A Glimpse of Hopjs

Manuel Serrano, Vincent Prunet

A Glimpse of Hopjs

Manuel Serrano, Vincent Prunet

Inria Sophia Méditerranée
2004 route des Lucioles - F-06902 Sophia Antipolis, France
{Manuel.Serrano,Vincent.Prunet}@inria.fr

Abstract

Hop.js is a multitier programming environment for JavaScript. It allows a single JavaScript program to describe the client-side and the server-side components of a web application. Its runtime environment ensures consistent executions of the application on the server and on the client.

This paper overviews the Hop.js design. It shows the JavaScript extensions that makes it possible to conceive web applications globally. It presents how Hop.js interacts with the outside world. It also briefly presents the Hop.js implementation. It presents the Hop.js web server implementation, the handling of server-side parallelism, and the JavaScript and HTML compilers.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages, Design languages; D.3.4 [*Programming Languages*]: Processors—run-time environments

Keywords Web Programming, Functional Programming

1. Introduction

Multitier programming languages unify within a single formalism and a single execution environment the programming of the different tiers of distributed applications. On the web, this programming paradigm unifies the client tier, the server tier, and the database tier when one is used. This homogenization offers several advantages over traditional web programming that relies on different languages and different environments for the two or three tiers: development, maintenance, and evolution are simplified by the use of a single formalism, global static analyses are doable as a single semantics is involved, debugging and other runtime tools are strengthened by the global knowledge of the whole execution [24].

The first three multitier platforms for the web, GWT (a.k.a., Google Web Toolkit), Links [8], and Hop [25], all appeared in 2006. Each relied on a different programming model and a different language. GWT mapped the Java programming model to the web, as it allowed Java/Swing like programs to be compiled and executed on the web; Links was a functional language with experimental features such as storing the whole execution context on the client; Hop was based on the Scheme programming language [15], adapted to the genuine web programming model where HTML is a container for embedded client-side expressions. These three pioneers have opened the way for other multitier languages that have followed (Ocsigen for OCaml [1, 2, 28], Ur/Web [5], Jscala [16, 23], iTask3 [10], Cheerp for C++, etc).

In spite of their interesting properties, multitier languages have not yet become that popular on the web. Today, only GWT is widely used in industrial applications but GWT is not a fully multitier language as it requires explicit JavaScript and HTML programming in addition to Java programming. We think that the lack of popularity of the other multitier systems comes mostly from their core based languages rather than from the programming model itself. This is why we are adapting the multitier paradigm to JavaScript, the mainstream web programming language. We have chosen to adapt the programming *à la Hop* for its proximity with traditional web programming.

JavaScript is the *de facto* standard on the web. Since the mid 90's it is the language of the client-side programming and more recently it has also become a solution for the server-side programming, popularized by Node.js[1]. Our proposal is to enable multitier programming in JavaScript by extending the language and by implementing a new execution platform that manages the server-side and the client-side of an application. The JavaScript extension is called HopScript, the execution environment is called Hop.js. This environment contains a builtin web server, on-the-fly HopScript and HTML compilers, and many runtime libraries.

HopScript is a super set of JavaScript, *i.e.,* all JavaScript programs are legal HopScript programs. Hop.js is a compliant JavaScript 5 execution environment [12]. It also integrates some functionalities of ECMAScript 6 (arrow functions, rest arguments, template strings, generators, ...). The Hop.js environment aims at Node.js compatibility. The current version supports about 90% of the Node.js runtime environment. In particular, Hop.js fully supports the Node.js modules, which lets Hop programs reuse existing Node.js modules as is.

The rest of the paper presents HopScript and Hop.js with the following organization: Section 2 presents the architecture of Hop.js web applications; Section 3 presents the server-side programming; Section 4 presents the client-side programming; Section 5 shows how an HopScript program interacts with the outside world; Section 6 shows how multitier programming helps debugging web applications; Section 7 sketches the Hop.js implementation. Section 8 compares it to other web programming languages and environments.

2. Hop.js Architecture

Hop.js applications are organized as traditional web applications, illustrated in Figure 1. A Hop.js application starts when a Hop.js

[1] `https://nodejs.org/`

server is spawned. It first loads its HopScript program, which starts (`bootstrap`) the server-side execution of the application. This might consists in a mere initialization stage or in a more complex task.

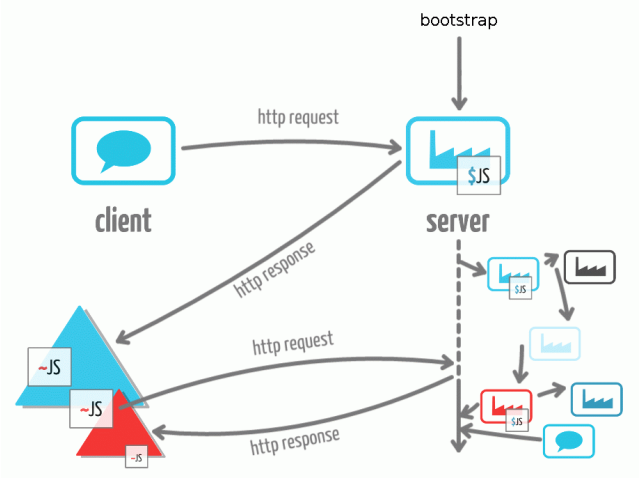


Figure 1. Anatomy of a Hop.js web application.

The second step of the execution starts when a client, typically a web browser, connects to the server and emits a request (`http request`). In response, the server *computes* a response which generally is a HTML page (`http response`). On reception the client starts executing the HopScript client-side part of the application. In that general framework, the role of HTML is twofold. It is the container of the client-side program and it is the description of the client-side UI.

User interactions may trigger new requests (`http request`), which in response might produce new HTML fragments that are inserted into the initial page. When these fragments contain client-programs, they are linked to the existing client-side HopScript program already executing.

Client-side HopScript programs may also trigger server requests (`http request`), which, in return may insert a new HTML fragment or may replace an existing one. A server-side computation may also emit requests to remote servers (the remote servers on the right hand side of the figure), which are not required to be Hop.js processes.

In the beginning of the paper server-side programs are annotated with $ marks and client-side programs with ~ marks (see Figure 1). A HopScript program consists of both programs, combined inside a single source code. The separation between the two ends takes place on the server, by compilation, as illustrated in Figure 2.

3. Server-Side Computation

The server-side of a HopScript program is a fully capable process running on a server. It can access all the resources of the hosting machine (file system, networking, sensors, ...). For reuse of existing code, HopScript is fully compatible with Node.js. This makes it possible to use most JavaScript npm packages on the Hop.js server-side. In addition, Hop.js also provides libraries of its own (builtin discovery, multimedia, etc.) and multitier operations that are described in this section.

A Hop.js application first starts as a HTTP server, which is builtin in the execution environment. The server delivers HTML documents that contain or point to some code that is installed in the runtime environment of the client and that constitutes the client-side program. During its execution, this client-side program may

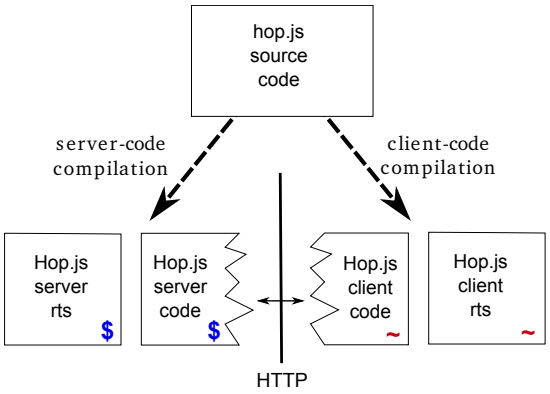


Figure 2. Separation between client-side and server-side code by compilation.

communicates with its server by invoking *services*. This is an incarnation of the Ajax (*Asynchronous JavaScript and XML*) programming paradigm. Services and service invocations are described in this section. Section 5 will show how to use Hop.js services for server-to-server communications and for third-party services interoperability.

