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

Gautier di Folco
gautier.difolco@insa-lyon.fr
IXXI – ENS Lyon
15 parvis René Descartes – BP 7000
69342 Lyon Cedex 07 FRANCE

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 a compiler to abstract web applications into a high-level language, allowing a high-level code reasoning. That reasoning may allow one to provide code mobility to dynamically cope with audience growth and decrease.

The high-level language allows to express Web applications 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 according to their interdependencies to minimize overall data transfers. We expect that this dynamic reorganization can allow an application to deal with its load.

Our high-level language proposition consists of an execution model which dynamically adapts to the execution environment, and a compiler to automate the technological shift between the classical model and the proposed one.

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 monolithic initial approach that supported the rapid growth in features satisfying the community, and adopt a more efficient processing model. Many of the most efficient models split the system into parts to reduce local coupling and are based on a cluster of commodity machines[11] to support incremental scalability. System S[15, 24], the Staged Event-driven Architecture (SEDA)[23] 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 [25], MillWheel [1], Timestream [22] and Storm [18]. However, these tools propose specific interfaces and languages, generally different from the tools used in the early step of a project. It requires the development team either to be trained or to hire experts and to start over the initial code base. Therefore, these modifi-

cations cause the development team to spend development resources in background without adding visible value for the user.

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

This paper present an early versions of this tool as a proof of concept for a change in programming model. The execution environment is described in section 2. The compiler is described in section 3. In section 4, we compare our work with related works. Finally, we conclude this paper.

2. FLUXIONAL EXECUTION MODEL

The tool we present in this paper transforms a monolithic application into a network of autonomous parts. There exist many execution models designed for such distributed system renowned precisely for their performances. However, we focus on the shift in programming model rather than the performance of a runtime. Therefore, we present in this section an extremely simplified execution model inspired by the literature[23][15, 24][25][1][18], only to support the confirmation of feasibility for the compilation process detailed in section 3.

2.1 Fluxions

The fluxional execution model role is to manage and invoke autonomous execution units. An execution unit accepts only streams as input and output, that is a continuous and infinite sequence of data encapsulated in messages. We named this execution unit a fluxion. That is a function, as in functional programming, only dependent from data streams. It is composed of a unique name, a processing function, and a persisted memory context.

Messages are carried by a distributed messaging system. They are composed of the name of the recipient fluxion and a body. At a message reception, the fluxion modifies its context, and sends back messages to downstream fluxions. The fluxion's execution context is defined as the set of state variables on which the fluxion depends between two executions - that is two messages receptions.

The fluxions form a chain of processing binded by data 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.

A message queue is at the core of this messaging system. Each message is processed one after the other by invocation of the recipient fluxion. The life cycle of a fluxional application is pictured on figure 1.

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 a name, the messaging system would be in a conflicting situation. This registration

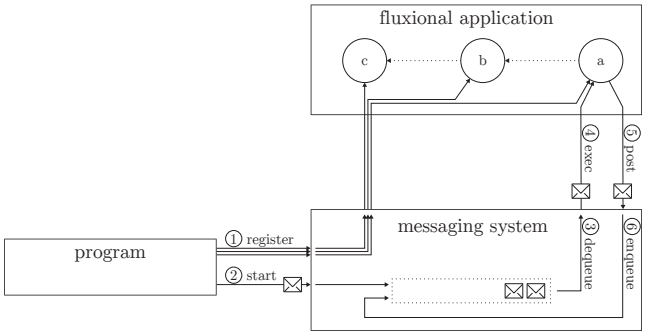

Figure 1: Messaging system details

matches a processing function with a unique name and an initial execution context. The registration is done using the function `register(<nom>, <fn>, <context>)`, step ① on figure 1.

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

The algorithms 2 and 1 precisely describe the behavior of the messaging system after the function `start` 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 External interfaces

In order to interact with other systems, we define external border interfaces. As a first approach, our goal is to interface Web architectures, so we need to communicate with a REST[10] client. We define two components in this interface :

In receives client connections. For every incoming connection, it relays a connection identifier to the **Out** component for the reply. It then relays the connection identifier and the request to the first fluxion by calling the `start` function.

Out sends the result of the processing chain back to the client. To receive messages from the processing chain, the component **Out** is registered in the messaging system under the name `out`.

Figure 2 pictures the specific elements of the web interface inside the fluxional system.

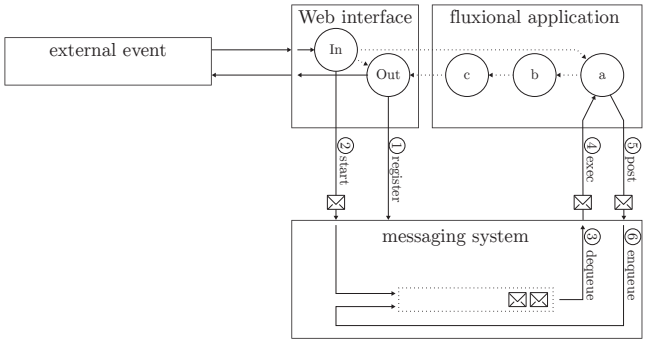

Figure 2: Fluxional application with web interface

2.4 Service example

In order to picture the fluxional execution model, we present an example of a simple visit counting service. This service counts the number of HTTP connections for each user, and sends him back this number in the HTTP reply. This example is not a compilation result, but only an illustration for the fluxionnal execution model.

```
1  var app = require('express')();
2
3  var count = {};
4
5  app.get('/:id', function reply(req, res){
6    count[req.params.id] = count[req.params.id]  || 1;
7    ++count[req.params.id]
8    var visits = count[req.params.id];
9    var reply = req.params.id + ' connected ' + visits
         + ' times.';
10   res.send(reply);
11 });
12
13 port = 8080;
14 app.listen(port);
15 console.log("Listening port: "+port);
```

Listing 1: Original code of a simple service example

The initial version of this service could look like code listing 1. In this code listing, three elements are worth noticing.

— The `count` object at line 3 is a persistent memory that stores each user visit count. This object is mapped to a fluxion *execution context* in the fluxional system.

— The `reply` function, line 5 to 11, contains the logic we want to express in the fluxional processing chain.

— The two functions `get` and `send`, respectively line 5 and 10, interface the logic with the external interface. The hidden processing chain of functions is : `get` → `reply` → `send`

This minimal service is transformed manually into the fluxions chain depicted in Figure 3. We expect a similar result with the compiler described in next section.

The circles in figure 3 represent registered fluxions. Envelope symbols represent exchanged messages between fluxions with the data transmitted from one fluxion to the other. Finally, squares stored in the messaging system hold the *execution context* for the logic and **Out** fluxions. When a new `get` REST message is received at the **In** end point, a `start` message triggers the flow. Concurrently the **In** fluxion set a `cid` parameter to the **Out** fluxion execution context. This `cid` is associated to the client connexion to which the last fluxion redirects the answer. The `cid` tags the request and is transmitted through the flow until the **Out** fluxion. Each fluxion propagates the necessary values from one fluxion to the other exclusively by messages. Horizontal dashed lines show message virtual transmission between fluxions although they all go through the messaging system.

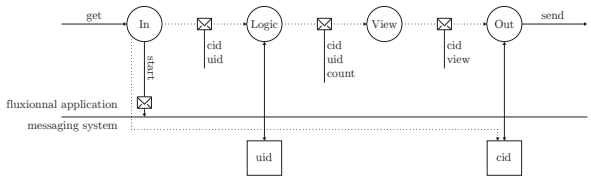

Figure 3: Count service fluxions chain

Listing 2 describes this counting service in our fluxional language. This new language brings a stricter segmentation than the initial code by allowing the developer to only define and register fluxions. A fluxion is defined by a name, and a list of destinations preceded by the operator `->`. Fluxions can access and manipulate only two objects : `msg` and `this`. The first is the received message, the second is the persisted context of the fluxion. Fluxions use the Javascript language syntax inside their definition.

```
1  use web
2
3  fluxion logic -> view
4    this.uid[msg.uid] = this.uid[msg.uid] + 1 || 1
5    msg.count = this.uid[msg.uid]
6    post msg
7
8  fluxion view -> output
9    msg.view = msg.uid + " connected " + msg.count + "
         times."
10   msg.uid = undefined
11   msg.count = undefined
12   post msg
13
14 register logic, {uid: {}}
15 register view
16
17 web.listen
```

Listing 2: Fluxional sample

Except from the two interface components, the service is organized as follow :

— The `logic` fluxion is the first to receive the client's message. It contains the whole logic of this simple service. A real service would need a more complex chain with logic distributed across multiple fluxions, instead of a single fluxion. It increments the count for the received user identifier, pushes this count inside the message, and relays it to the next fluxion.

— The `view` fluxion receives this message, formats it for the user, and relays it to the output fluxion.

We use this interface to develop web services using the fluxional execution model. But our goal, as described in the introduction, is to automate this architecture shift, not to impose a new programming paradigm onto the developer.

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. Since 2009, *Node.js*[5] provides a simple 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 simple Node.js application into a fluxional system compliant to 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 finds fluxions into most of Web-based Javascript application. It finds component isolation through and assure memory consistency.

We do not target all Javascript Web-based application as this work is only a proof of concept of the compilation process. But 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 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 communication point continues processing an independent thread of execution while waiting for the answer to his request.

Promises[17], as well as *Node.js* callbacks, are abstractions that transform blocking synchronous operations into non-blocking asynchronous operations. These asynchronous operations run concurrently with the main thread, until the requested value is available for the main thread to continue the computation needing this value. The callback asynchronism splits the execution in two concurrent execution paths, one that needs the requested value and one that doesn't. A rupture points is where the execution flow forks in two concurrent paths due to this concurrent asynchronism. 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 fork in the execution flow due to an asynchronous operation. They are composed of an asynchronous function, and a callback to process the result of the operation. The first execution flow path is the suite of instructions following after the asynchronous function. The second execution flow path is the callback. A rupture point is an interface between two application parts. Listing 3 is an example of a rupture point in a simple application.

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

There are two types of rupture points : root and following. Figure 4 and 5 are used to illustrate the different types of interface in the two rupture points. In these figures, the two concurrent execution paths distributed in two application parts are indicated by ① and ②.

Root rupture points are on the interface between the whole application and the outside, continuously receiving incoming user requests, like `app.get()` in listing 4. The callbacks of these functions indicate the input of a data stream in the program, and the beginning of a chain of application parts following this stream. Because the asynchronous function is called only once, while the callback is triggered multiple times, this interface is placed between the two, as illustrated in figure 4.

Following rupture points represent a continuity in the execution flow after a finite asynchronous operation, such as reading a file in listing 3. As the result of this read operation probably being a voluminous object, this frontier 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 inversed so the frontier is placed between the asynchronous operation and the callback, like for a *root rupture point*, as illustrated in figure 4.

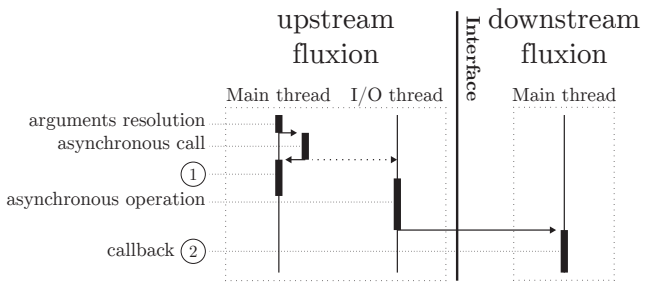

Figure 4: Basic rupture point interface. The interface is placed after the asynchronous operation, for the downstream application part to be triggered at each event.

In the following, *root* rupture points are called *start*, and *following* rupture points are called *post*.

```
1 var app = require('express')();
2 var fs = require('fs');
```

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

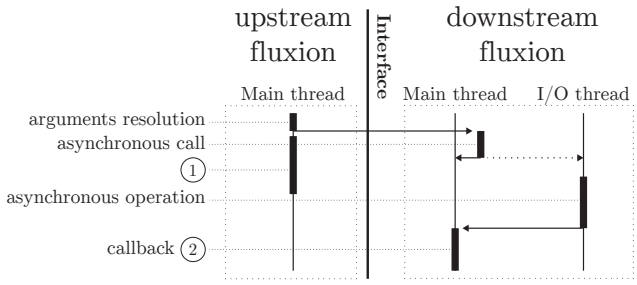

Figure 5: Special rupture point interface. The interface is placed before the asynchronous operation, to avoid moving the result from one application part to another.

```
3
4  app.get('/:filename', function handleRequest(req, res
       ) {
5    fs.readFile(__dirname + '/' + req.params.filename,
         function reply(err, data) {
6      res.send(err || data);
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()`

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 module names exposing asynchronous function, like the *express*, and the *fs* module in listing 4. To detect asynchronous calls, the compiler register variables holding such modules, to later detects their asynchronous function calls. In listing 4, the compiler register both variables `app` and `fs`. When the compiler encounter a call expression, it compare its callee name to the registered ones to spot asynchronous functions.

Callback function

For each asynchronous call expression detected, the compiler test if an argument is a function. 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 functions.

Variable tracking

To detect both the asynchronous function and the callback function, the compiler needs to statically track the states of the variables. Missing rupture points by false negatives in the detection is sub-optimal, but false positives eventually produce an unstable compilation result. 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.

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 is chained to the scope of the parent function. Rupture points take place in between scopes, linking application parts in message streams. A rupture point placed between a child scope and its parent break a chain of scope, and makes the child unable to access its parent as expected, eventually leading the application to crash during execution. The linker analyzes how scopes are distributed among the application parts to detect and resolve unmet dependencies between these scopes. Depending on how a shared variable is used in the functions scopes, there are different situations to resolve. In the following, 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, 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 the code inside the application part for the signature's references to point to the message signature instead of the function scope.

3.2.2 Scope

The scope of an application port holds 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 becomes part of the signature sent downstream. An example of such a variable is a request counter. Initialized to 0 in the global scope, the counter is incremented for each request. This counter would be in the scope of the application part handling requests reception, and sent downstream for visit metrics processing. 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 distributed application parts, this variable needs to be synchronized between the fluxions' scopes. The synchronization of a distributed memory is a well-known subject, with Brewer's theorem[12][13], and the ACID (Atomicity, Consistency, Isolation, Durability) versus BASE (Basically Available, Soft state, Eventual consistency) opposition[11]. 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 sharing variables with modification access.

3.3 Guidelines and Future work

This compiler aims at transforming efficiently a subset of Javascript web applications presenting a specific syntax and design. In this section, we describe briefly the main situations to avoid for an efficient compilation.

— The compilation silently fails if a variable holding a callback or a module is assigned a second time. The variable tracker is unable to track accurately all the modification of a variable to detect this situation and screen out the corresponding rupture point.
— The compiler is still unable to track the value of dynamically resolved variables. If this variable is then used as a callback, the compiler won't be able to detect it as a rupture point.
— Variables poorly encapsulated or used too globally tighten dependencies accross the code. In these conditions, the compiler is unable to efficiently break the application in parts, resulting in poor distribution and scalability.
— The *express* module uses an uncommon pattern to initialize the object holding the asynchronous functions. The compiler is still unable to understand this pattern, and fails if the initialization differs from code listing 4.

Most of the 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 extending further the reach of this component to extend the subset of compilable applications.

We believe that our work will keep scalability concerns out of the way from the development team, who could then focus on the core logic of their application. In future developments of this project, we aim at making application dynamically reactive to the load of user requests. By monitoring only the input stream, the `start` rupture points, we believe it is possible to infer the load propagation through the application. Using analogy with fluid dynamics, each fluxion is like a pipe, traversed by a fluid of user requests. The input and output throughput of this pipe could be calibrated before production use, generating an approximative model of the application reaction to input load. Using this model, we want to make the application's reorganize itself in a cluster to handle pikes in the user request throughput.

3.4 Compilation example

To illustrate the compiler features, we compiled a very simple, yet representative, application. 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 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`, after installing the dependencies with `npm`, containing the fluxional execution model `flx`.

```
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 source contains two asynchronous function call with *in situ* callback, one to receive user requests and one to read its own source, respectively `app.get` line 5 and `fs.read` line 7. It also uses a global variable for the download counter defined line 3 and used only in the `reply` function, line 8 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(error, data)
         {
7      count += 1;
8      var code = ('' + data).replace(/\n/g, '<br>').
           replace(/ /g, ' ');
9      res.send('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 hgh-level language is in the file `result.flx`, presented in listing 6. The analyzer detects both asynchronous call as rupture points. The first one is a *start* rupture point, it receives the stream of user requests. The second one is a *post* rupture point, it reads the source file. These two rupture points result in three different application parts. The first application part is the root fluxion, named after the filename, `source.js`. It initialize the system to route the user requests to the next fluxion, named after the callback, `handler-1000`. This second fluxion reads the file, and send the result to the next fluxion `reply-1001`.

The linker detects that the fluxion `reply-1001` needs two variable to send the result back to the user : `res` and `count`, as seen line 10 in listing 6. The variable `res` depends on the user connection and change for each new request. It needs to be part of the *signature* of the message transfered until the last fluxion. The variable `count` is global, and needs to increment at each new request. So the compiler store it in the context of this fluxion.

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

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

The compiler also produce an executable targeting the fluxional execution model in the file `result.js`, presented in listing 7.

```
1  var flx = require('flx');
```

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

```
var app = require('express')(), fs = require('fs'),
    count = 0;
app.get('/', function placeholder() {
  return flx.start(flx.m('handler-1000', {
    _args: arguments,
    _sign: {}
  }));
});
app.listen(8080);
console.log('>> listening 8080');

// reply-1001 -> null

flx.register('reply-1001', function capsule(msg) {
  if (msg._update) {
    for (var i in msg._update) {
      this[i] = msg._update[i];
    }
  } else {
    fs.readFile(__filename, function reply(error,
        data) {
      this.count += 1;
      var code = ('' + data).replace(/\n/g, '<br>').
          replace(/ /g, ' ');
      msg._sign.res.send('downloaded ' + this.count +
          ' times<br><br><code>' + code + '</code>')
          ;
    }.bind(this));
  }
}, { count: count });

// handler-1000
// -> reply-1001 [res(signature), count(scope)]

flx.register('handler-1000', function capsule(msg) {
  if (msg._update) {
    for (var i in msg._update) {
      this[i] = msg._update[i];
    }
  } else {
    (function handler(req, res) {
      flx.post(flx.m('reply-1001', {
        _args: arguments,
        _sign: { res: res }
      }));
    }.apply(this, msg._args));
  }
}, { fs: fs });
```

Listing 7: Fluxional execution model result of the compilation example

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[23], System S developped in the IBM T. J. Watson research center[15, 24], 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 [25], MillWheel [1], Timestream [22] and Storm [18]. The idea to split a task into independent parts goes back to the Actor's model[14] 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)[19] and Functional Reactive Programming (FRP)[7]. Both FBP and FRP, recently got some attention in the Javascript community with, respectively, the projects *NoFlo*[20] and *Bacon.js*[21].

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[17] 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[16], 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 to enable a *Node.js* application to be dynamically and automatically scalable. The emerging design for an application to be scalable is to split it into parts to reduce coupling. From this insight, we designed an execution model for applications structured as a network of independent parts communicating by streams of messages. In a second part, we presented a compiler to transform a Javascript application into a network of independent parts. To identify these parts, we spotted rupture points as indicators for a possible parallelism and memory distribution. This compilation tool allow for the use of the distributed architecture previously described to enable scalability, with a minimum change on the imperative programming style mastered by most developers.

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.11906.

[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] RT Fielding et RN Taylor. ■ Principled design of the modern Web architecture ■. In : *Proc. 2000 Int. Conf. Softw. Eng.* (2002).

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

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

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

[14] 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).

[15] 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).

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

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

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

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

[20] *NoFlo*.

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

[22] 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).

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

[24] 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).

[25] 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).