implementing a photograph gallery, while another may be implementing a music media center, a third one, a web mail, etc. These servers can be considered as *smart proxies*, however, since they are full-fledged Web servers we prefer the term *Web broker*.

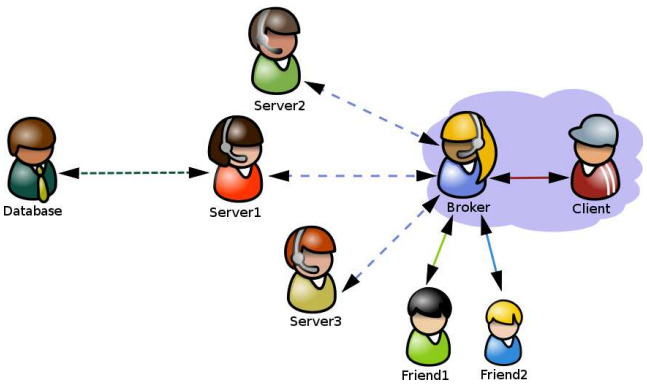

Figure 1: The typical Hop advocated architecture.

Installing a broker is more complex than installing a plugin. However, this drawback is balanced by several advantages: *i)* a Web broker can be accessed by different clients, then it can be used to implement multi-users applications such as an application opened to all the members of family not necessarily located on the same sub-network (an application for sharing photographs, for instance). *ii)* The broker is a regular process managed by the operating system. As such, it is administrated as any regular process. In particular, it can be *sandboxed* using mechanisms *à la* `chroot` or virtual machines. *iii)* The broker can be used as a daemon for running background tasks such as nightly updates of a podcast database. Figure 1 shows the topology of a common Hop application, such as a *mashup* involving three different content servers.

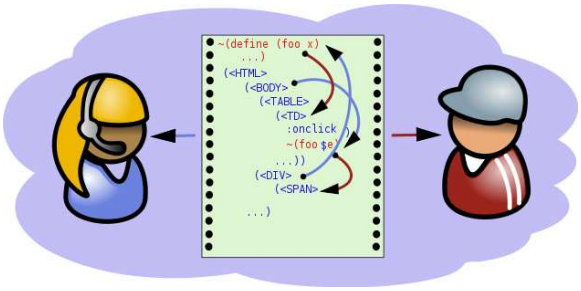

Figure 2: A Hop program is composed of a single source code that addresses simultaneously the server and the client.

As illustrated in Figure 2, a Hop program specifies *in a single source code* the behavior of the broker *and* and the client. The code source contains syntactic annotations that specify if an expression belongs to the client or to the broker. That is, if an expression has to be evaluated on the client or on the broker. Expressions evaluated on the broker may refer to variables and functions of the client and *vice-versa*.

When a program is loaded on the broker, two compilers transform it into executable codes. A native compiler compiles the expressions belonging to the broker. This compilation is highly optimized so it delivers programs whose

performance is approximatively two to three times slower than equivalent C hand-written programs. The client code is compiled by A Hop-to-JavaScript compiler. The generated code generated by this compiler is as fast as hand-written JavaScript code [1].

3. A PHOTOGRAPH GALLERY

In this section a simple application for displaying photographs extracted from a digital camera is presented. The application collects all the photographs available on the repository located on the broker's disk, builds a thumbnail list, and let clients visualize photographs one by one. When a photograph is selected, it appears on screen with a rotation and zooming effect. Of course this application is executed from a Web browser. Figure 3 presents a screenshot of this application.

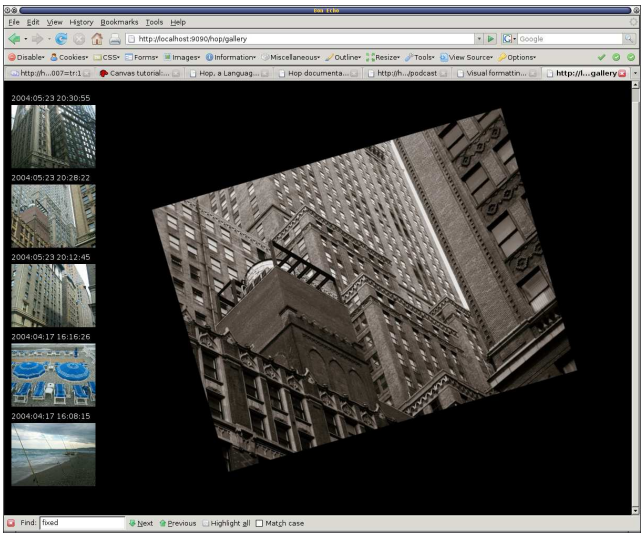

Figure 3: A screenshot of Firefox executing the photograph gallery.

