Anatomy of a Ubiquitous Media Center

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

The Web is such a rich architecture that it is giving birth to new applications that were unconceivable only few years ago in the past. Developing these applications being different from developing traditional applications, generalist programming languages are not well suited. To help face this problem, we have conceived the Hop programming language whose syntax and semantics are specially crafted for programming Web applications. In order to demonstrate that Hop, and its SDK, can be used for implementing realistic applications, we have started to develop new innovative applications that extensively relies on the infrastructure offered by Web and that use Hop unique features. We have initiated this effort with a focus on multimedia applications.

Using Hop we have implemented a distributed audio system. It supports a flexible architecture that allows new devices to catch up with the application any time: a cell phone can be used to pump up the volume, a PDA can be used to browse over the available musical resources, a laptop can be used to select the output speakers, etc. This application is intrinsically complex to program because, *i)* it is distributed (several different devices access and control shared resources such a music repositories and sound card controllers), *ii)* it is dynamic (new devices may join or quit the application at any time), and *iii)* it involves different heterogeneous devices with different hardware architectures and different capabilities.

In this paper, we present the two main Hop programming forms that allow programmers to develop multimedia applications more easily and we sketch the parts of the implementation of our distributed sound system that illustrate *when* and *why* Hop helps programming Web multimedia applications.

Keywords: Web 2.0, mobile Web, ubiquitous Web, programming languages, functional languages.

1. INTRODUCTION

The Web is a new territory for programmers, yet to be explored. New Web applications have already started to show up with the Web 2.0 but we are likely to be only at the beginning of a revolution. There are some reasons to believe that Web applications will take over traditional applications because, in addition to connecting computers, the Web is also one of the most efficient architecture ever invented for connecting people. On top of the Web, one may build a telecommunication system that embraces telephony, radio or television broadcasting, a platform for exchanging public or private photographs, music, movies, books, a navigation system, etc. These are just some examples, many others exist. These new possibilities have been hardly used by computer programs yet, probably because developping applications that uses them is still complex.

Developping Web applications is difficult for several reasons. One of the most important ones is the lack of convenient development tools. In particular, traditional languages are poorly equipped for coping with the specificities of the Web: some cannot easily deal with the common formats and protocols of the Web such as XML, RSS, JSON, CSS, or WebDAV, and most of them can hardly address the asymmetric distributed model of the Web (Web servers and Web clients have different roles and different capabilities). In an attempt to bridge that gap between programming languages and Web applications, Hop, a new programming language, has been proposed [**?bib sgl:dls06**]. Hop differs from traditional programming languages by some syntactic elements and by some forms specially crafted for the Web. The Hop execution environment relies on a full-fledged Web server that embeds a compiler generating HTML and JavaScript on-the-fly. This technique eliminates the need of any Web browser plugin or extension.

Hop exists as a software that can be downloaded at `http://hop.inria.fr` and also as an academic research project that has been described by some technical publications (Ref. [**?bib sgl:dls06**],[**?bib ls:tfp07**]). We are now trying to ground a community of users. In particular, we are trying to appeal programmers by developing and releasing new original Web applications that are difficult to program using traditional systems. Our first effort in this direction focuses on the domain of interactive multimedia applications. This is the subject of this paper.

Using Hop we have implemented a *ubiquitous home media center* named HopAudio. Taking advantage of the ubiquity of the Web, this application implements features that other programs used for playing music rarely propose. HopAudio can use many sources of music and radios and it can control several output speakers. All the electronic devices that can run a Hop broker (i.e., a dedicated Web server) can be used to *serve* musical content. All the devices that can run a stock web browser can be used to *control* the music being played back. HopAudio can be considered as a *realistic prototype*. It is operational and used on a daily basis althought some implementation details must still be polished and some minor features must still be added.

Multimedia applications have specific demands on programming languages. For instance, the runtime systems of the languages must obviously provide means for playing music! This was missing in the previous versions of Hop and has been added to the system. This extension and its implementation are presented in **Section 5**. In addition to these domain-specific extensions, we have found it useful to provide the language with a new communication channel between servers and clients that allows the formers to *push* information to the latters. This feature is not specific to multimedia applications and can be used in many different contexts, for instance, when servers and clients must be synchronized. In HopAudio, it is used to synchronize the graphical user interfaces of all the devices. This new facility is presented in **Section 4**.

Once Hop's extensions are presented, the **Section 6** presents the main components of HopAudio as well as some implementation details that illustrate the programming patterns favored by Hop. Since the readers of this paper are not expected to be familiar with the Hop programming language, a brief presentation of its main features is first sketched in **Section 3**.

2. A UBIQUITOUS MEDIA CENTER

HopAudio, the *Home Media Center* presented in this paper, implements a distributed HI-FI system on the Web. As shown in Figure 1, it consists in a aggregate of relatively inexpensive heterogeneous electronic devices such as PDAs, cell phones, modems/routers, laptops, Network Attached Storages (NASes), etc. They all can be used to browse and select the music, to select the speakers used to play the music, and to control the various settings associated with the players (e.g., volume level, fading, etc.). These devices are spread out amongst the various rooms of a personal habitation. They all communicate using the HTTP protocol. They are all controlled by a Hop program. The architecture of the application is flexible because new devices can be *dynamically* added or removed at any time.