3.1 HopScript Services

HopScript services are remote JavaScript functions callable via the network, using the HTTP protocol. Syntactically, they are distinguished from regular functions by the use of the `service` keyword. For instance, the following declares the service `readDir`, which scans the files of the directory `dir`, selecting only those whose name matches the regular expression held by the parameter `re`:

```
var fs = require( "fs"  );

service readDir( dir, re ) {
   let rx = new RegExp( re && ".*" );
   return fs.readdirSync( dir )
      .filter( s => s.match( rx ) );
}
```

A service is associated with a URL made from its name, `http://localhost:8080/hop/readDir` in our example. The routing of the HTTP requests and the marshalling/unmarshalling of the arguments are automatic and transparent to the programmer.

As any regular JavaScript function, a service may accept any number of arguments. All the arguments are packed into the special `arguments` variable. Missing arguments are filled with the `undefined` value. Services return exactly one result. In the body of a service, `this` is a JavaScript object containing informations relative to HTTP request that has triggered that service execution: the remote host name, the HTTP header, the HTTP verb, etc.

3.1.1 Service Invocation

Services are JavaScript *callable* object, but counter-intuitively, calling a service does not evaluate its body. A service call is a purely local operation. It does not involve any network traversal. It merely produces a *frame* object, which supports the two following operations:

1. *URL construction:* a frame can be converted into a URL via the `toString()` method. Using that URL in a HTTP `GET`, `PUT`, or `POST` request triggers an evaluation of the associated service body. As such, all tools and constructs that use URLs (a web

browser navigation bar, a `src` attribute of an HTML image, a `wget`-like tool, etc.) can provoke service invocations.

2. *Service invocation:* a frame can be used by HopScript programs to trigger remote service invocations, via the `post(callback)` method, or its synchronous declination `postSync()`. Invoking one of these two methods traverses the network and executes the service on the remote host. The result is sent back to the caller on the other end of the network connection. On that client, the optional `callback` passed to the `post` method is invoked asynchronously. The method `postSync` does not take a callback as parameter. It returns the result of the service invocation. Under the hood, the `post` and `postSync` methods are implemented on top of HTTP requests. The frame object merely contains the informations needed to create that request, *e.g.*, the remote host address, a pseudo path associated with the remote service, and the marshalled arguments.

Services may receive any serializable HopScript value, that is, all values but functions, sockets, and workers. As JSON serialization, the HopScript serialization does not follow the prototype chain of JavaScript objects but contrary to JSON, it preserves sharing between values, even amongst parameters.

Services URLs are compatible with the CGI protocol (*a.k.a.,* the Common Gateway Interface) that originally interfaces web servers with executable programs. This protocol describes how program arguments should be packed and encoded in URLs. Hop.js follows this convention for denoting service invocation. Service URLs can be manually forged, knowing the name of the service and the name of arguments. This feature is intensively used to start Hop.js applications or to export HopScript services as regular web services. When the CGI convention is used to invoke a service, all the arguments of the URL are encoded as JavaScript strings and packed together into on fresh JavaScript object that is passed as the single argument of the service.

3.2 Service Responses

Invoking a service produces a value. The delivery of that value to the client is handled by the Hop.js server. If needed, the server first *compiles* the value and then transmits the response, encoded as a sequence of characters, through a HTTP connection. The nature of the response compilation depends on the type of the value produced by the service. For instance, if the response is a HopScript program, it is compiled into a plain JavaScript. If the response is a JavaScript literal, it is compiled into a JavaScript expression that will recreate it on the client. If it is a HTML DOM, it is compiled into plain HTML encoded as a sequence of characters, etc.

For the sake of the example, let us assume a service `unameButton` that returns a HTML button that when clicked pops up a client window showing the server up time, collected at the service execution time.

```
// server.js
service unameButton() {
   return {
      node:
         <BUTTON onclick=~{alert(${process.uptime()})}>
           uptime
         </BUTTON>,
      version: 1.0
   }
}
```

Let us consider a client code that invokes the `unameButton` service and that appends its result to the current document.

```
// client.js
~{
   ${unameButton}
      .post( function( el ) {
         document.appendChild( el.node );
      } );
}
```

In that very example, the `unameButton` result mixes HTML and JavaScript. For the client delivery, the Hop.js server compiles it into:

```
{"node":hop_create_encoded_element("%3Cbutton%20id%3D%27
G0%27%20onclick%3D%27alert(2.102)%3B%27%3E%0A%20%20%20
%20%20%20%20%20%20%20%20%20%20%20uptime%0A%20%20%20%20
%20%20%3C/button%3E"), "version":1.0}
```

This sequence of characters is sent, accompanied with a mime type that informs the client on the procedure to be used for decoding it. In that particular example, the decoding will merely consist in *evaluating* the string, as it denotes an executable JavaScript expression.

In the example. a `HTTPResponse` is silently and automatically created without the programmer even noticing it. When needed, these objects can also be created by services explicitly. The HopScript `HTTPResponse` class (*i.e.*, JavaScript constructor) is the super class of all possible responses. Its subclasses correspond to the different ways of sending responses to clients. If the service return value is not an `HTTPResponse` object, it is silently converted using the following rules:

- XML values, which are first class HopScript values, are converted into `HTTPResponseXML` instances;
- JavaScript values (including objects and arrays) are converted into `HTTPResponseHop`;
- String literals are converted into `HTTPResponseString`.

For instance, using an explicit `HTTPResponseString` object, the `unameButton` service can be rewritten as:

```
// server2.js
service unameButton() {
   var head = { startLine: "HTTP/1.1 200 ok",
                contentType: "text/hopscript"};
   var body =
'{"node":hop_create_encoded_element("%3Cbutton%20id%3D%27'
+'G0%27%20onclick%3D%27alert('
+process.uptime()
+')%3B%27%3E%0A%20%20%20'
+'%20%20%20%20%20%20%20%20%20%20%20uptime%0A%20%20%20%20'
+'%20%20%3C/button%3E"), "version":1.0}';

   return hop.HTTPResponseString( body, head );
}
```

Constructing explicit `HTTPResponse` objects enable services to specify particular HTTP settings that might be needed, for instance, for communicating with non Hop.js servers or clients. Hop.js provides various classes for building various sort of responses. In the rest of this section, we present the most frequently used ones.

The three main properties of response objects are their HTTP status code, their payload, and their mime type. The status code is required by the HTTP protocol. It informs the client about the success or failure of its request. The mime type, also standard to the HTTP protocol, lets clients interpret the characters they receive. Using different mime types for a same payload yields to different interpretations on the client-side.

The `HTTPResponseString` class is used to deliver arbitrary sequences of characters. It can be used to send all sort of responses such status or error messages, content of a file, pre-compiled HTML values, etc.

In practice, `HTTPResponseString` are only used for transmitting values that do not denote HopScript or HTML objects or when a specific mime type is to be specified. For instance, the following example shows a service that returns the content of a file, if that file exists, and a 404 error message otherwise[2].

```
var fs = require( "fs" );

service getFile( path ) {
   if( fs.lstat( path ) ) {
      var head = { startLine: "HTTP/1.1 200 ok",
                   contentType: "text/plain" };
      var body = fs.readFileSync( path );
      return hop.HTTPResponseString( body, head );
   } else {
      var head = { startLine: "HTTP/1.1 404 no" };
      return hop.HTTPResponseString( path, head );
   }
}
```

The class `HTTPResponseString` is flexible as it can encode all sort of responses but in general it is too low level for the actual needs of the program because it requires an explicitly encoding of the payload as a string of characters. This is burdensome for the programmer and it may impact the performances as it forces the payload to be loaded into the memory and encoded before being transmitted. To solve these two problems (convenience to use and performance) Hop.js supports other classes for responses.