At first glance, a Hop page looks like an HTML page where each markup is preceded with an open parenthesis, where closing markups are replaced with closing parenthesis, and where attributes name start with a colon.

```
1: (define can
2:    (<CANVAS> :width ~(- window.innerWidth 200)
3:              :height ~window.innerHeight))
```

In addition to providing markups, Hop also supports variable and function declarations. Line *1* contains the declaration of a variable name `can` whose value is an HTML tree denoting a canvas, *i.e.,* a drawing area. Another important departure from HTML, that can be seen on line *2*, rests in the value of attributes. In HTML these values are static strings of characters. In Hop, any legal expression of the language can be used to feed an attribute. In our example, we build a canvas slightly smaller than the browser's window in order to leave room for the photograph thumbnails that will be introduced afterward. The `~` character, that is used in the attribute value, plays a fundamental role in Hop: it switches from broker-side code to client-side code. That is, in our example, the canvas is built on the broker, because the page is built on the broker and *then* shipped to the client. However, the broker ignores the current width of the client window. Then, the actual width of the canvas is only known when the canvas is actually created on the client. The `~` character delays the evaluation of the expression until this occurs.

We are now ready to present the main part of our application:

```
 5: (define-service (gallery)
 6:   (<HTML>
 7:     (<HEAD> (<TITLE> "A Web Photograph Gallery"))
 8:     (<BODY> :style "background: black"
 9:       ~(define img (new Image))
10:       ~(define ctx #f)
11:       ~(define angle 0)
12:       ~(define zoom 0.1)
13:       ~(define (draw-image)
14:           (set! zoom (+ 0.1 zoom))
15:           (set! angle (+ 0.4 angle))
16:           (ctx.fillRect 0 0 $can.width $can.height)
17:           (ctx.save)
18:           (ctx.translate
19:            (/ $can.width 2) (/ $can.height 2)))
20:           (ctx.rotate angle)
21:           (ctx.scale zoom zoom)
22:           (ctx.drawImage img
23:            (- (/ img.width 2)) (- (/ img.height 2)))
24:           (ctx.restore))
25:       ~(define (reset-anim! src)
26:           (set! img.src src)
27:           (set! ctx ($can.getContext "2d"))
28:           (set! angle 0)
29:           (set! zoom 0.1)
30:           (timeout-start "anim" 100 draw-image))
31:       (thumbs repository))))
```

The form `define-service` introduces a function that can be invoked from a browser following the REST convention. That is, our program is started by redirecting a Web browser to the URL `http://<your-server>/hop/gallery`. The drawing of the image being traditional it does not deserve many explanations. It is noteworthy that standard Web APIs can be reused *as is* in Hop without any declaration or wrapping.

This example shows another important construction of Hop, namely the `$` escape sign. This is the exact opposite of `~`. It switches from *client mode* to *broker mode*. In this example, the variable `can` which has been declared on the broker is unbound on the client. So, in order to be used by client-code, it has to be marked with the `$` sign.

The definition of the function `thumbs`, that extracts the photographs from the disk and presents them on the Web page, can be defined as follows:

```
32: (define (thumbs repo)
33:   (<TABLE>
34:     (<TR> (<TD> (map <THUMB> (directory->list repo)))
35:           (<TD> canv))))
```

The function `directory->list` can only be used on broker code since it actually reads the disk of the broker and packages the file names inside a list. The function `map` is an iterator that applies its first argument to the elements found in its second argument. This example shows that, in Hop, markups play a role similar to functions. They can be applied to arguments in order to build HTML trees but they can also be used as value, for instance passed as argument.

In order to complete this first application, the implementation of the <THUMB> markup has to be presented. Contrary to regular HTML markups, it is not predefined in Hop.

```
35: (define (<THUMB> path)
36:  (let ((exif (jpeg-exif path)))
37:    (<DIV>
38:      (<DIV> (exif-date exif))
39:      (<IMG>
40:         :src (img-base64 (exif-thumbnail exif))
41:         :onclick
42:         ~(with-hop ($(service () (img-base64 path)))
43:             reset-anim!)))))
```

The broker library function jpeg-exif reads a JPEG image from the disk and extracts the embedded EXIF information. It returns a data structure which is used on line *38* for getting the date of the photograph and on line *40* for extracting the thumbnail generated by the camera. The function img-base64, as suggested by its name, builds a base64 inline representation of an image.

The form with-hop is central to Hop, it lets client code invoke functions defined on the broker. In the example on line *42* it is used to invoke an anonymous function returning the actual content of a photograph. When this function call completes, the result is sent to the function reset-anim! defined on line *25*. In this example, the with-hop function call has been used for minimizing the memory footprint of the client. Instead of shipping all the photographs with the main interface, the application only ships lightweight images thumbnails and, on demand, selected photographs.

4. A PODCAST RECEIVER

This second example, illustrates the Hop's ability to parse XML documents on the broker and play music on clients. These features are used for building a simple yet operational Podcast receiver. This receiver presents a set of predefined podcast streams to the user which can start and stop playing the streams.

```
1: (define-service (podcast)
2:   (<HTML>
3:     (<HEAD> :include "hop-sound")
4:     (<BODY>
5:        ~(define snd #f)
6:        ~(define (play-url url)
7:            (set! snd (make-sound url :stream #t)))
8:        (<BUTTON> "Stop" :onclick ~(sound-stop snd))
9:        (map <PODCAST> podcast-list))))
```

The client-side function make-sound (line *7*) downloads and plays a sound. Once initiated, the sound can be interrupted by the means of the sound-stop function (line *8*). In our example, the user defined markup <PODCAST> associates, in the GUI, a button to each podcast stream.

```
10: (define (<PODCAST> podcast)
11:   (let ((url (rss->podcast-mp3 (car podcast))))
12:     (<BUTTON> :onclick ~(play-url $url )
13:        (cadr podcast))))
```

The function with-url (line *15*) opens, from the broker, an internet connection with the host referenced to in the URL it receives as first argument and it invokes the function it receives as second argument, providing it with a stream of characters obtained from the connection.

```
14: (define (rss->podcast-mp3 url)
15:  (with-url url
16:    (lambda (p)
17:      (bind-exit (return)
18:        (xml-parse p
19:          (lambda (markup attrs body)
20:            (if (eq? markup 'enclosure)
21:              (return (cdr (assq 'url attrs))))))))))
```

The function xml-parse (line *18*) parses a stream of characters denoting an XML document and seeks for the `enclosure` element denoting the actual MP3 document implementing the sound. When such a markup is found on the stream, its `url` attribute is extracted and the parsing aborted. The list of podcast rss streams can be simply defined as follows:

```
24: (define podcast-list
25:   '(("http://rf.net/rss_10053.xml" "France Musique")
26:     ("http://rf.net/rss_20000.xml" "FIP") ...))
```

5. CONCLUSION

This paper has presented the Hop platform, an execution environment for Web applications. Its programming language has been informally presented as well as its main components, the Hop Web broker, and its two compilers. Two operational multimedia applications have been presented: a photograph gallery and a podcast receiver. These examples show how compact Hop programs can be. The first example counts less than 50 lines of code while the second only 30!

As of May 2007, the Hop Web broker can be installed on any regular PC running any main-stream operating system. In addition, successful experiments for installing it on NASes (Network Attached Storages) have been conducted. On these architectures, it has been used for implementing a cheap, convenient, and highly available, multimedia home server. We plan to port it to personal modem/routers. This is appealing because these equipments are cheap and highly available. This is also challenging because they have dramatically limited resources.

In parallel, Hop is also ported to modern handheld computers. These small devices are port-friendly because, in spite of their reduced size, they propose resources comparable to that of low-end laptops. On these platforms, we will use Hop for running the same applications as mentioned above.

6. REFERENCES

[1] Loitsch, F. and Serrano, M. – **Hop Client-Side Compilation** – Proceedings of the 8th Symposium on Trends on Functional Languages, New Yort, USA, Apr, 2007.

[2] Serrano, M. – **The HOP Development Kit** – Invited paper of the Seventh Acm sigplan Workshop on Scheme and Functional Programming, Portland, Oregon, USA, Sep, 2006.

[3] Serrano, M. and Gallesio, E. and Loitsch, F. – **HOP, a language for programming the Web 2.0** – Proceedings of the First Dynamic Languages Symposium, Portland, Oregon, USA, Oct, 2006.