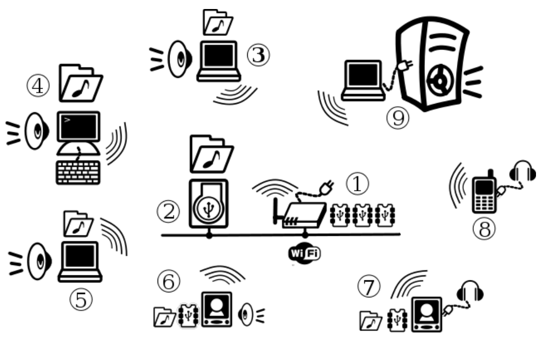

Figure 1: A personal *Ubiquitous Home Media Center* using common electronic devices (PDAs, laptops, NAS, ...).

- ① is a modem/router. Currently it only assumes the role of a wireless access point but in the close future it is envisioned to extend it with facilities for downloading and storing podcast feeds on memory flash storage.

- ② is a personal Network Attached Storage (NAS). It is equipped with a PowerPC G3 processor and 64MB of memory which is more than needed for running a Linux system and a Hop Web broker (see Section 3). Its processor uses between 1W and 6W depending on its activity. It controls a large hard disk. This NAS being equipped with USB connectors it could be connected to external speakers.

- ③,④, and ⑤ are regular x86 personal computers. They manage local music repositories. They can play music using embedded speakers.

- ⑥, and ⑦ are two PDAs. They can store up to 32GB of data on Flash memory cards. They are equipped with speakers that offer mediocre sound quality. Although this is enough for playing informative radio broadcasting, these devices have audio output connectors and Bluetooth adapters that can be used to connect headsets or external speakers for better sound quality. These PDAs have 128MB of memory, they run Linux operating system. They use an ARM v6 processor. These small computers offer quite decent CPU performance (approximatively a fifth of a contemporary PC) which enables them to run unmodified full-fledged Web browsers such as Opera and Firefox, to play music, and to serve music files via a Hop server. They use very little energy. They are totally silent since they are fanless and diskless.
- ⑧ is a wifi cell phone. It runs a Linux operating system. It comes with 64MB and an ARM v5 processor which is enough for hosting a Hop broker and running a Opera Web browser. Although it could be used to play music, this device is mostly used for controlling the music played by the other computers.
- ⑨ is a low-end Pentium III laptop connected to a HiFi system. This laptop is used for playing high quality music. It has 128MB of memory, it runs the Linux operating system and, as all the other devices, it runs a Hop broker.

Even though all these devices have different hardware specifications they are all able to run full-fledged Web browsers and to run a Hop Web broker. Some devices are able to server music content (②,③,④,⑤,⑥, and ⑦). Some others are able to play music (③,④,⑤,⑥,⑦, ⑧, and ⑨). However, *all* devices can be used to browse the audio content available and *all* devices can be used to control the music being played. For instance, they can *all* select the audio speakers to activate. They can *all* change the volume level. The roaming cell phone ⑧, can be used to switch off the audio output from the HiFi system installed in the living room and to switch on the speaker of the PDA ⑥ installed in the kitchen. The same cell phone can be used to control the audio volume and to select a new audio content, for instance an MP3 file available on the NAS ② located in the cellar. All the configuration panels and controls are accessible through Web interfaces. Figure 2 shows a photograph of a Web GUI implementing the audio controller when executed on a PDA.

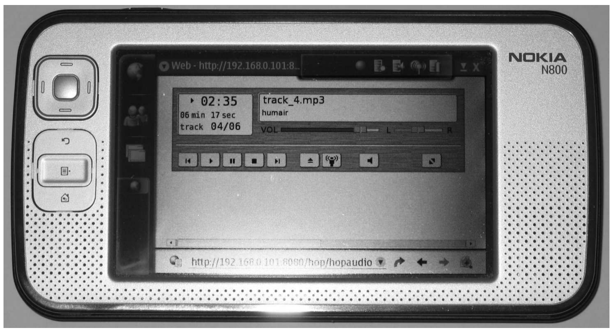
NOKIA
N800
Web - http://192.168.0.101:8...
02:35
06 min 17 sec
track 04/06
track_4.mp3
humair
VOL
http://192.168.0.101:8080/hop/hopaudio

Figure 2: HopAudio on a PDA.

The key point of this installation is that the audio content, the selection of the audio speakers, and the volume level, can all be controlled by any of the devices already mentioned, wherever they are located in the apartment. The second important caracteristic is that new devices can be *dynamically* added to the running application *at any time*. Each newly added device can be used to either control the application or bring new resources such as new musics or new output speakers. These new resources are instantly visible by all the other devices. All communications and data transfers *exclusively* use the HTTP protocol. All GUIs rely on Web technologies and they are all executed inside *unmodified* bare Web browsers (i.e., no plugin is needed). In spite of the complexity of the application, mostly due to the management of various distributed resources, the whole source code of the application counts less than 3.000 lines of Hop code. This compactness, that we regard as an advantage of Hop, comes from the high-level features specially designed for the Web programming, this system provides. The rest of this paper explains the most relevant parts of this implementation.

3. HOP

Hop is an execution platform for running interactive and multimedia applications on the Web [**?bib serrano:mm07**]. It consists of: *i)* a **programming language** specially designed for addressing the distributed aspects of Web programming, *ii)* **two compilers** for producing *server-side* and *client-side* executable codes [**?bib ls:tfp07**], the client-side compiler generates HTML and JavaScript code, hence, no plugin nor extension is needed on the browsers *iii)* a **rich set of libraries** for dealing with music files, sounds, pictures, photographs, etc., and, *iv)* a full-fledged lightweight multi-threaded **Web server** for executing the *server-side* components of the applications. This server has a small memory footprint and it is highly portable so it has been successfully installed on various architectures such as the ones presented in Section 2. Frequently, we refer to the Hop Web server as a *Web broker*.

The Hop programming language exposes a model based on two computation levels. The first one is in charge of executing the logic of an application while the second one is in charge of executing its 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 notifications (see Section 4) which allow both ends to initiate communications. The most visible specificity of Hop is to allow programmers to write entire Web applications using a *single* formalism.

3.1 Hop syntax

At first glance, the Hop programming language seems to be a mere variation around HTML because a HTML document can be syntactically translated into a Hop source text by inserting an extra open parenthesis in front of each opening markup and replacing each closing markup with a single closing parenthesis. The following example shows a HTML document and its equivalent counterpart in Hop:

```
1: <HTML>                                         (<HTML>
2:   <TABLE>                                        (<TABLE>
3:     <TR><TD onclick="alert( ’1’ )">1</TD></TR>        (<TR> (<TD> :onclick ~(alert "1") 1))
4:     <TR><TD onclick="alert( ’2’ )">2</TD></TR>        (<TR> (<TD> :onclick ~(alert "2") 2))
5:     <TR><TD onclick="alert( ’3’ )">3</TD></TR>        (<TR> (<TD> :onclick ~(alert "3") 3))))
6:   </TABLE>
7: </HTML>
```

In spite of the similarities, this example also exposes an important difference between the HTML version and the Hop version. While in HTML, client-side expressions (expressions evaluated by the Web browsers for reacting to user actions) are expressed in a foreign programming language, namely JavaScript (Ref. 3,4), in Hop, they are expressed in the same language as the one used for markups. Hop client-side expressions are regular expressions but distinguished from server-side expressions by a syntactic mark: client-side expressions are prefixed with a ~ sign (see, for instance, on line *3*).

Since the large majority of Web sites are now dynamic, several solutions have emerged for generating HTML pages. Most of them consist in inserting, inside HTML documents, special *sentinel* markups used by the Web servers for invoking dedicated interpreters on the text of these markups. The following example shows two versions of a program generating a HTML table of `MAX_ROW` rows. The left hand-side contains a HTML+PHP version, the right hand-side contains a Hop version. Since Hop is a full-fledged programming language, there is no need to embed *alien* expressions inside programs. The dynamic parts of the interface are expressed in the same language as the static parts:

```
1: <HTML>                                          <HTML>
2:   <TABLE>                                         (<TABLE>
3: <?php                                               (map (lambda (j)
4: for( $i=1; $i<=MAX_ROW; $i++ )                               (<TR> (<TD> :onclick ~(alert $j) j)))
5:    echo "<TR><TD onclick=’alert($i)’>$i</TD></TR>"        (iota MAX-ROW))))
6: ?>
7:   </TABLE>
8: </HTML>
```

In Hop, the `$` sign inside the client-side expressions is the opposite of the ~ sign. It allows client-side expressions to refer to server-side expressions. Hence, on line *4*, the `$j` expression inserts, in the client-expression, the value of the server variable `j`. [*]

[*]Readers familiar with Lisp, will recognize a strong analogy between Hop’s ~ and `$` and Lisp’s ‘ and ,.

3.2 Hop Ajax

Hop allows programs to define functions located on the server but invoked by the clients. These special functions are introduced by the keyword `service`. As any regular function, a service may receive arguments and return values of any type. The following example defines a service that scans the server hard disk (using the server function `directory->list`) and that returns a list of pairs made of the name of the files and their size (given by the server function `file-size`).

```
(define du
   (service (dir)
      (<TABLE>
         (map (lambda (e) (<TR> (<TH> e) (<TD> (file-size e))))
              (directory->list dir)))))
```

Services are invoked by clients by the means of the `with-hop` special form that takes two parameters: a service call and a callback function which is invoked when the call completes. In the following example, the `with-hop` form is used to invoke the server service named "`du`" with the string `"/tmp"` as argument. The resulting list is used, by the client, to build a new HTML page that is inserted in the initial HTML document.

```
(let ((console (<DIV> "")))
  (<HTML>
     (<BUTTON> "du" :onclick ~(with-hop ($du "/tmp") (lambda (table) (innerHTML-set! $console table)))
     console)))
```

This presentation of the Hop programming language is, on purpose, as succinct as possible. It only contains the forms that are used in the rest of the paper. A more complete presentation of the language and its libraries can be found in [**?bib sgl:dls06**], or visiting the Hop Web site located at `http://hop.inria.fr`.

4. SERVER EVENTS

All the devices participating to HopAudio have to be aware of changes occurring on the system. They all have to be *synchronized*. When the volume is raised by a client, all the other clients have to reflect this modification in their own GUI. When a new song starts, all clients have to show information relative to that new song. When a speaker is switched off and another one switched on, clients must be notified. For ensuring the synchronization, clients have to *react* to *external events*. In order to support these notifications, Hop extends the traditional Web event model (the DOM 2 event model) with a new kind of events: the *server* events which allow the server to initiate communication with one or all its clients. In this section, the rationale of these events and their implementation are presented. First, the Hop API for dealing with *client* events is presented. Then its extension for supporting *server* events is shown.

4.1 Hop event model

In Hop, event listeners are managed by two functions, `add-event-listener!` and `remove-event-listener!` which conform to the DOM 2 event model and which, contrary to JavaScript, hide the idiosyncrasies and portability issues of the Web browsers. The rest of this presentation omits the description of `remove-event-` `-listener!` because its behavior can be deduced by symmetry from `add-event-listener!`. The signature of `add-event-listener!` is as:

```
add-event-listener!: <obj> × <string> × <procedure> [× <bool>]
```

The first argument denotes the HTML element of the DOM tree that will hold the new event listener. The second argument, a string of characters, denotes the type of events the listener is attached to (e.g., "mouseclick", "mousemove", ...). The third argument is a listener. It is a procedure of one argument that will be bound to the reified raised events. The optional fourth argument is a boolean controlling events capture and propagation, as specified in the DOM 2 event model. These being standard in Hop, they are not presented in this paper. The example below shows a source code implementing an area reactive to mouse moves:

```
1: (let ((area (<DIV> :style "height: 200px" "")))
2:    (<HTML>
3:       ~(define (show e) (innerHTML-set! $area (format "~a x ~a" (event-x e) (event-y e))))
4:       ~(define (enter e) (add-event-listener! document "mousemove" show))
5:       ~(define (leave e) (remove-event-listener! document "mousemove" show))
6:       ~(add-event-listener! $area "mouseover" enter)
7:       ~(add-event-listener! $area "mouseout" leave)
8:       area))
```

When the mouse enters the reactive area the `mouseover` listener (line *6*) is invoked. This installs a global listener for "mousemove" events (line *4*). This listener inserts inside the element `info` the position of the mouse each time it moves. When the mouse exists the reactive area, the listener `leave` is invoked (line *7*). This removes the `mousemove` listener. Note that since HTML only supports detection of mouse moves on the global document, this example shows the *standard* way to intercept mouse moves.

4.2 Server events API

A *server event* is a new type of event to which listeners can be attached. A *server event* is emitted by a server and intercepted by clients. It is identified by its name. It carries a value. Conforming to the same origin security policy advocated by the Web, only the server that has provided a page may signal events to that page. An application that uses server events is sometimes refereed to as a *comet* application.

The Hop machinery exposed to programmers for dealing with server events on the client-side code is minimalist since it relies on a tiny extension of the event model as presented in Section 4.1. Clients simply attach listeners using the already mentionned `add-event-listener!` function to the event type named "`server`". Such listeners being always associated with the entire document, the first argument of `add-event-listener!` no longer denotes a node of the document but the name of the server event that is listened to. For instance, a client needing to intercept the server events named `tick` may use:

```
(add-event-listener! "tick" "server" (lambda (e) (alert "tick received: " e.value)))
```

On the server side, two functions emit events. The first one, `hop-event-signal!`, emits an event to *at most one* client. The second one, `hop-event-broadcast!`, emits an event to *all* clients. These functions are designed after `pthread_cond_signal` and `pthread_cond_broadcast` of the Posix Threads library (Ref. 8). The following excerpt shows a server loop continuously emitting the event named "`tick`":

```
(let loop ((count 0))                   ;; start the loop
   (hop-event-broadcast! "tick" count) ;; signal all interested clients
   (sleep 1000000)                      ;; wait for a second
   (loop (+ count 1)))                  ;; loop unconditionally
```

The section 5.1 shows how server events are used for implementing the Home Multimedia Application.

4.3 Server events implementation

A detailled description of the implementation of server events being out of the scope of the present paper, only a brief sketch of the implementation is given here. Hop uses two different schemes for implementing server events. The first one, is based on a small Flash library embedded inside the Hop client-side runtime system. It is fast and reliable. It opens a socket per clients which is used to send events. A second implementation, used when Flash is not supported by the browser, exclusively relies on JavaScript. Although it is portable, it is unfortunately intrinsically unefficient and not reliable because JavaScript does not support keep-alive HTTP connections. This imposes to create a new HTTP connection each time the server has to send an event to a client. The period during which the server and the client are disconnected may yield to lose some events. To minimize the risk of losing events, Hop uses remember sets. However, since these sets are finite they cannot accumulate infinitely new events. Hence, as unlikely as it is, it might happen situations where the disconnected period last too long which forces the remember set to flush out some accumulated events.

5. THE NEW <AUDIO> MARKUP

A multimedia application for playing music obviously needs some sort of interface for browsing the available musical contents, for controlling the volume level, and for selecting the audio speakers. In the context of the Web, for the sake of coherence, such interfaces must be implemented inside Web pages and then presented in the language as a HTML binding. Henceforth, Hop has been enriched with the new `<AUDIO>` markup whose interface is inspired by the current HTML 5 working draft (Ref. 5). This extension is a central element for building applications such as HopAudio.

Although the Hop markup is compatible with the current HTML 5 proposal, it brings a major extension: it does not impose the actual music player to be hosted on the Web browser. That is, `<AUDIO>` can be used, inside a graphical user interface, for controlling a remote music player, for instance, located on the Hop broker. The *Ubiquitous Web Applications Working Group Charter*, refers to such a markup as a *proxy* to an actual hardware device. In the rest of this paper, this device is referred to as the *back-end* player. This extension to the `<AUDIO>` markup has only required the addition of one new attribute for selecting the back-end player. Hence, Hop's `<AUDIO>`, in addition to supporting the regular HTML 5 attributes `src`, `autoplay`, `start`, `loopstart`, `loopend`, `end`, `loopcount`, and `controls`, also supports the new attribute: `player`. A large part of HTML 5 audio events have been implemented in Hop. That is, `progress`, `loadedmetadata`, `load`, `error`, `play`, `pause`, `ended`, and `volumechange` are supported. The description of these events is omitted here because their names are intuitive and because they conform to the HTML 5 draft. The following example illustrates how to use the Hop's `<AUDIO>` markup.

```
 1: (define (podcast->mp3 url)
 2:    (parse-xml-from-url url
 3:       (lambda (markup attributes body)
 4:          (when (eq? markup 'enclosure)
 5:             (attribute-value attributes 'url)))))
 6:
 7: (define-service (podcast)
 8:   (<HTML>
 9:      (<AUDIO> :controls #t
10:        :player (instantiate::mpd (host "localhost") (port 6600))
11:        :src (podcast->mp3 "itpc://radiofrance-podcast.net/rss_10466.xml"))))
```

This program spawns, inside the client browser, an audio player (line *9*) which streams the MP3 material whose URL is obtained from parsing (line *2*) a *podcast* stream. The `controls` attribute of the markup being set to true (line *9*), a graphical controller is associated with the music player such as the one presented Figure 2. On this example, the audio player is executed on the server (line *10*) using MPD.[†]

5.1 Implementing the <AUDIO> markup

Although Hop implements a large part of the HTML 5 audio specification, it can be executed by any browser that *i)* implements HTML 4 or XHTML 1.0, *ii)* implements the DOM level 2, and *iii)* is equipped with a standard Flash plugin. In other words, the Hop audio implementation does not require modified or HTML 5 compliant Web browsers. Practically, it can be executed on all mainstream browsers.

The audio facility involves two different implementations. A first one is used for controlling local players, i.e., Flash players embedded in Web browsers. A second one is used for controlling remote players. The first implementation consists in executing a Flash movie object in which a small audio API for MP3 play back has been implemented. This technique is straightforward and fairly standard so it is not detailed in this paper. The second implementation that is used for controlling remote audio players is unusual, in particular because it extensively uses the Hop *server* events that have been presented in Section 4. This implementation is detailed in the rest of this section.

[†]Music Player Daemon available at: `http://www.musicpd.org`.

5.1.1 The client-side implementation

On a client, an audio player is implemented by an object containing a DOM node for the graphical controller (a node which can either be visible or not) and a proxy for allowing the client to communicate with an actual *back-end* player. This audio proxy implements all the methods required for controlling the play back, e.g., `play`, `stop`, `pause`, `load`, `volume-set`, and `volume-get`. Using the browser back-end these methods are directly mapped to Flash methods. Using the server back-end these methods are mapped to Hop service calls (see Section 3.2). For instance, the `play` and `volume-set` proxy methods are implemented as:

```
(set! audio.proxy.play (lambda () (with-hop ($server-player audio.key "play"))))
(set! audio.proxy.volume-set (lambda (v) (with-hop ($server-player audio.key "volume-set" v))))
```

The service `server-player` is in charge of issuing the `play` and `volume-set` commands to the back-end player it controls. The argument `audio.key` identifies one particular player in the server that might be controlling several.

Players may be used simultaneously by several clients. Hence, any change to a player state, for instance, when the volume level changes or when a play-back completes, must be sent to all clients. In order to avoid continuously *polling* information from the server, these clients register a listener to a server event that is broadcast when a change on the player occurs. This can be implemented as:

```
(add-event-listener! "audio-play" "server"
 (lambda (e)
    (set! audio.pause #f)
    (set! audio.current-duration e.value)
    (audio-gui-trigger-event audio.gui e.id)))
```

When a server broadcasts an audio event, all the clients invoke their listener. The parameter `e` of these listeners is bound to an event descriptor. Before completing, a proxy listener triggers itself an event for the DOM object containing the GUI that must update its content.

5.1.2 The server-side implementation

The implementation of the `server-player` service is straightforward:

```
1: (define-service (server-player key command . args)
2:    (let ((player (find-player key)))
3:       (cond
4:          ((string=? command "play") (%music-play player))
5:          ((string=? command "volume-set") (%music-volume-set! player (car args)))
6:          ...)))
```

First, the actual player associated with the service parameter `key` is retrieved from a hash table (line *2*). This object is used by the server functions that control the hardware music player (lines *4* and *5*). In order to ease the reading of the source code, these low-level functions have been all prefixed with the `%` sign.

Hop supports bindings for several music controllers (MPD, MPlayer, Gstreamer, mpg123, etc.). When a low-level back-end does not support events notification, Hop actively polls information for the player by executing an infinite loop inside a dedicated thread. It can be implemented as:

```
(define (server-player-loop key player)
  (let loop ((oldstate 'stop) (oldvol -1) (oldsong -1))
     (multiple-value-bind (st pl sng pos len v err)
        (%music-info player)
        (cond
           ((string? err) (hop-event-broadcast! "audio-error" err))
           ((not (= v oldvol)) (hop-event-broadcast! "audio-volume" ,v))
           ((not (eq? st oldstate)) (hop-event-broadcast! "audio-state" `(,st ,pl ,sng, pos ,len)))
           ...))))
```

At each iteration of the loop, information is fetched from the music player. The form `multiple-value-bind` (line *3*) is used to invoke a function (here the function `%music-info`) that returns several values that are bound to a list of variables (here `st`, `pl`, etc.). When a change in the state is observed, the server broadcasts an event whose value denotes the new music player state (lines *6*, *7*, and *8*).

6. ANATOMY OF A MULTIMEDIA APPLICATION

When the HopAudio application starts on a broker, it reads a configuration file that describes the broker's resources. A broker may be in charge of controlling musical contents, output speakers, or both. A broker controls its own resources which means that it uses them by itself, for instance for playing music, and that it may also provide them to other brokers. New brokers can catch up with the application at any time, offering also new resources. In the section 5 it has already been presented how music players are programmed. In this section, it is presented how resources are mananaged in HopAudio.

6.1 Dealing with dynamic music repositories

Each Hop broker participating to HopAudio can offer musical materials and output speakers to the rest of the community. Since, from an implementation point of view, managing these two kinds of resources is roughly equivalent, this section only presents the implementation of the musical contents management. This aspect of HopAudio is a good representative of the Hop genuine distributed programming style because it involves several participants, one client and several brokers.

Offering music repositories

Once the configuration file is loaded, the instance of HopAudio knows its repositories of musics. It offers this list to the other brokers using the `hopaudio/repositories` service which is implemented as:

```
1: (define-service (hopaudio/repositories)
2:    (if (authorized-service? (current-request) 'hopaudio/repositories)
3:        (hopaudio-repositories)
4:        (user-access-denied (current-request))))
```

The standard Hop function `current-request` returns a descriptor of the request being currently served by the broker. It contains fields such as the client's IP address, name and port. In addition, if the request is authenticated, the descriptor also contains a pointer to the user authentity.

Of course, not all Hop brokers of the planet are allowed to execute the `hopaudio/repositories` service. Hence, the service first checks (see on line *2*) if the authenticated user is allowed or not to get the list of HopAudio repositories.

The GUI for adding repositories

One of the panels of the HopAudio's GUI allows users to dynamically add new servers of musical contents. It is implemented by a HTML table which, by convenience, is initially filled with information relative to the client that is using the application.

```
(let ((host (<INPUT> :type 'text :value (request-hostname (current-request))))
      (port (<INPUT> :type 'text :value "8080")))
 (<DIV>
   (<TABLE> (<TR> (<TH> "host") (<TD> host))
            (<TR> (<TH> "port") (<TD> port)))
   (<BUTTON> "Add" :onclick ~(add-music-server! host.value port.value))))
```

When the `Add` button is clicked, the function `add-music-server!` is invoked on the client with the values extracted from the GUI panel.

Adding repositories: the client side

Because of the *same-origin* security policy enforced on the Web, a client is not allowed to contact remote Hop brokers. Hence, when a client wishes to add a new server to the application, it must delegate this registration to its origin broker. The resources offered by the various devices involved in HopAudio must be obviously protected from undesired use. In particular, the access to the musical materials must be granted only to correctly authenticated participants. In consequence, when a client requests an access to a broker, it must be ready to authenticate itself. A client registers to a new music server using the service `hopaudio/add-music-server!` which might then return a special code for requesting an authentication, on behalf of a remote broker. It should be noted that this kind of remote authentication differs from the direct authentication that takes place between a server and a client which is natively supported by Web browsers. In order to provide the remote authentication information, the client-code pops up a login panel (see on line *7*) that lets the user identify himself.

```
1: ~(define (add-music-server! host port)
2:   (let loop ((user #f)
3:              (pass #f))
4:    (with-hop ($hopaudio/add-music-server! host port user pass)
5:      (lambda (h)
6:        (case h
7:          ((authenticate) (login-panel :onlogin (lambda (e) (loop e.user e.password))))
8:          ((success) ...)
9:          ((failure) ...))))))
```

This example shows that the execution flow of a Hop application, keeps switching from clients to servers and vice-versa.

Adding repositories: the broker side

The broker-side implementation for handling music server registration consists in contacting the remote server in order to obtain its list of repositories.

```
1: (define-service (hopaudio/add-music-server! host port user password)
2:    (with-hop (hopaudio/repositories)
3:       :host host :port port :user user :password password
4:       (lambda (paths)
5:          (for-each (lambda (p) (store-music-url! (format "http://~a:~a~a" host port p))) paths)
6:          'success)
7:       (lambda (xhr) (if (=fx (xml-http-request-status xhr) 401) 'authenticate 'failure))))
```

As clients, servers can also use the `with-hop` form (see on line *2*). In contrast to clients, servers must specify a host name and port number to identify the target broker (see on line *3*). As previously seen, the service `hopaudio/repositories` either returns a list of directories or a failure status code. On success, the absolute paths are transformed into URLs (see on line *5*) and stored locally on a broker. On failure, the appropriate result code is returned to the client which will have to authenticate itself.

These examples have shown how Hop helps programming using the *Web style* of distributed computing. By supporting an original execution control flow that lets a program go back and forth from client to broker, broker to broker, and broker to client, Hop enables compact source codes for Web applications.

7. RELATED WORK

In this section, we first compare Hop to other programming languages and systems for the Web. Then, we compare server events to other proposals. We also presents some Web applications related to HopAudio.

7.1 Related languages

Hop assumes that programs execute in parallel on at least two computers: a server and a client. Hence, Hop programs are not sand-boxed inside Web browsers and they can control as many resources as regular non Web-based programs. This feature enables, for instance, a Hop program, whose graphical user interface is executed on a cell phone, to control the music played back by a desktop computer, and vice-versa. To our knowledge no other system offers simultaneously a programming language and an execution platform (i.e., a lightweight Web server) that cover the whole spectrum of the resources needed by the Web applications.

Some systems, such as *Ruby on Rail* or *Google GWT*, attempt to unify the programming language used for developing the whole Web application. However the philosophy of these systems differs from the one promoted by Hop. These systems attempt to adapt the mainstream programming models to the Web. For instance, the key idea of GWT, is, as much as possible, to hide the underlying Web technologies to programmers, for instance, by using an object oriented model for programming GUIs and by enabling a Java-like programming for Web applications. Hop promotes an opposite design choice: it relies on the *Web-way* to implement GUIs. That is, a Hop program *generates* HTML pages, it deals with the DOM, it uses Cascade Style Sheets, it can use directly JavaScript libraries, etc. On the one hand, the Hop's approach carries the obvious drawback of forcing programmers to learn a new system with new unusual concepts. On the other hand, we think that once Hop is tamed, it allows programmers to write simple and compact source codes.

Few other research works focus on designing languages for the Web that embrace both client- and server- side programming. To our knowledge, only Hilda (Ref. 12) and Links (Ref. 2) share this characteristic with Hop. These three languages enforce a single-paradigm programming model (the functional paradigm for Links and Hop) and then, we think that they avoid, at least partially, many of the pitfalls of traditional Web programming (Ref. 7). Neither Hilda nor Links focus on developing reactive or multimedia applications. Hilda focuses on data-driven applications, that is applications that run on top of a back-end data system. Links is a generalist programming language but it seems that its implementation is still not mature enough to permit one to use it for developing realistic applications such as the one presented in this paper.

7.2 Related Events

An application using server events (*server push*) is sometime refereed to a *comet* application (Ref. 10). In a previous study [**?bib sgl:dls06**] we have proposed a different model for server events. The former API relied on the a new HTML markup dedicated to intercepting server events. An event handler associated with that markup was responsible for handling characters received from the server. The current HTML5 *Server-sent DOM events* (Ref. 5) follows a similar path. It extends HTML with the new markup <`event-source`> to which a JavaScript `onmessage` handler can be attached. This handler is responsible for parsing strings of characters in order to reconstruct event values. The new approach chosen by Hop and presented in this paper differs from these two design proposals because it does not extend the set of markups but it extends the DOM event API. Contrary to other systems, Hop server events are not transmitted as strings of characters but regular values of the language.

Hop server events are related to *Remote Events for XML (REX)* as presented in (Ref. 11). Remote events constitute a notification mechanism for remotely sending events to HTML nodes. On the one hand, Hop server events, which can only be used to notify an entire documents, are simpler than REX events. On the other hand, Hop server events are not restricted to transport information about the nodes of DOM. They are "general purpose" because they can carry any kind of information. A REX-compliant implementation could be easily built on top of Hop server events. For that, it would only be required to pack, inside a event value, the identify of the node to be notified and the operation to be executed on that node, and to implement a standard listener that would locally handle the nodes notifications.

The current Hop API of server events is asymmetric since only servers can send values to clients. In the future, it might be worth extending this model to allow client-to-server notifications as well as server-to-server notifications. It might also be considered adding more higher order primitives than the current ones. For instance, implementing the Erlang messages and mailboxes (Ref. 1) on top of current Hop events could be considered in the future.

7.3 Related multimedia applications

Personal applications relying exclusively on Web technologies obviously pre-existed to Hop. For instance, most of the administration tools for modems and routers are implemented inside dedicated embedded Web servers. Some rare multimedia applications also already existed. For instance, S5 and Slidy (Ref. 9) are two systems for authoring slides on the Web. These two applications are on the radar of Hop which is exactly designed for simplifying the implementation of this kind of applications.

Inside the InHoNets project an experiment for building a distributed media center (DMC) (Ref. 6) has been conducted. At first glance DMC shares many characteristics with HopAudio. Both systems rely on Web technologies, both focus on audio playback, both assume distributed independent devices for browsing and controlling the music. In spite of these similarities, the two systems deeply differ. First, it seems that DMC requires a centralized architecture with a server always running because it is in charge of scheduling all the communications between all the devices. In contrast, the model proposed in the paper can be used in a peer-to-peer fashion and the system is usable as soon as one device (which can be a cell phone or a PDA) is up. The second major difference comes from the implementation of these two systems. The solution presented in the paper for implementing the application relies on a programming SDK that is designed for supporting distributed applications. Hence, it enables compact and simple implementations. The description of DMC does not give any indication of the complexity of the development *per se* but it details its architecture. The system is built on the top of a whole stack of technologies (XML, XSTL, REX, XPath, ...) that necessarily carry their own complexity.

8. CONCLUSION

This paper presents a novel distributed audio system exclusively based on Web technologies. All the electronic equipments that can browse the Web or run a Hop Web broker can participate to this application. In the paper, it has been presented an experiment where all sort of regular electronic devices (PDAs, cell phones, laptops, NASes, etc.) have been equipped with a Hop server. Since, all these devices can also run a Web browser, they all play a symmetric role: they can all serve musical content and/or control the music currently played back by other devices. Even if this application still is a prototype, it is robust enough to be used by the author on a daily-basis. The paper has emphasized the simplicity of implementing such distributed multimedia applications on the Web with the Hop system.

9. REFERENCES

[1] Armstrong, J. *et al.* – **Concurrent Programming in ERLANG** – *Prentice Hall*, 1996.

[2] Cooper, E. *et al.* – **Links: Web Programming Without Tiers** – 5th International Symposium on Formal Methods for Components and Objects, Nov, 2006.

[3] Ecma, – **ECMA-262: ECMAScript Language Specification** – **http://www.ecma-international.org/publications/standards/Ecma-262.htm**, 1999.

[4] Flanagan, D. – **JavaScript – The definitive guide (fourth edition)** – *O'Reilly Associates*, USA, 2002.

[5] Hickson, I. and Hyatt, D. – **HTML 5 W3C Editor's Draft 25 September 2007** – World Wide Web Consortium, Sep, 2007.

[6] Kleimola, J. and Vuorimaa, P. – **Declarative techniques in Distributed Media Center System** – proceedings of the W3C workshop on Declarative Models of Distributed Web Applications, , Jun, 2007.

[7] Mikkonen, T. and Taivalsaari, A. – **Web Applications – Spaghetti Code for the 21st Century** – SMLI TR-2007-166, Sun microsystems, Jun, 2007.

[8] Nichols, B. and Buttlar, D. and Proulx Farrell, J. – **Pthreads Programming** – *O'Reilly Associates*, USA, 1996.

[9] Ragget, D. – **Slidy - a web based alternative to Microsoft PowerPoint** – Xtech'06: Building Web 2.0, Amsterdam, The Netherlands, May, 2006.

[10] Wilkins, G. – **Comet is Always Better Than Polling** – Comet Daily, **http://cometdaily.com/2007/11/06/comet-is-always-better-than-polling**, , 2007.

[11] World Wide Web Consortium, – **Remote Events forXML (REX) 1.0** – W3C Working draft, Oct, 2006.

[12] Yang, F. e. a. – **A Unified Platform for Data Driven Web Applications with Automatic Client-Server Partitioning** – 16th International World Wide Web Conference (WWW'07), Alberta, Canada, May, 2007, pp. 341–350.