Speed: reading a file into memory for delivering it to clients as a string is inefficient on most platforms. Modern operating systems provide more efficient methods that directly write files content to sockets, without even entering the operating system user land. The Hop.js `HTTPResponseFile` class maps this feature to HopScript. It can be used as follows:

```
service getFile( path ) {
   if( fs.lstat( path ) ) {
      var head = { startLine: "HTTP/1.1 200 ok",
                   contentType: "text/plain" };
      return hop.HTTPResponseFile( path, head );
   } else {
      var head = { startLine: "HTTP/1.1 404 no" };
      return hop.HTTPResponseString( path, head );
   }
}
```

`HTTPResponseFile` enables Hop.js to deliver static files as efficiently as most modern web servers. It is internally used to deliver all system files to clients (default CSS styles, client-side runtime environment, ...).

Convenience: the `HTTPResponseHop` class is used to deliver JavaScript serialized objects. With such values, the server extracts the payload of the response, serializes it and transmits the encoded characters to the client. The `HTTPResponseXML` class is used to deliver HTML or more generally XML documents. As HTML values are represented by data structures forming abstract syntax trees (ASTs) on the server (see Section 4), they first need to be compiled into actual textual HTML, before being delivered to a client. The `HTTPResponseXML` object enables this compilation to be differed to the very moment where the response is send to the client, avoid temporary buffers and string constructions.

[2] For simplicity, it is assumed that no race condition with potential file removal can happen in this example.

3.3 Asynchronous Responses

Mainstream web servers are able to answer many requests simultaneously. This can be obtained using two different programming paradigms: concurrency by means of threads or processes and non-blocking asynchronous operations. As JavaScript is a sequential language, it relies on the latter but Hop.js extends it with ad-hoc constructions that support the former. With Hop.js the two paradigms can be mixed. JavaScript service executions are sequential and rely on non-blocking IO but while a service computes, the server is free to answer other requests (those that do not involve JavaScript executions) or to prepare the data structure of the next service to be executed, or to compile the final result of a previous service invocation for the client delivery (for instance a `HTTPResponseXML` object).

As presented before, services are synchronous: they produce values that are converted into response objects that are in turn used by the server main loop to respond to client requests. But what happens if a response can only be computed asynchronously? For instance, the previous examples have used the synchronous version of the IO functions (`lstat`, `readFileSync`, `readdirSync`). What about using their asynchronous counterparts? To let services use asynchronous APIs, Hop.js supports another kind of responses: the asynchronous responses. It enables Hop.js services to postpone the delivery of the client responses, without blocking the JavaScript execution. Asynchronous responses rely on ECMAScript 6 *promises* that are a recent addition to the language and that are used to handle deferred and asynchronous computations. A promise represents an operation that hasn't completed yet, but is expected in the future. When a service returns a promise, the Hop.js runtime system keeps the underlying system socket open for a postponed use. The actual response is only sent when the promise resolves or rejects. Let us illustrate how to use these responses on a concrete example.

The Node.js `readFile` function of the `fs` module, implements an asynchronous file read. It accepts as parameters a file name and a callback function to be called when the read completes. Asynchronous functions such as `readFile` can be used inside services, provided they are wrapped into promises as in the following example:

```
1 var fs = require( "fs" );
2
3 service getFile( path ) {
4    return new Promise( function( resolve, reject ) {
5       fs.readFile( path, function( err, d ) {
6          if( err ) {
7             reject( 'Cannot open ' + path );
8          } else {
9             resolve( <pre>${d}</pre> );
10         } );
11   } );
12 }
```

When the `resolve` argument of the promise executor (line 5) is invoked, the Hop.js runtime system converts the resolved value into a response object as described in Section 3.2 and sends that value to the client. In our example, it is called line 5, when the asynchronous `readFile` completes.

Using promises as service return values bridges the gap between the asynchronous world of Node.js and the synchronous world of the Hop.js web server loop.

3.4 Web Workers

Quoting the Mozilla Developer Network[3], *"Web Workers provide a simple means for web content to run scripts in background threads. Once created, a worker can send messages to the spawning task by posting messages to an event handler specified by the creator. However, they work within a global context different from the [global scope]... The Worker interface spawns real OS-level threads, and concurrency can cause interesting effects in your code if you aren't careful. However, in the case of web workers, the carefully controlled communication points with other threads means that it's actually very hard to cause concurrency problems. There's no access to non-thread safe components or the DOM and you have to pass specific data in and out of a thread through serialized objects. So you have to work really hard to cause problems in your code"*. Conceptually Web workers are more like OS processes than like OS threads as they do not share memory heaps.

Hop.js supports web workers as full-fledged execution entities. They all execute in parallel. They are fully integrated to the execution environment. In particular, services are associated to workers, so several Hop.js services may execute concurrently and simultaneously. During its routing stage, the Hop.js server not only selects the proper service that handles a request. It also selects the worker with which that service is associated with.

Web workers can team up with asynchronous responses to implement some sort of load balancing amongst workers. The `resolve` function of an asynchronous response can be exchanged with a worker using the `postMessage` method. For instance, let us consider the following module which implements a time consuming task:

```
1 // longlasting.js (worker thread)
2 function longTask( arg ) { ... }
3
4 onmessage = function( e ) {
5    var reply = e.data.resolve;
6    reply( longTask( e.data.val ) );
7 }
```

Slave workers can be spawned to execute these tasks in background and to respond when done:

```
8 // workers pool
9 var workers = [ 0, 1, 2, 3 ].map( function() {
10   return new Worker( "./longlasting.js" );
11 } );
12 var windex = -1;
13
14 // main worker
15 service longLasting( arg ) {
16   return Promise( function( resolve, reject ) {
17     if( ++windex > workers.length ) windex = 0;
18     workers[ windex ]
19       .postMessage( { resolve: resolve, val: arg } );
20   } );
21 }
```

In our example, the main worker implements a service that creates promises to delegate the actual response to the background worker (line 18). When the background worker completes, it sends its result using the replier it has received from the main thread (line 6).

[3] `https://developer.mozilla.org`

4. Client-Side Computation

Hop.js is meant to run web applications whose server side parts are distributed across multiple Hop.js processes and whose client-side parts are executed by web browsers. As such, generating and manipulating HTML elements is an essential task for many Hop.js applications. HopScript supports HTML as a primordial value as strings or numbers. This is explained in this section.

4.1 HTML Values

HopScript treats HTML indifferently on the server-side and on the client-side of the program. HTML fragments are always represented by values that are trees of HTML elements. These values are created and manipulated using a classical HTML representation and a DOM level 2.0 compliant API[4].

HopScript proposes an extension of the regular JavaScript function call syntax that makes the creation of HTML values identical to plain HTML. Any HopScript functions can be called with the conventional JavaScript function call and also with an equivalent XML-like syntax. This makes the DOM API more comfortable to use. Using the conventional JavaScript syntax, the API might be used as:

```
var table = TABLE({class: "klass", id: "tbl24"},
  TR({},
     TH({}, "value"), TD({}, "99")))
```

Using the XML-based syntax, it might be used as:

```
var table = <TABLE id="tbl24" class="klass">
  <TR><TH>value</TH><TD>99</TD></TR>
</TABLE>
```

The two forms are strictly equivalent but the latter is more familiar to web developers and designers as it does not deviate from plain HTML, it is compatible with interface builders, and it is arguably more readable.

