Definition of a fluxionnal execution model

How to abstract parallelisation constraints from the developer
Research 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
107-109 boulevard Vivier Merle
69438 Lyon Cedex 03

Frédéric Oblé
frederic.oble@worldline.com
Worldline
107-109 boulevard Vivier Merle
69438 Lyon Cedex 03

ABSTRACT

The audience's growth a web application needs to adapt to, often leads its development team to quickly adopt disruptive and continuity-threatening shifts of technology. To avoid these shifts, we propose an approach that abstracts web applications into an high-level language, which authorizes code mobility to cope with audience dynamic growth and decrease.

We think a web application can be depicted as a network of small autonomous parts moving from one machine to another and communicating by message streams. The high-level language we propose aims at expressing these parts and their streams. We named these parts fluxions, by contraction between a stream [1] 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 cope with its load.

Our high-level language proposal consists of an execution model which dynamically adapts itself to the execution environment, and a tool 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. flux in french

1. INTRODUCTION

The growth of web platforms is partially caused by Internet's capacity to stimulate services development, allowing very quick release of 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, the community will grow gradually as the service gain popularity. To cope with this growth, the resources quantity taken up by the service shall grow exponentially. This continues until the amount of data to process requires the development team to use a more efficient processing model to make better use of the resources. Many of the most efficient models split the system into parts to reduce their coupling and migrate them to more resourceful environment. MapReduce [5] is an example of this trend. Once split, the different service's parts are connected by a messaging system, often asynchronous, using communication paradigms like *three-tiers* architecture, events, messages or streams. Many tools have been developed to express and manage these different service's parts and their communications. We can cite Spark [18], MillWheel [1], Timestream [17] and Storm [12]. However these tools use specific interfaces and languages. Thus, it requires the development team to be trained, to hire experts and to start over the initial code base, while this new architecture is not as flexible and adaptable for quick modifications, as the initial code base was. Thus, these modifications implies the development team to take risks without adding concrete value to the service.

We propose a tool able to automate this technical shift without the need of an architecture shift. Such a tool might lift

the risks described above. We aim at providing this tool to Web applications for which load comes from users requests streams. Applications for which initial development uses a simple web paradigm consisting of a web server, data processing logic, and a database. We think that it is possible to analyze this type of application to express it using autonomous, movable functions communicating by data streams. And to shift architecture as soon as the first public release, without wiping off the initial code base.

We assume these applications are developed in a dynamic language like Javascript using *Node.js* execution environment, and we propose a tool able to identify internal streams and stream processing units, and to dynamically manage these units. The tool aims not to modify the existing code, but proposes a layer of meta information over the initial code. This layer uses the paradigm of fluxion which we define in section 2, and will be at the core of our proposition of automation, described section 3. Section 5, we link our work with related works. Finally, we conclude this paper in section 6.

2. FLUXIONNAL EXECUTION MODEL

2.1 Fluxions

The fluxionnal 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 contained 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 composed of the name of the recipient fluxion, a body, and are carried by a messaging system. While processing a message, the fluxion modifies its context, and sends back messages on its output streams. The fluxion's execution context is defined as the set of state variables whose the fluxion depends on, between two rounds of execution.

The fluxions make up a chain of processing binded by data streams. All these chains make up a directed graph, managed by the messaging system.

2.2 Messaging system

The messaging system is the core of our fluxionnal execution model. It carries messages along stream, and invokes fluxion at a message reception.

It is built around a message queue. Each message is processed one after another by invocation of the recipient fluxion. Using a message queue allows to execute multiple processing chain fairly and concurrently, without difference in scheduling local messages, or network messages. The life cycle of a fluxionnal application is pictured on figure 1.

The messaging system needs every fluxion to be registered. This registration matchs a processing function with a unique name and an initial execution context. The messaging system carries messages streams based on the names of the recipients fluxions. That's why two fluxions with the same name would lead the system in a conflicting situation. The regis-

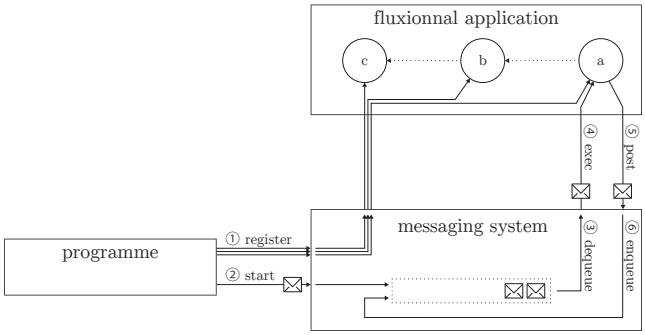

Figure 1: Messaging system details

tration is done using the function `register(<nom>, <fn>, <context>)`, step ① on figure 1.

To trigger a fluxions chain, a message is sent using `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 `post(<msg>)`, step ