Dynamic scalability for web applications

How to abstract scalability constraints from the developer
Industrial Paper

Etienne Brodu
etienne.brodu@insa-lyon.fr
IXXI – ENS Lyon
15 parvis René Descartes – BP 7000
69342 Lyon Cedex 07 FRANCE

Stéphane Frénot
stephane.frenot@insa-lyon.fr
IXXI – ENS Lyon
15 parvis René Descartes – BP 7000
69342 Lyon Cedex 07 FRANCE

Fabien Cellier
fabien.cellier@worldline.com
Worldline
Bât. Le Mirage
53 avenue Paul Krüger
CS 60195
69624 Villeurbanne Cedex

Frédéric Oblé
frederic.oble@worldline.com
Worldline
Bât. Le Mirage
53 avenue Paul Krüger
CS 60195
69624 Villeurbanne Cedex

ABSTRACT

The audience's growth of a web application is highly uncertain, it can increase and decrease in a matter of hours if not minutes. This uncertainty often leads the development team to quickly adopt disruptive and continuity-threatening shifts of technology to deal with the higher connections spikes. To avoid these shifts, we propose to abstract web applications into a high-level language, allowing a high-level code reasoning. This reasoning may allow later to provide code mobility to dynamically cope with audience growth and decrease.

Our approach expresses a web application as a network of small autonomous parts moving from one machine to another and communicating by message streams. We named these parts *fluxions*, by contraction between a flux and a function. *Fluxions* are distributed over a network of machines.

Our high-level language proposition consists of a compiler and its execution model to express a web application into a network of *fluxions* and streams. We expect that the dynamic reorganization of these independent parts in a cluster of machine can help an application to deal with its load as network routers do with IP traffic, without imposing this disruptive shift of technology.

Categories and Subject Descriptors

Software and its engineering [**Software notations and tools**]: Compilers—*Runtime environments*

General Terms

Compilation

Keywords

Flow programming, Web, Javascript

1. INTRODUCTION

The growth of web platforms is partially due to Internet's capacity to stimulate services development, allowing very quick releases of a minimal viable products. In a matter of hours, it is possible to upload a first product and start gathering a user community around. *"Release early, release often"* is commonly heard as an advice to quickly gather a user community, as the size of the community is a factor of success.

If the service complies successfully with users requirements, its community might grow with its popularity. To cope with this growth, the resources quantity available to the service shall grow proportionally. Eventually this growth requires to discard the initial monolithic approach that supported the rapid enhancements in features satisfying the community, and requires instead to adopt a more efficient processing model. Many of the most efficient models split the system into parts to reduce local coupling and can distribute the execution on a cluster of commodity machines[10] to support incremental scalability. System S[14, 23], the Staged Event-driven Architecture (SEDA)[22] and MapReduce [6] are examples of that trend. Once split, the service parts are connected by a messaging system, often asynchronous, using communication paradigms like events, messages or streams. Many tools have been developed to express and manage these service parts and their communications. We can cite Spark [24], MillWheel [1], Timestream [21] and Storm [17]. However, these tools propose specific interfaces and lan-

guages, different from the initial monolithic approach generally used in the early steps of a project. It requires the development team either to be trained or to hire experts, and to start over the initial code base. These modifications cause the development team to spend development resources in background without adding visible value for the users.

To lift the risks described above, we propose a tool to compile the initial code base into a high-level language compatible with a more efficient processing model. We focus on web applications driven by users requests, developed in a dynamic language like Javascript using the *Node.js* execution environment. We think that it is possible to analyze this type of application as a stream of requests, to compile it into a network of autonomous, movable functions communicating through data streams. This tool and its runtime aim not to modify the existing code, but rely on a high-level language expression over the initial code base.

This paper presents an early version of this tool as a proof of concept for this compilation approach. The execution environment is described in section 2. The compiler is described in section 3. We compare our work with related works in section 4. And finally, we conclude this paper.

2. FLUXIONAL EXECUTION MODEL

We transform a monolithic application into a network of autonomous parts communicating by message streams. Many execution models designed for such distributed system are renowned for their performances[22, 14, 23, 24, 1, 17]. However, we focus on a compilation approach to replace the shift in programming model rather than the performance of the runtime. Therefore, we present in this section an extremely simplified but generic execution model inspired by the literature, only to support the confirmation of feasibility for the compilation process detailed in section 3.

2.1 Fluxions

The fluxional execution model manages and invokes autonomous execution units. An execution unit listens for, and sends back data in streams. A stream is a continuous and infinite sequence of data encapsulated in messages. A fluxion is a function, as in functional programming, consuming an input stream and generating one or more outputs streams. It is composed of a unique name, a processing function, and a persisted memory called a *context*.

Communications are carried between fluxions by a messaging system. A message is composed of the recipient fluxions' names and a body. At a message reception, the fluxion modifies its *context*, and sends back messages to downstream fluxions. The *context* of a fluxion contains all of the state variables that a fluxion depends on between two executions - that is two message receptions.

Fluxions form a processing chain linked by message streams. All these chains constitute a directed graph, operated by the messaging system.

2.2 Messaging system

The messaging system is the core of our fluxional execution model. It carries messages along streams, and invokes fluxions at a message reception.

The messaging system loops over a message queue to process messages one after the other by invocation of the destination fluxion. The life cycle of a fluxional application is illustrated in figure 1.

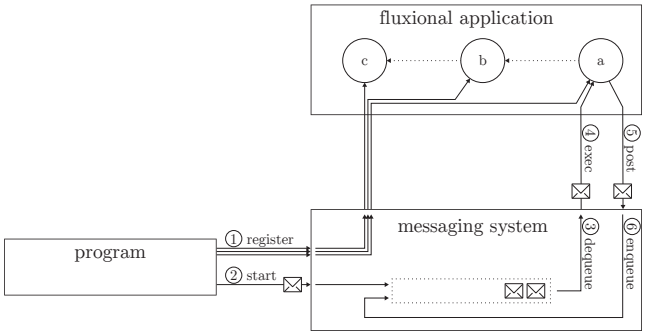


Figure 1: Messaging system details

The messaging system carries message streams based on the names of the recipient fluxions. So it needs every fluxion to be registered. If two fluxions share the same name, the messaging system would be in a conflicting situation. This registration associates a processing function with a unique name and an initial *context*. The registration is done using the `register(<nom>, <fn>, <context>)` function, ①.

To trigger a chain of fluxions, a message is sent using the function `start(<msg>)`, ②. This function pushes a first message in the queue. Immediately, the system dequeues this message and invokes the recipient processing function, ③ and ④. The recipient function sends back messages using the function `post(<msg>)`, ⑤, to be enqueued in the system, ⑥. The system loops through steps ③ and ④ until the queue is empty. This cycle starts again for each new incoming request causing a *start* message.

Algorithms 1 and 2 describe the behavior of the messaging system after the `start` function invocation.

Algorithm 1 Message queue walking algorithm

```
function LOOPMESSAGE()
    while msg presents in msgQueue do
        msg ← DEQUEUE()                          ▷ ③
        PROCESSMSG(msg)
    end while
end function
```

Algorithm 2 Message processing algorithm

```
function PROCESSMSG(msg)
    for dest in msg.dest do
        fluxion ← lookup(dest)
        message ← EXEC(fluxion, msg.body)        ▷ ④ & ⑤
        ENQUEUE(message)                         ▷ ⑥
    end for
end function
```

2.3 Service example

To illustrate the fluxional execution model, we present an example of a simple web application. This application reads the file containing its own source code, and sends it back along with a request counter.

The original source code of this application is available on github[3], and in listing 1. In this source code, some points are worth noticing.

— The `reply` function, line 5 to 11, contains the logic we want to split into the fluxional processing chain. It receives the user request in the variable `res` which is used by the last function of the chain, `reply`.
— The `count` object at line 3 is a persistent memory that increments the request counter. This object needs to be mapped to a fluxion *execution context* in the fluxional execution model.
— The `get` and `send` functionss, respectively line 5 and 9, interface the application with the clients. The processing chain of functions occurs between these two functions : `get` → `handler` → `readFile` → `reply` → `send`.

```
1  var app = require('express')(),
2      fs = require('fs'),
3      count = 0;
4
5  app.get('/', function handler(req, res){
6    fs.readFile(__filename, function reply(err, data) {
7      count += 1;
8      var code = ('' + data).replace(/\n/g, '<br>').
           replace(/ /g, ' ');
9      res.send(err || 'downloaded ' + count + ' times<
           br><br><code>' + code + '</code>');
10   });
11 });
12
13 app.listen(8080);
14 console.log('>> listening 8080');
```

Listing 1: Simple web application. this application replies to every user request with its own source code and the value of a request counter

This application is transformed manually into the fluxions chain depicted in Figure 2. We expect a similar result with the compiler described in section 3. Circles represent registered fluxions. Envelope symbols represent messages streams between fluxions with the variables transmitted from one fluxion to the other. The square in the messaging system holds the *context* of the `reply` fluxion. When a new REST request `GET` is received, a `start` message triggers the flow. The `handler` fluxion receives this `start` message, reads the source file and forwards it to the `reply` fluxion which increments the counter, and sends the result back. Each fluxion propagates the necessary values from one fluxion to the other exclusively by messages. Horizontal dashed lines show virtual transmission of messages between fluxions although they all go through the messaging system.

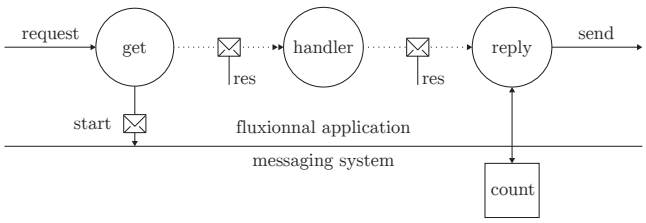


Figure 2: Fluxions chain manually extracted from the example application

Listing 2 describes this counting application in our high-level fluxional language. There is two types of fluxion depending on their input streams. A fluxion with no input stream is a *root* fluxion, while the others are *following* fluxions. The *root* fluxion initializes the application.

There is two types of strea based on the two functions to send messages, *start* and *post*. A stream is defined with an operator indicating its type, a list of recipient fluxions separated by commas, and a description of its message body. The operator to represent a *start* stream is a double arrow, ↠ or `>>`, while for a *post* stream it is a single arrow, → or `->`. The description of the message body is a comma separated list of variables identifiers enclosed in square brackets. It is optional and only gives indications about the message content. A stream is defined with this syntax :
`<operator> <recipients list> [<body description>]`
For the last fluxion of the chain, the output stream is a special stream called `null`.

A fluxion is defined with a header followed by a body with an indentation of two spaces. The header starts with the keyword `flx`, followed by its name and a description of its memory *cakeontext*. After this first line, there is a list of output streams as defined above, and finally the body of the fluxion containing the processing function. The processing function is defined with the Javascript syntax, augmented with the stream syntax to allow it to send output streams. It can access and manipulate only two objects : `msg`, the received message and `this`, the persisted *context*. The *root* fluxion is an exception, as it doesn't have input streams its body isn't a processing function but initialization code. A fluxion is defined with this syntax :

```
flx <name> <context>
<output streams>
<processing function>
```

```
1  flx get
2  >> handler [res]
3    var app = require('express')(),
4        fs = require('fs'),
5        count = 0;
6
7    app.get('/', >> handler);
8    app.listen(8080);
9    console.log('>> listening 8080');
10
11 flx handler
12 -> reply [res]
13   function handler(req, res) {
14       fs.readFile(__filename, -> reply);
15   }
16
17 flx reply {count}
18 -> null
19   function reply(error, data) {
20     count += 1;
21     var code = ('' + data).replace(/\n/g, '<br>').
           replace(/ /g, ' ');
22     res.send('downloaded ' + count + ' times<br><br><
       code>' + code + '</code>');
23   }
```

Listing 2: Manual transformation of the example application in our high-level fluxional language

The application is organized as follow :
— The `get` fluxion is the *root* fluxion. It initializes the application to listen for user requests by calling `app.get`. Every request is forwarded on the stream to the `handler` fluxion, line 7.
— The `handler` fluxion reads the file containing the source code of the application, and forwards the result to the `reply` fluxion, line 14.
— The `reply` fluxion increments the counter, line 20, formats the reply, and sends it back to the user using the function `res.send`, line 22.

Our goal, as described in the introduction, is not to propose a new programming paradigm with this high-level language but to automate the architecture shift with a compiler.

3. DESIGNING A COMPILER FOR FLUXIONAL COMPLIANCY

Current Web applications are mostly written in Java. The langage proposes both data encapsulation and a threading model that ease the development of distributed applications. Yet, Java framework for developing efficient applications are complex systems that impose new APIs[4] to the developers. They are error-prone, and leads to deadlocks and other synchronization problems. Since 2009, *Node.js*[5] provides a Javascript execution environnment for network applications. We focus on this promising environment for its initial simplicity and efficiency. We develop a compiler that transforms a *Node.js* application into a fluxional system compliant with the architecture described in section 2. As javascript forbids user-space thread API, a javascript application is developed as a mono-threaded application. Moreover, in Javascript the memory is hierarchical and the root scope may be accessed by any function, which leads to bad component isolation. Our compiler uses a new approach to find rupture points marking out independent parts into most of *Node.js* applications. It assures isolation and memory consistency for the application parts.

We do not target all Javascript Web-based application as this work is only a proof of concept for the compilation. If we are able to transform a consequent subset of currently running applications without external developer help, we expect a real execution gain in a cloud environment. The rest of this section describes the two parts of the compiler responsible to isolate application parts. Section 3.1 explains how the *analyzer* detects rupture points in the web application to mark out the independent parts. Section 3.2 explains how the *linker* resolves the missing memory dependencies due to the distribution of the central memory.

3.1 Analyzer : execution parallelism

The Sun programming guide [1] defines **parallelism** as *a condition that arises when at least two threads are executing simultaneously*, and **concurrency** as *a condition that exists when at least two threads are making progress. A more generalized form of parallelism that can include time-slicing as a form of virtual parallelism.* **Asynchronism** is a condition that arises when a client continues its execution while waiting for the result to its request.

Promises[16] and callbacks, are abstractions that transform blocking synchronous operations into non-blocking asynchronous operations. Asynchronous operations run in a thread in parallel with the main, and only, thread of the application ; they don't block the application. When the asynchronous operation completes, the requested result is available for the application to retrieve it. The asynchronous operation splits the execution in two concurrent execution paths, one that needs the requested result, the callback or the promise, and one that doesn't. These two concurrent paths are executed by the main thread of the application. A rupture points is where the execution forks in two concurrent paths. These points mark out the limits between the independent parts of an application.

The analyzer detects rupture points to break the application into independent parts. In this section, we define what a rupture point is, and how the compiler detects them.

3.1.1 Rupture points

Rupture points represent a concurrency between two execution paths. They are composed of an asynchronous function, and a callback to process the result of the operation. The first execution path is the suite of instructions following after the asynchronous function. The second execution path is the callback. A rupture point is an interface between these two execution paths and splits them in two application parts. Listing 3 is an example of a rupture point in a simple application. The division of this application into two application parts is illustrated figure 3. The asynchronous function call `fs.readFile` and the callback `function display` represent the rupture point between the first execution path, line 8 and the second, line 4. The first application part is the whole program minus the `function display`, the second application part contains the `function display`, lines 3 to 6.

```
1 var fs = require('fs');
2
3 fs.readFile(__filename, function display(err, data) {
4   console.log('>> second concurrent execution path');
5   console.log(err || data.toString());
6 })
7
8 console.log('>> first concurrent execution path');
```

Listing 3: Example of a rupture point : an asynchronous function call, `fs.readFile()`, with a callback parameter, `function display`

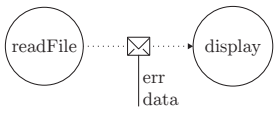


Figure 3: Division of the listing 3 into two application parts, the asynchronous call to `readFile` and the callback `function display`

There are two types of rupture points : *start* and *post*. Figure 4 and 5 illustrate the different interfaces implied by rupture points between two application parts. In these figures, the two concurrent execution paths are indicated by ① and ②, the applications parts are encapsulated in upstream and downstream fluxions.

Start rupture points are on the border between the application and the outside, continuously receiving incoming user requests, like `app.get()` in listing 4. These functions indicate the input of a data stream in the program, and the beginning of a chain of application parts following this stream. The asynchronous function is called only once, while the callback is triggered for each new request. The interface of this rupture point is placed between the two, as illustrated in figure 4, because it implies the least modifications on the program structure.

Post rupture points represent a continuity in the execution flow after a finite asynchronous operation, such as

1. http ://docs.oracle.com/cd/E19455-01/806-5257/6je9h032b/index.html

reading a file in listing 3. As the result of this read operation probably is a voluminous object between two application parts, this interface is specially placed before the call to the asynchronous function, but after the resolution of the arguments. This placement allow the asynchronous function call to occur in the same application parts as the callback, avoiding the transfer of this voluminous result, as illustrated in figure 5. For a write operation, the data transfer is the opposite so the interface is placed between the asynchronous operation and the callback, like for a *start rupture point*, as illustrated in figure 4

The impact of these rupture points on the application structure is illustrated in listing 7 and explained in section 3.3.

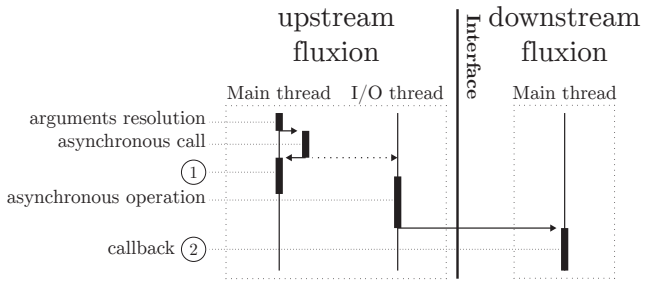


Figure 4: Basic rupture point interface. The rupture point interface is placed between the asynchronous operation and the callback to reduce the impact on the application structure.

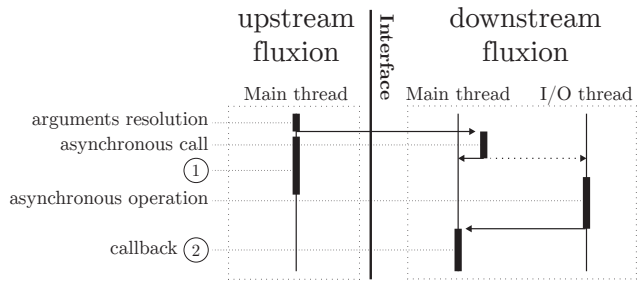


Figure 5: Special rupture point interface. The rupture point interface is placed before the asynchronous operation, to avoid moving the result accross the interface from one application part to the other. However, the resolution of the arguments for this asynchronous call is kept upstream to reduce the impact on the application structure

The application in listing 4 presents the two types of rupture points. The division of this application is illustrated figure 6.

```
1  var app = require('express')(),
2      fs = require('fs');
3
4  app.get('/', function handleRequest(req, res) {
5    fs.readFile(__filename, function reply(err, data) {
6      res.send(err || data.toString());
7    })
8  });
9
10 app.listen(8080);
11 console.log('server listening to 8080');
```

Listing 4: Example of an application presenting the two types of rupture points : a `start` with the call to `app.get()`, and a `post` with the call to `fs.readFile()`

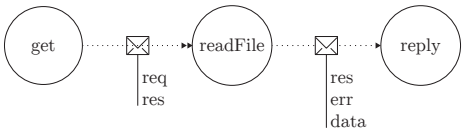


Figure 6: Division of the listing 4 into three application parts

3.1.2 *Detection*

Detecting a rupture point requires detecting its two components : the asynchronous function and the callback function.

Asynchronous functions

The compiler is prebuilt knowing some module names exposing asynchronous function, like the *express*, and the *fs* module in listing 4. To detect asynchronous calls, the compiler keeps a list of variables holding these modules. In listing 4, the compiler adds both variables `app` and `fs` in this list. When the compiler encounter a call expression, it compares its callee name with this list to spot asynchronous functions.

Callback function

For each asynchronous call expression detected, the compiler test if one of the arguments is of type `function` to spot the callback. Some callback functions are declared *in situ*, and are trivially detected. For every other variable identifier, we track the declaration up to the initialization value to detect its type.

Variable tracking

To detect both the asynchronous function and the callback function, the compiler needs to statically track the types of variables. Missing rupture points by false negatives in the detection is sub-optimal, but false positives are more critical, as they eventually produce a runtime execution. Therefore, the detection needs to be as accurate as possible to screen out false positives. We use a technique similar to a Program Dependency Graph (PDG)[9] to track changes in the value of variables. This method walks an intermediate representation of the source code to spot the statements modifying a certain variable. From this intermediate representation, the variable tracker builds a dependency graph which helps the analyzer to detect the type of a variable at a certain point in the execution. The variable tracker is still in early development and is limited to only a few cases.

3.2 Linker : memory distribution

Parallelism is not enough for an application to be distributed. The compiler also needs to distribute the memory into the application parts for the application to be compliant with the fluxional execution model. In Javascript, scopes are nested one in the other up to the all-enclosing global scope. Each function creates a new scope containing variables local to itself, and chained to the scope of the parent function. Rupture points always take place in between scopes. A rupture point placed between a child scope and its pa-

rent breaks a chain of scopes, and makes the child unable to access its parent as expected, eventually leading the application to a runtime error. The linker analyzes how scopes are distributed among the application parts to detect and resolve unmet dependencies between them. Depending on how a shared variable is used in the scopes, there are different situations to resolve. In this section, we explain the different cases of inconsistency emerging from the partitioning of a central memory.

3.2.1 Signature

The signature is the part of a message containing all the variables to send downstream. If a variable is needed for read-only access by at least one downstream application part, like the variable `res` in figure 3, it is added to the signature of the rupture point. As fluxions are chained one after another, a fluxion must provide every dependency for the next one, even if some of this dependencies are not needed by the current application part. These dependencies must be passed fluxion after fluxion from the producing fluxion to the consuming fluxion. The compiler modifies those variable identifiers inside the application part to make them point to the signature instead of the function scope.

3.2.2 Scope

The scope of an application part consists of all the variables declared outside, but needed for modification in only this application part. If one of these variables needs to be read by another application part downstream, this variable also becomes part of the signature sent downstream. An example of such a variable is the `counter` in listing 5. Initialized to 0 in the global scope, it is incremented for each request by the `function reply`. This counter is in the scope of the application part containing `function reply`. The scope of an application part is stored in the *context* of the encapsulating fluxion.

3.2.3 Sync

If a variable is needed for modification by more than one application parts, this variable needs to be synchronized between the fluxions. The synchronization of a distributed memory is a well-known subject, with Brewer's theorem[11][12], and the ACID (Atomicity, Consistency, Isolation, Durability) versus BASE (Basically Available, Soft state, Eventual consistency) opposition[10]. We choose to stay out of this topic, the objective for this compiler is to be able to transform only a subset of web application with a satisfying result. The compiler merges the parts too tightly coupled by modification accesses on a shared variable.

3.3 Compilation example

To illustrate the compiler features, we compiled the application used as an example for the execution model in section 2. The source and compiled results of this application are available on github[3] [2]. The compiler source code is the property of Worldline, and is not publicly available, but we are planning of releasing it as an open source project in a near future.

To test the source or the result of the compilation, one would launch respectively `source.js` or `result.js` with `node` and check the service available at `localhost:8080`. Both executable needs their dependencies to be resolved with `npm` before execution.

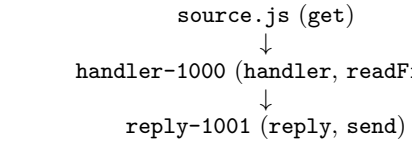
```
git clone https://github.com/etnbrd/flx-example
cd flx-example
npm install
node result.js
open http://localhost:8080
```

The file `source.js`, in listing 5, is the source of this compilation example. This application sends back its own source along with a download counter. The processing chain of function is : `get` → `handler` → `readFile` → `reply` → `send`. It uses two asynchronous function call with *in situ* callback, one to listen for user requests and one to read its own source, respectively `app.get` line 5 and `fs.read` line 6. It uses a global variable to increment the download counter defined line 3. This global variable is used only in the `reply` function, line 7 and 9.

```
1  var app = require('express')(),
2      fs = require('fs'),
3      count = 0;
4
5  app.get('/', function handler(req, res){
6    fs.readFile(__filename, function reply(err, data) {
7      count += 1;
8      var code = ('' + data).replace(/\n/g, '<br>').
           replace(/ /g, ' ');
9      res.send(err || 'downloaded ' + count + ' times<
           br><br><code>' + code + '</code>');
10   });
11 });
12
13 app.listen(8080);
14 console.log('>> listening 8080');
```

Listing 5: Source of the compilation example

The result of the compilation into our high-level language is in the file `result.flx`, presented in listing 6. The analyzer detects both asynchronous calls as rupture points. The first one is a *start* rupture point, associated with the `app.get` asynchronous function call which makes the callback `handler` listen for the stream of user requests. The second one is a *post* rupture point, associated with the `fs.readFile` asynchronous function call which reads the source file and hands it to the callback `reply`. These two rupture points result in three application parts. The first application part is encapsulated in the root fluxion, named after the filename, `source.js`, line 18. It initializes the system to route the user requests to the fluxion `handler-1000`, line 9. This second fluxion reads the file, and sends the result to the next and last fluxion `reply-1001`, line 1. We can identify the processing chain of functions in this chain of fluxion.



The linker detects that the fluxion `reply-1001` needs two variable to send the result back to the user, `res` and `count`. The variable `res` depends on the user connection and is initialized for each new request. It needs to be part of the *signature* of the message transfered to the last fluxion. The variable `count` is global, and the function `reply` in the fluxion `reply-1001` needs to increment it at each new request. This

2. https ://github.com/etnbrd/flx-example/releases

global variable is in the *scope* of only this application part, so the compiler stores it in the *context* of this fluxion. The division by the compiler of this application is illustrated figure 7. This result is very similar to the manual division illustrated figure 2.

```
1  flx source.js {}
2  >> handler-1000 [res]
3    var app = require('express')(), fs = require('fs'),
         count = 0;
4    app.get('/', >> handler-1000);
5    app.listen(8080);
6    console.log('>> listening 8080');
7
8  flx handler-1000 {fs}
9  -> reply-1001 [res]
10   function handler(req, res) {
11     fs.readFile(__filename, -> reply-1001);
12   }
13
14 flx reply-1001 {count}
15 -> null
16   function reply(err, data) {
17     count += 1;
18     var code = ('' + data).replace(/\n/g, '<br>').
           replace(/ /g, ' ');
19     res.send(err || 'downloaded ' + count + ' times<
           br><br><code>' + code + '</code>');
20   }
```

Listing 6: High level fluxional language result of the compilation example

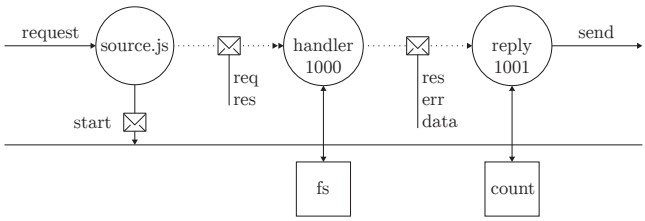


Figure 7: Division of the listing 6 into three application parts. This result is similar to the manual division illustrated in figure 2

The compiler also produces an executable targeting a simple implementation of the fluxional execution model. This result is in the file `result.js`, presented in listing 7. The root fluxion is not registered because it doesn't need to receive any messages by another fluxion, it only initializes the application. The two following fluxions are registered in the messaging system. This registration encapsulates the processing function in a `function capsule`. The *special rupture points* implies the asynchronous call to be in the downstream fluxion. The `function capsule` encapsulates the asynchronous call from a *special rupture points* or the callback from *basic rupture points* in a unified processing function.

The original callback is replaced with a `function placeholder`, line 3, sending the *start* message to the fluxion `handler-1000`. Line 17, the fluxion `handler-1000` pushes the user request `res` in the message and `post` it directly to the next fluxion, `reply-1001`. Because there is a special rupture point between the fluxion `handler-1000` and `reply-1001`, the asynchronous call is moved to the next fluxion and the `function post` doesn't replace the callback, like the placeholder line 3, but is directly called. Finally, the fluxion `reply-1001` receives the message containing the user request and read the file. The callback of this asynchronous operation, line 27, increments the variable `counter`, line 28, and sends the reply, line 30.

```
1  var flx = require('flx');
2  var app = require('express')(), fs = require('fs'),
       count = 0;
3  app.get('/', function placeholder() {
4    return flx.start(flx.m('handler-1000', {
5      _args: arguments,
6      _sign: {}
7    }));
8  });
9  app.listen(8080);
10 console.log('>> listening 8080');
11
12 // handler-1000
13 // -> reply-1001 [res(signature), count(scope)]
14
15 flx.register('handler-1000', function capsule(msg) {
16   (function handler(req, res) {
17     flx.post(flx.m('reply-1001', {
18       _args: arguments,
19       _sign: { res: res }
20     }));
21   }.apply(this, msg._args));
22 }, { fs: fs });
23
24 // reply-1001 -> null
25
26 flx.register('reply-1001', function capsule(msg) {
27   fs.readFile(__filename, function reply(err, data) {
28     this.count += 1;
29     var code = ('' + data).replace(/\n/g, '<br>').
           replace(/ /g, ' ');
30     msg._sign.res.send(err || 'downloaded ' + this.
           count + ' times<br><br><code>' + code + '</
           code>');
31   }.bind(this));
32 }, { count: count });
```

Listing 7: Fluxional execution model result of the compilation example

3.4 Limitations

This compiler aims at transforming a subset of Javascript web applications presenting a specific syntax and design. In this section, we describe briefly the current limitations of our compiler and how we plan to overcome them in future works.

- Variables poorly encapsulated or used too broadly tighten dependencies accross the code, and might result in a coarser division of the application.
- The compilation silently fails if a variable holding a callback or a module is overwritten, or not defined in the declaration. The variable tracker is unable to track accurately all the modification of a variable to detect these situations which may lead the compiler either to miss rupture points, or to detect non existing one.
- The compiler is unable to track a dynamically resolved value, even if the value is deducible statically. If this variable is used in a potential rupture point, the compiler screens it out.
- The Javascript language offers rich composition possibilities leading to many corner cases. The compiler is not robust enough to understand all corner cases. For example, the *express* module is only detected if initialized like in listing 4.

There may be other limitations we aren't aware of.

The three last limitations described above are caused by the variable tracker - described in section 3.1 - being in an early stage of development. We are currently in the process

of improving the robustness of the compiler to extend the subset of compilable applications.

4. RELATED WORKS

The execution model, is inspired by some works on scalability for very large system, like the Staged Event-Driven Architecture (SEDA) of Matt Welsh[22], System S developped in the IBM T. J. Watson research center[14, 23], and later the MapReduce architecture[6]. It also drew its inspiration from more recent work following SEDA. Among the best-known following works, we cited in the introduction Spark [24], MillWheel [1], Timestream [21] and Storm [17]. The idea to split a task into independent parts goes back to the Actor's model[13] in 1973, and to Functional programming, like Lucid[2] in 1977 and all the following works on Data-Flow leading up to Flow-Based programming (FBP)[18] and Functional Reactive Programming (FRP)[7]. Both FBP and FRP, recently got some attention in the Javascript community with, respectively, the projects *NoFlo*[19] and *Bacon.js*[20].

The first part of our work stands upon these thorough studies, however, we are taking a new approach on the second part of our work, to transform the sequential programming paradigm into a network of communicating parts known to have scalability advantages. Promises[16] are related to our work as they are abstractions from a concurrent programming style, to an asynchronous and parallel execution model. However, our approach using Node.js callback asynchronism to automate this abstraction seems unexplored yet.

The compiler uses AST modification, as described in[15], and an approach similar to the Program Dependency Graph (PDG)[9]. Our implementation is based on the work by Ryan Dahl : *Node.js*[5], as well as on one of the best-known web framework available for *Node.js* : *Express*[8].

5. CONCLUSION

In this paper, we presented our work on a high-level language allowing a high-level code reasoning. We presented a compiler to transform a *Node.js* web application into a network of independent parts communicating by message streams. To identify these parts, the compiler spots rupture points in the application indicating an independence between two parts, possibly leading to parallelism and memory distribution. We also presented the execution model to operate an application expressed in our high-level language. This distributed approach allows code-mobility which may lead to a better scalability. We believe this high-level approach can enable the scalability required by highly concurrent web applications without discarding the familiar monolithic and asynchronous programming model used in *Node.js*.

Références

[1] T Akidau et A Balikov. ■ MillWheel : Fault-Tolerant Stream Processing at Internet Scale ■. In : *Proceedings of the VLDB Endowment 6.11* (2013).

[2] Edward A Ashcroft et William W Wadge. ■ Lucid, a nonprocedural language with iteration ■. In : *Commun. ACM* 20.7 (1977), p. 519–526.

[3] Etienne Brodu. *flx-example*. doi : 10.5281/zenodo.11945.

[4] D Coward et Y Yoshida. ■ Java servlet specification version 2.3 ■. In : *Sun Microsystems, Nov* (2003).

[5] Ryan Dahl. *Node.js*. 2009.

[6] J Dean et S Ghemawat. ■ MapReduce : simplified data processing on large clusters ■. In : *Commun. ACM* (2008).

[7] C Elliott et Paul Hudak. ■ Functional reactive animation ■. In : *ACM SIGPLAN Not.* (1997).

[8] *Express*.

[9] J Ferrante, KJ Ottenstein et JD Warren. ■ The program dependence graph and its use in optimization ■. In : *ACM Transactions on ...* (1987).

[10] A Fox, SD Gribble, Y Chawathe, EA Brewer et P Gauthier. *Cluster-based scalable network services*. 1997.

[11] S Gilbert et N Lynch. ■ Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services ■. In : *ACM SIGACT News* (2002).

[12] Coda Hale. *You Can't Sacrifice Partition Tolerance | codahale.com*. 2010.

[13] C Hewitt, P Bishop, I Greif et B Smith. ■ Actor induction and meta-evaluation ■. In : *Proc. 1st Annu. ACM SIGACT-SIGPLAN Symp. Princ. Program. Lang.* (1973).

[14] N Jain, L Amini, H Andrade et R King. ■ Design, implementation, and evaluation of the linear road benchmark on the stream processing core ■. In : *Proceedings of the ...* (2006).

[15] J Jones. ■ Abstract syntax tree implementation idioms ■. In : *Proc. 10th Conf. Pattern Lang. Programs* (2003).

[16] B Liskov et L Shrira. *Promises : linguistic support for efficient asynchronous procedure calls in distributed systems*. 1988.

[17] Nathan Marz, James Xu, Jason Jackson et Andy Feng. *Storm*. 2011.

[18] JP Morrison. *Flow-based programming - introduction*. 1994.

[19] *NoFlo*.

[20] Juha Paananen. *Bacon.js*. 2012.

[21] Z Qian, Y He, C Su, Z Wu et H Zhu. ■ Timestream : Reliable stream computation in the cloud ■. In : *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)* (2013).

[22] M Welsh, SD Gribble, EA Brewer et D Culler. *A design framework for highly concurrent systems*. 2000.

[23] KL Wu, KW Hildrum et W Fan. ■ Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on System S ■. In : *Proceedings of the 33rd ...* (2007).

[24] M Zaharia et M Chowdhury. ■ Spark : cluster computing with working sets ■. In : *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* (2010).