In the example "`klass`", "`tbl24`", "`99`", and "`value`" are string literals. In real programs, it is frequent that dynamic values need to be to *injected* into the generated HTML documents. This is the role of the "`${`" syntactic mark, which follows ECMAScript 6 templates string[5]. It inserts dynamic values into HTML fragments, as in the following example where the constant `99` has been replaced with the variable `val`:

```
var table = <TABLE id="tbl24" class="klass">
  <TR><TH>value</TH><TD>${val}</TD></TR>
</TABLE>
```

Tag elements are flattened: they can be HTML elements or nested arrays of HTML elements. This allows HTML constructors and JavaScript higher order operators to be combined together. For instance, this flattening allows an HTML table to be constructed from a JavaScript array using the `map` iterator as in:

```
var arr = new Array( ... );
var table = <TABLE id="tbl24" class="klass">
   ${arr.map( el => <TR><TD>${el}</TD></TR> )}
</TABLE>
```

The HopScript tag syntax can be used with any function. It can be used in client-side source code as well as server-side source

[4] `http://www.w3.org/DOM`

[5] ECMA6Script template strings are defined at: `http://www.ecma-international.org/ecma-262/6.0/#12-2-9`.

code, with the same semantics. It can also be used to invoke methods as in the following example:

```
// The module fib.js exports the tag FIG
var f = require( "./fig.js" );

function htmlPage( photo, legend ) {
   return <HTML>
      <f.FIG><IMG src=${photo}/>${legend}</f.FIG>
    </HTML>
}
```

4.2 Multitier HTML

HTML elements are *active* when they hold *listeners*, which are JavaScript client-side functions triggered after users interactions with the HTML graphical interface. In HTML, listeners are introduced as JavaScript fragments, which are attributes values. For instance, making a table cell active can be done with:

```
<TD onclick='alert( event.button )'>99</TD>
```

HopScript allows listeners, and more generally, any client-side expression, to be added to HTML elements wherever they are created. These client-side expressions are introduced by a new syntactic form: the ~-expressions, which evaluate to values that can be serialized and sent over the network to clients. ~-Expressions can be used on the two tiers of the application. Example:

```
<TD onclick=~{alert( event.button )}>99</TD>
```

With this form, client-side expressions are not opaque strings of characters but an expression of the language whose correctness is ensured by compilation. This improves over traditional HTML string representations.

Values can be inserted inside ~-expressions using the already presented '${' mark. This insertion takes place when the HTML element is created. Example:

```
function mkTable( msg ) {
   return <TABLE class="example" id="tbl45">
     <TR>
       <TH>value</TH>
       <TD onclick=~{alert( ${msg} )}>99</TD>
     </TR>
   </TABLE>;
}
```

The function `mkTable` creates HTML tables whose cell is reactive. The variable `msg` is used at the moment where the table is constructed, to compute the client-side expression that will be executed when, later on, a user will click on the table cell. Note that the function `mkTable` is multitier as it can be used in server-code and client-code indifferently.

~-Expressions can also be used as HTML elements. In that case, they get automatically compiled into `<script>` elements. Example:

```
<html>
  ~{ var un = ${process.uname()} }
  <button onclick=~{ alert( un ) }>
    uname
  </button>
</html>
```

4.3 Full Staging Programming

In the previous section ~{ and ${ were exclusively used to let the server-side program generate the client-side program. This was a simplification as actually the two forms generalize to a full fledged tower of recursive generator/generated elements: a ~{ expression can be nested inside another ~{ expression. That is, a client-side program can generate other client-side programs. In that case, the ~{ and ${ forms denote the generated client-side and generator client-side. The ~{ mark enters a new level of client program; a ${ mark escapes exactly one level. This is illustrated by the following example:

```
1  service tower() {
2    return <html>
3      ~{
4         function clicked (msg) {
5            var but=<button onclick=
6                      ~{clicked( ${msg + "+"} )}>
7              ${msg}
8            </button>;
9            document.body.appendChild( but );
10        }
11     }
12     <button onclick=~{clicked( "click me also" )}>
13       click me
14     </button>
15   </html>
16 }
```

The `tower` service creates one button, which when clicked creates one button, which when clicked creates one button, etc. Each nesting level of generated button adds a character to the message of its generator. Clicking again on a button already inserted starts a new thread of button insertions, as illustrated in Figure 3.

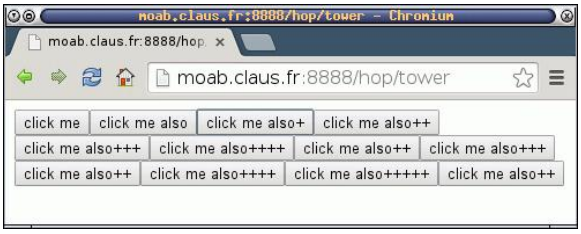


Figure 3. The Chromium browser executing the `tower` HopScript function.

The ~{ mark line 6 is nested in the client-side expression starting line 3. The ${ mark line 6, escapes one level of client-side expression. Hence the concatenation `msg + "+"` is executed when the next button is created, before being inserted in the page.

The HopScript multi-staging [27] is not an exotic feature. It is required to unify the server-side programming language and the client-side programming language. It makes HopScript a uniform and global programming language for the servers and the clients. Without this feature, all codes containing a ~{ construct would be *tainted* as server-side programs and would not be usable on the client-side. This would typically prevent the creation of multitier widgets libraries that can be used indifferently on the server-code and on the client-code.

4.4 HTML Assimilation

HTML is part of the full HopScript syntax. HTML texts are then regular HopScript programs. As such, they can then be loaded using the module `require` library function. For instance:

```
var doc = require('http://slashdot.org/')
```
loads the Slashdot page of the day, parses the HTML document, creates a DOM data structure, and binds it to the variable `doc`. This document can then be manipulated using the standard DOM api. For instance, creating a table of contents can be done with:

```
function toc( doc ) {
   return <OL>
     ${doc.getElementsByTagName( "h2" ).map( LI )}
   </OL>;
}
```

This `toc` function can be either called from server-side or client-side programs. It shows the benefit of making HTML a builtin construct of the language as it shows how simple it is to re-use and manipulate many third party resources.

5. Machine to Machine Communications

Machine-to-machine communication is builtin in Hop.js and it has several different facets. The prominent one is the concept of services that enable clients-to-server communications, as presented in Section 3, but also server-to-server communications. This is presented in this section as well as the other constructs that let two machines communicate.

5.1 Remote Services

A Hop.js client component, *i.e.*, a Hop.js program running inside a web browser, can only invoke the services of its origin server. A server may invoke the services of other servers. For that, three new Hop.js ingredients are needed. First, the HopScript runtime environment provides the `hop.Server` class that is used to designate remote machines. Second, services can be *imported*. It enables a Hop.js program to invoke remote services without disposing of their implementations. Third, when a service is invoked as a method of a server object, the constructed frame designed that remote machine. Example:

```
1 var mysrv = new hop.Server( "cloud.net", 8080 );
2 mysrv.translate = service translate();
3
4 mysrv.translate( "Hello world!", "en|fr" )
5    .post( console.log );
```

Line 1 is the creation of remote server object. This denotes a remote server, named `cloud.net`, listening connections on its port `8080`. Line 2 imports the service `translate`. This creates a fake service that can only be used to build service frames (see Section 3.1.1). Line 2 also binds it to the `translate` property of the server. Line 4 is the service invocation. Service calls follow the syntactic convention of the regular JavaScript function calls, which distinguishes function calls from method calls. When a service is called in function position, the frame it builds designates the local host. When a service is called in the syntactic position of an attribute (here, the attribute `translate` of the `mysrv` object), the frame it builds designates the server described by the receiver of the method, here the `mysrv` server. Using the JavaScript `apply`, the expression line 4 could be re-written as:

```
4 translate.apply( mysrv, [ "Hello world!", "en|fr" ] )
5    .post( console.log );
```

5.2 Third Party Web Services

Third party services are an important component of many modern applications. HopScript makes them almost as easy to use as native services. Remote services are created by the `webService` constructor. Once created, they are only distinguished from HopScript services by their arguments passing: third party services can only receive strings. On the other hand, they can deliver results that are automatically unmarshalled by Hop.js, by simply providing an adequate mime type with their responses.

For the sake of the example, here is a HopScript program using a third party natural translation site:

```
1 var mymemory =
2     new webService( "http://mymemory.net/api/get" );
3
4 function translateText( text, lang = "en|fr" ) {
5    var o = mymemory( { q: text, langpair: lang } )
6       .postSync();
7    if( o.responseStatus === 200 ) {
8       return o.responseData.translatedText;
9    }
10 }
```

Line 2 is the creation of the third party service. Line 6 is the call. This service includes in its HTTP response the `content-type: text/json` header. This tells Hop.js to parse the result accordingly and to create a fresh JavaScript object which is bound to the variable `o` line 5.

5.3 Broadcasting

Broadcasting is the second ingredient of the machine-to-machine communication tool set. It abstracts WebSockets to let Hop.js servers send events to connected clients (either web browsers or Hop.js client processes). Connections originate from the client to the server, so broadcast can be used even in the asymmetric web topology where clients often lie behind a NAT router or firewall and would not accept an incoming connection from a remote server (forbidding the remote server to invoke services running on the client process).

The expression `hop.broadcast( event, value )` generates an event named `event` with payload `value`. The event is broadcast over the network to all registered clients. Note that private channels may be established by peeking private random event strings only shared by the participants. The argument `event` is a string, `value` can be any serializable object, including JavaScript objects, Hop.js services, and XML elements. Clients register to specific broadcast events with the `addEventListener` method of their builtin `server` object. Example:

```
function updateGUI( event ) { ... }
server.addEventListener( 'refreshScore', updateGUI );
```

Hop.js processes can also receive broadcast events by creating remote server objects on which they attach listeners:

```
mysrv.addEventListener( 'refreshScore', refresh );
```

Each time the server designated by the `mysrv` object will broadcast the event `refreshScore`, the current Hop.js process will be notified and will react by calling its local listener (here the function `refresh`).

A server and a client may establish a private broadcast communication channel using broadcast by agreeing on a random key only known by the two parties. This can be implemented in a straightforward manner as:

```
service ping() {
   var privkey = randomPrivateKeyGenerator();
   // broadcast periodically
   setInterval( function() {
      hop.broadcast( privkey, process.title );
   }, 1000 );


   return <html>
     ~{ server.addEventListener( $privkey, handler ); }
     ...
   </html>
}
```

6. Debugging HopScript Programs

As already presented in Sections 3 and 4, multitier programming enables a unique and global view of the program being implemented. It also exposes a global view of the execution. The server-side and the client-side executions are under the control of a unique global runtime environment. This gives the opportunity to the runtime environment to report errors in the global execution context [24]. This is illustrated by the following program:

```
1 service svc() {
2    return <html>
3      ~{ function clientAction() {
4         ${serverAction}({x: -23}).post() } }
5      <button onclick=~{clientAction()}>
6        click me
7      </button>
8    </html>
9 }
10
11 service serverAction( arg ) {
12    arg.call( 3 );
13 }
```

Line 12 erroneously assumes a method `call` for the argument `arg`. When a user clicks on the "click me" button of this web page, the `clientAction` function is called, which will in turn remotely invoke the `serverAction` service, that will raise an error. HopScript will report it as follows (slightly embellished for accommodating the paper layout):

```
File "bug1.js", line 12, character 3:
|   arg.call( 3 );
|   ^
TypeError: call1: not a function "undefined"
    1. serverAction, bug1.js:11
Service trace:
    1. ~serverAction.post(...)
    2. ~clientAction, /hop/svc:3
    3. ~button#G0.onclick, bug1.js:5
    4. $<html>, bug1.js:2
    5. $svc, bug1.js:
```

The stack traces presented by Hop.js contain informations about the two ends of the execution. Not being oblivious of the other side of the execution makes debugging arguably easier.

7. Implementation

The whole Hop.js system including the various compilers and the web server is implemented in a multi-threaded variant of the Scheme programming language [15]. The HopScript code is either compiled to JavaScript for client-side execution, or compiled to Scheme code for server-side execution. This section presents the main components of the Hop.js implementation in order to bring a different perspective to its design.

7.1 The Web Server

The builtin web server is a central component of the Hop.js runtime environment as it handles the low level communications between server-side code and client-side code. More precisely, it handles the incoming connections, selects the appropriate HopScript services, invokes the appropriate compiler to generate client responses on the fly, and sends the responses to the clients.

Pipelining is a well known architecture for implementing fast web servers [7, 29, 30]. It consists of separating in different stages the main treatments applied to incoming HTTP messages. Figure 4 shows the Hop.js pipeline. The first stage named "**accept**" waits for incoming network connections. It allocates the object representing the client socket. The stage "**request**" parses the whole HTTP incoming message, including the URL and the HTTP header. It constructs a Hop object representing the request. The stage "**response**" constructs the response to be sent to the client. For that, it unmarshals service arguments and it routes the request to the proper request handler. The stage "**reply**" compiles and ships the response obtained from the previous stage to the client. The compilation of the HTML DOM into actual HTML textual representation takes place during that stage, as the client-side program generation. In the case of HTTP/1.1 requests, the same socket can be re-used by the client to make a second request, this improves performance dramatically as accepting a connection and creating a socket object are slow operations [19]. When a socket is reused the pipeline is re-entered in the "**request**" stage .

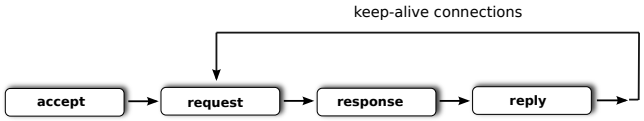


Figure 4. Internal pipeline of the Hop web server.

This architecture supports parallelism as several pipelines can run concurrently. Hop.js runs 20 of them in its default configuration, each being executed inside a dedicated lightweight thread as shown in Figure 5.

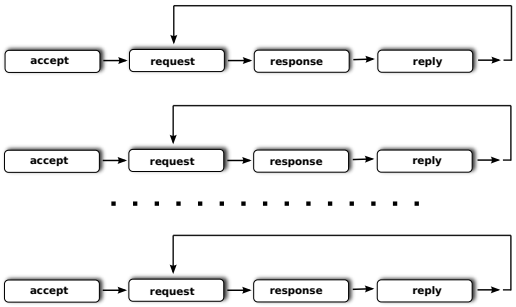


Figure 5. The parallel pipelines of the Hop web server.

The benefits of the pipelined architecture have been studied long ago [3] but it appears to be particularly well suited to Hop.js. In the pipeline, only the "**response**" stage involves JavaScript executions. All the other stages are handled by the Hop.js runtime environment. Hence, all but the "**response**" stage can be executed in parallel. In

practice, the server can be simultaneously accepting new connections, parsing incoming HTTP headers, unmarshalling services arguments, delivering files, generating HTML, compiling client-side programs, and executing *one* JavaScript program for preparing a response in parallel. The pipelining architecture of the web server enables Hop.js to take benefit of multi-core architecture, even with JavaScript being intrinsically sequential.

The pipelined architecture enables parallel treatments of HTTP connections, without requiring parallel executions of JavaScript programs. However, as all the dynamic contents Hop.js delivers involve JavaScript executions, a slow or long lasting JavaScript evaluation would downgrade significantly the performance of the whole server. For these particular situations, Hop.js supports a restricted form of parallelism for JavaScript in addition to the asynchronous coding style of Node.js.

Hop.js workers (see Section 3.4) execute in parallel in the "**response**" stage, as shown in Figure 6.

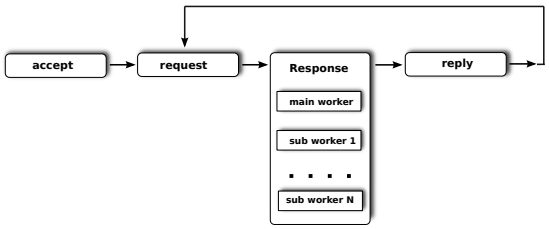


Figure 6. The JavaScript workers of the reply stage.

Hop.js services are associated to workers, which enables the HTTP routing to also select the thread that handles a request. After the request object has been created in the "**request**" stage, the appropriate worker is selected by the Hop.js runtime system for elaborating the response.

Hop.js workers are implemented as OS lightweight threads. The memory isolation is implemented by compilation. Each worker receives a fresh copy of the JavaScript global object, which makes memory sharing impossible. However, for some particular objects, those that are under the exclusive control of the Hop.js runtime system, some sharing is permitted. This is used to exchange promises resolvers and rejecters from one worker to another. As these two functions are not regular JavaScript functions (they are created by the runtime system when a promise object is created) they can benefit from a specific implementation that enables locking and sharing.

Asynchronous responses (see Section 3.3), that can team up with workers, change the normal control flow of the pipeline as shown in Figure 7. When the **response** stage delivers an asynchronous response, *i.e.*, a JavaScript promise object, the HTTP socket is kept open and the **reply** stage is delayed until the promise resolves or rejects.

Figure 7. Internal pipeline of the Hop web server with asynchronous responses.

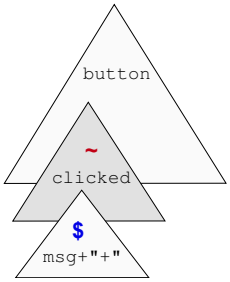
7.2 JavaScript Compilation

The HopScript compiler is multi-backend. It produces Scheme code for the servers and JavaScript code for the clients. On the server, the Scheme code is interpreted in debug mode, otherwise compiled to native code. The same AST structure is used for both targets but some compilation stages are skipped when generating JavaScript client code, typically, all the optimizations that aim at improving the speed of the generated Scheme code.

During the parsing of the source program, the XML syntax is desugared to be represented as HTML constructor calls. The `~{` and `${` expressions are represented by two special AST nodes. The AST built for the expression

```
<button onclick=~{clicked( ${msg + "+"} )}>
   ${msg}
</button>
```

has the following shape:



Let us illustrate the Hop.js compilation[6] with the `tower` example of Section 4. First, JavaScript modules are compiled as Scheme functions that take two parameters, the global object (`%this`), the global scope (`%scope`). Node.js distinguishes between the global scope in which global variables are defined, and the global object in which builtin values and prototypes are defined. When compiled to Scheme, all functions take an explicit `this` parameter, which is `undefined` when the function is not called as a method. The function `js-create-service` takes as parameters, the function implementing its body, its name, for the URL construction, and the current worker, for the web server routing.

```
1  (define (module %this %scope)
2     (define @js-expr1 ...)
8     (define @js-expr2 ...)
10    (define (@tower this) ...)
22    (define tower
23       (js-create-service %this
24          (js-make-function %this @tower 0)
25          "tower"
26          (js-current-worker))))
```

The function `@tower` implements the service. It calls the `HTML` global function with two values: a client-side expression (line 12) and an HTML button (line 14).

```
10 (define (@tower this)
11    (js-call %this (js-get %scope 'HTML)
12       (instantiate::JsTilde
13          (%js-expression @js-expr1))
14       (js-call %this (js-get %scope 'BUTTON)
15          (with-access::JsGlobalObject %this (__proto__)
16             (instantiate::JsObject
17                (elements
18                   (instantiate::JsTilde
19                      (%js-expression @js-expr2)))
20                (__proto__ __proto__)))
21          "click me")))
```

The client side expressions are compiled as follows:

[6] The compilation results presented here a manually modified to fit the constraints of the paper.

```
(define @js-expr1
   "function clicked(msg){
    var but=BUTTON(
       {'onclick':function(event){clicked(msg + '+')}},
       msg);
    document.body.appendChild(but)")
(define @js-expr2
   "clicked('click me also')")
```

The variable `@js-expr1` holds a JavaScript statement as it results of the compilation of a client-side block (see Section 4.3). The variable `@js-expr2` holds an expression as it results of the compilation of a node attribute.

When `tower` is invoked in the **response** pipeline stage, it calls the `@tower` function that builds an abstract syntax tree of the HTML document to be responded to the client. That AST is passed to the **reply** stage that compiles it into actual HTML.

7.3 HTML Compilation

The compilation of HTML is mode-dependent. The initial mode is the "node" mode, in which the compilation is mostly a straightforward pretty-print of the HTML AST built in the **response** stage. The version of HTML to be generated (4.01, 5.0, XHTML, ...) is specified by a compiler flag contained in response objects. When HTML nodes attributes are compiled, the compiler switches to "attribute" mode. In that mode, HTML objects are compiled as references to client-side DOM nodes. This is illustrated by the following compilation example. Let us assume the following expression:

```
1 service foo() {
2    var el = <div/>
3    return <html>
4      ${el}
5      <input onclick=~{${el}.innerHTML=this.value }/>
6    </html>
7 }
```

The `el` occurrence (line 4) is compiled as an HTML tag element (line 2 below). The second occurrence (line 5) is inside a TAG attribute. It is then compiled as a reference to a DOM node (line 3 below). Note that Hop.js adds identifiers automatically to all HTML nodes, unless the program provides one already.

```
1 <html>
2   <div id="g4503"></div>
3   <input onclick="document.getElementById('g4503')
4                     .innerHTML=this.value"/>
5 </html>
```

The HTML compilation normally takes place in the pipeline "**reply**" stage. The characters produced by the HTML compiler are then directly written to the network socket and there is no overhead associated with storing first the compilation result on a disk or into a string. We have observed that in general, this delivers sufficient performance for most applications. However, in some situations, it might be interesting to implement an application cache for avoiding HTML recompilation. This is left to the charge of the programmer that can deploy the strategy that best fits his needs.

8. Related Work

Starting in the late 90's we have witnessed an intensive effort in modeling and improving the programming model of web 1.0 applications that was mainly relying on HTML forms and explicit workflows between pages. This has been initially studied by C. Queinnec, that has observed that most form based web applications have to deal with continuations. In his early publication [21] he has shown that a browser is a device that can call continuations multiply and simultaneously. Hence, he has concluded that an operator for capturing and restoring continuations is a natural tool of choice for implementing web pages. This point has been deeply developed and thoroughly studied by the PLT Scheme team in various publications [14, 17]. In the same vain of research, continuations have inspired the design of several languages such as Seaside [11] and Links [8]. Nowadays, the advent of web *single page application* has made form-based applications less frequent and continuations less popular. Hop.js, which focuses on interactive GUIs and interactive communications between distributed participants does not support them.

Functional reactive programming has many incarnations on the web including Flapjax [18], Elm [9], Ur/Web [5, 6], multi-tier FRP programming in JS-Scala [22], React, React.js, StratifiedJS, etc. With these languages, the visible page is implemented as a function over some streams of values that evolve over time or after user interactions. As the streams evolve, the GUI is automatically updated. This alleviates the programmers from explicit, tedious, and error prone DOM manipulations. Hop.js currently support no such things and GUI updates have to be programmed explicitly. We are considering extending HopScript to support *lightweight reactive programming*. Instead of considering a radical change in the programming model, we are opting for a mere extension, probably provided by the means of a dedicated library made of stream constructors, explicit reactors inserted in the DOM, and a synchronous DSL.

Most web multitier programming languages including Ocsigen [1, 2, 28], Ur/Web, and iTask3 [10], are also *tierless* languages [20]. They fade out the boundary between the client and the server sides of the programs. Typically a mere annotation distinguished between both ends. This is not the approach followed by Hop.js that, on the contrary, promotes the tiers as first class values of the language. With Hop.js, a client-side program is a server-side value, explicitly computed by a server-side program. In that respect, Hop.js does not deviate from mainstream web server programming that explicitly creates HTML client-side pages containing JavaScript client-side programs.

Hop.js being based on JavaScript it shares many similarities with all the platforms that use this language, especially Node.js. However although Hop.js is compatible with Node.js, *i.e.,* all Node.js programs can be executed in Hop.js[7], the two systems propose different models for programming web applications. The most notable difference is the Node.js agnostic view of the web. Node.js only provides a low level system API that supports sockets, HTTP, non blocking IOs, some parsing facilities, and an event based loop. Node.js is a runtime environment for programming server-side applications only. Of course, it can be completed by frameworks[8] that raise the level of abstraction but it remains that Node.js is ignorant of the most important aspects that are addressed by Hop.js: the asymmetric client/server architecture of all web applications and HTML.

The support of hardware parallelism also makes Hop.js different from Node.js. As JavaScript is a sequential language, and as the Node.js runtime environment does not support a builtin native loop for handling HTTP requests, the only means Node.js programmers have at their disposal for handling simultaneous requests is asynchronous programming. This model scales naturally well when the traffic load increases as handling more simultaneous requests only

[7] Packages that depend on native libraries need special attention.

[8] For instance the Express framework `http://expressjs.com/`.

involves only allocating more closures that are automatically and efficiently reclaimed when the requests are responded. It has also a down side: it makes program development and maintenance difficult. Hop.js addresses this problem by supporting native parallelism that collaborates with JavaScript sequential codes without being placed under the responsibility of the application programmer.

In PHP and most popular web programming environments, HTML elements are treated as mere strings. With these systems, HTML documents are generated on the server by concatenating strings of characters. This is error prone as it is generally difficult to test the correctness of this dynamically created strings. Using such a textual representation has several drawbacks: the lack of dedicated syntactic constructs makes it inconvenient to program with as the structure of the program diverges from the structure of the elaborated HTML document, it is difficult for a compiler to detect syntax errors, it imposes an asymmetric view to the server-side and to the client-side, it imposes an early decision on the document charset encoding and on the HTML dialect, it exposes the server to SQL-injection attacks [26], and many others. Using a textual representation on the server and a DOM based representation on the client is also opposed to the multitier approach which aims at providing an uniform view of the data structures and execution environments.

The HTML tag extension naturally solves these problems as it is based on the genuine HTML syntax. Prior to HopScript, other systems have already opted for the same solution [4, 6, 8, 13]. JSX, *aka* ReactiveJS seems particularly similar to Hop.js but behind the syntactic resemblance, lies an important difference: JSX does not support anything close to the HopScript Multitier HTML. The consequence is that server-side code and client-side code are asymmetric with JSX, while they are symmetric with HopScript, and then usable on the server-side and the client-side of the application. This is illustrated by the following example. Let us consider an expression creating a reactive HTML `div`. In JSX it might look like:

```
<div onClick={function() { alert( ... ) }}>
   {body}
</div>
```

In HopScript the same code should be written:

```
<div onclick=~{ alert( ... ) }>
   ${body}
</div>
```

The difference seems marginal and purely syntactic (an explicit function in JSX and a `~{` mark in HopScript; an extra `$` sign for including computed values as the body of the `div`). It is not. JSX represents active attribute (`onclick`, `onkeypress`, ...) as functions and it does not use staging for elaborating HTML. In JSX, there is only one execution context: the client-side program. This introduces a discrepancy between the HTML code of the server-side program and the XML syntax the client-side program. In standard HTML, the function construction of the active attributes are implicit, while they are explicit in JSX. This implies that the server-side HTML and client-side HTML cannot be used interchangeably, while in HopScript, they are.

At last, as JSX is not a staged language the only way to create function bodies dynamically is to rely on `eval`, with all the known consequences in term of performance and security. Without `eval` it is impossible to "inject" server sides values in the active values. This precisely what HopScript `~{` and `${` do.

Modern web browsers also start supporting custom tags declarations[9]. The client-side API they support is shaped by the asynchronous nature of the tag declarations: a web page starts loading, possibly using custom tags, before the JavaScript engine evaluates the custom definitions. Handling this difficulty makes the HTML custom tags elements API more complex than the simple server-side mapping from tags to functions used by HopScript.

StratifiedJS (the Conductance runtime environment) is another approach to multitier programming in JavaScript. It does not rely on a multitier DOM approach as HTML values are constructed using what is now known as template literals. Conductance aims at eliminating all syntactic differences between using local functions or variables and remote ones, as other languages such as Ocsigen, Links, or Ur/Web do. Hop.js relies on different option as the language supports no means for accessing remote variables or functions. Hop.js only supports services that are remotely invoked using dedicated method calls. The Hop.js design decision is motivated by the different nature of calling local and remote functions. The network traversal of the latter makes the speed of the operation uncomparable with the speed of local function calls, and a different semantics as the classical call-by-value semantics has to be abandoned in favor of a call-by-copy semantics.

9. Conclusion and Perspectives

Hop.js is a new platform for web applications, potentially involving interconnected servers. The server-side execution is compatible with Node.js. Programmers then benefit from numerous existing libraries and applications. Hop.js also introduces distinctive programming features that are expressed in the HopScript programming language, a multitier extension of JavaScript. The Hop.js runtime embeds a multi-backends HopScript compiler.

The HopScript language extends JavaScript to consistently define the server and client part of a web application. HopScript supports syntactic forms that help creating HTML elements. It supports services that enable function calls over HTTP. Being higher level than traditional Ajax programming, Hop.js services avoid the burden and pitfalls of URL management and explicit data marshalling. They combine the benefits of a high level RPC mechanism and low level HTTP compatibility.

Hop.js supports server-side and client-side parallelism. On the server, it first relies on its built-in pipelining architecture that automatically decodes HTTP requests in parallel. It also relies on server-side web workers that programs may explicitly launch to perform background tasks (functions and services). Each worker runs its own system thread. The service invocation and execution API fully integrates with the JavaScript execution flow, allowing synchronous and asynchronous operations on both client and server processes. The asynchronous response API can be combined with the worker API, allowing processing and asynchronous service responses to be delegated between workers. On the browser client-side parallelism relies on standard web workers.

Although Hop.js can be used to develop traditional web servers, it is particularly adapted to the development of web applications embedded into devices, where the server and client part of the application are intimately interoperating with each other. The programing model of Hop.js fosters the joint specification of server and client code, and allows the rapid development of web user interfaces, on the client, controlling the execution of the distributed application. By defining a single data model, providing functions that can run indifferently on both sides, and almost forgetting about client-server protocols, Hop.js seems well suited for agile development of web applications for this class of applications.

[9] See `https://w3c.github.io/webcomponents/spec/custom/`

As an example, Hop.js has already been successfully used as the core framework to develop embedded and cloud applications for connected robots and IoT devices. In the context of a European industrial collaborative project, it has been used by various categories of programmers (mostly undergraduate internships, robotic experts, and professional engineers familiar with web development techniques) to build complex distributed applications, where various sort of digital equipments (computers, robots, small devices) communicate with each other, discover themselves, and collaborate. In all cases we have observed an easy adoption from everyone. The tons of JavaScript resources and examples available on the web helped internship students to rapidly become productive. Robotic experts were instantly able to start implementing Hop.js applications. Web experts seemed to feel at home with Hop.js as it let them build working applications with Hop.js core features and then extend them with existing JavaScript third party modules, typically npm modules.

In the future, we foresee different axes of research for Hop.js. First, there is room for performance improvements of the JavaScript compilation. We are willing to explore situations where JIT compilation is not necessarily adapted, for instance, when the execution environment only offers a very limited memory capacity. Another research direction we will focus on is orchestration and synchronous/asynchronous programming. Programming a web application involves a lot of events handling. Until recently, this was mostly programmed with callbacks. Recent proposals combine JavaScript programming and dataflow programming. They constitute an interesting alternative to callback programming but we think that there is still room for alternative proposals that not only simplify event handling but that also integrate harmoniously with traditional JavaScript programming techniques.

Acknowledgments

This work has been supported by the European FP7 RAPP project (FP7-ICT-2013-10, 610947) and the French FUI UCF 8980 project. Special thanks are due to Christian Queinnec for his early comments on this paper and to the ICFP program committee members that have helped improving the paper.

References

[1] V. Balat, P. Chambart, and G. Henry. 2012. Client-server Web applications with Ocsigen. In *Proceedings of the WWW'2012 conference*. Lyon, France.

[2] V. Balat, J. Vouillon, and B. Yakobowski. 2009. Experience report: ocsigen, a web programming framework. In *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*. 311–316. DOI:http://dx.doi.org/10.1145/1596550.1596595

[3] G. Banga and P. Druschel. 1997. Measuring the Capacity of a Web Server. In *USENIX Symposium on Internet Technologies and Systems*. http://citeseer.ist.psu.edu/banga97measuring.html

[4] H. Binsztok, A. Koprowski, and I. Swarczewskaja. 2013. *Opa: Up and Running*. O'Reilly Media.

[5] A. Chlipala. 2015a. An Optimizing Compiler for a Purely Functional Web-Application Language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. 10–21. DOI:http://dx.doi.org/10.1145/2784731.2784741

[6] A. Chlipala. 2015b. Ur/Web: A Simple Model for Programming the Web. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 153–165. DOI:http://dx.doi.org/10.1145/2676726.2677004

[7] G. Choi, J-H. Kim, D. Ersoz, and C. Das. 2005. A Multi-Threaded PIPELINED Web Server Architecture for SMP/SoC Machines. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*. ACM, New York, NY, USA, 730–739. http://doi.acm.org/10.1145/1060745.1060851

[8] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. 2006. Links: Web Programming Without Tiers. In *5th International Symposium on Formal Methods for Components and Objects (FMCO)*. Amsterdam, The Netherlands, 266–296.

[9] E. Czaplicki and S. Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 411–422. DOI:http://dx.doi.org/10.1145/2491956.2462161

[10] L. Domoszlai and R. Plasmeijer. 2015. Tasklets: Client-Side Evaluation for iTask3. In *Central European Functional Programming School*, V. Zs'ok, Z. Horv'ath, and L. Csat'o (Eds.). Lecture Notes in Computer Science, Vol. 8606. Springer International Publishing, 428–445. DOI:http://dx.doi.org/10.1007/978-3-319-15940-9_11

[11] S. Ducasse, A. Lienhard, and L. Renggli. 2007. Seaside: A Flexible Environment for Building Dynamic Web Applications. *IEEE Software* 24, 5 (2007), 56–63. DOI:http://dx.doi.org/10.1109/MS.2007.144

[12] ECMA International. 2011. *Standard ECMA-262 - ECMAScript Language Specification* (5.1 ed.). http://www.ecma-international.org/publications/standards/Ecma-262.htm

[13] Facebook. 2015. JSX. https://facebook.github.io/jsx/. (2015). https://facebook.github.io/jsx/

[14] P. Graunke, R. Findler, S. Krishnamurthi, and M. Felleisen. 2003. Modeling Web Interactions. In *European Symposium on Programming*. Poland.

[15] R. Kelsey, W. Clinger, and J. Rees. 1998. The Revised(5) Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation* 11, 1 (Sept. 1998). http://www-sop.inria.fr/indes/fp/Bigloo/doc/r5rs.html

[16] G. Kossakowski, N. Amin, T. Rompf, and M. Odersky. 2012. Javascript as an embedded DSL. In *Proceedings of the ECOOP'2012 conference*. Beijing, China, 409–434.

[17] J. McCarthy. 2009. Automatically RESTful Web Applications: Marking Modular Serializable continuations. In *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*. 299–310. DOI:http://dx.doi.org/10.1145/1596550.1596594

[18] L. Meyerovich, A. Guha, J. Baskin, G. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. 2009. Flapjax: A Programming Language for Ajax Applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 1–20. DOI:http://dx.doi.org/10.1145/1640089.1640091

[19] H-F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'Hommeaux, H. Lie Wium, and C. Lilley. 1997. Network Performance Effects of HTTP/1.1, CSS1, and PNG. In *Proceedings of the ACM SIGCOMM'97 conference*. Cannes, France.

[20] L. Philips, C. De Roover, T. Van Cutsem, and De Meuter. W. 2014. Towards Tierless Web Development without Tierless Languages. In *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20-24, 2014*. 69–81. DOI:http://dx.doi.org/10.1145/2661136.2661146

[21] C. Queinnec. 2000. The Influence of Browsers on Evaluators. In ACM SIGPLAN *Int'l Conference on Functional Programming (ICFP)*. Montréal, Canada, 23–33.

[22] B. Reynders, D. Devriese, and F. Piessens. 2014. Multi-Tier Functional Reactive Programming for the Web. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms,*

and Reflections on Programming & Software (Onward! 2014). ACM, New York, NY, USA, 55–68. DOI:http://dx.doi.org/10.1145/2661136.2661140

[23] J. Richard-Foy, O. Barais, and J-M. Jézéquel. 2013. Efficient high-level abstractions for web programming.. In *GPCE*, FOO (Ed.). ACM, 53–60. http://dblp.uni-trier.de/db/conf/gpce/gpce2013.html#Richard-FoyBJ13

[24] M. Serrano. 2014. A Multitier Debugger for Web Applications. In *Proceedings of the 10th WEBIST conference (WEBIST'14)*. Barcelona, Spain.

[25] M. Serrano, E. Gallesio, and F. Loitsch. 2006. HOP, a language for programming the Web 2.0. In *Proceedings of the First Dynamic Languages Symposium (DLS)*. Portland, Oregon, USA.

[26] Z. Su and G. Wassermann. 2006. The Essence of Command Injection Attacks in Web Applications. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. ACM, New York, NY, USA, 372–382. DOI:http://dx.doi.org/10.1145/1111037.1111070

[27] W. Taha. 1999. *Multi-Stage Programming: Its Theory and Applications*. Ph.D. Dissertation. Oregon Graduate Institute of Science and Technology, USA.

[28] J. Vouillon and V. Balat. 2013. From bytecode to Javascript: the Js_of_ocaml compiler. *Software: Practice and Experience* doi: 10.1002/spe.2187 (Feb. 2013). DOI:http://dx.doi.org/10.1002/spe.2187

[29] M. Welsh, D. Culler, and E. Brewer. 2001. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Symposium on Operating Systems Principles*. 230–243. http://citeseer.ist.psu.edu/welsh01seda.html

[30] N-M. Yao, M-Y. Zheng, and J-B. Ju. 2002. Pipeline: A New Architecture of High Performance Servers. *SIGOPS Oper. Syst. Rev.* 36, 4 (2002), 55–64. http://doi.acm.org/10.1145/583800.583807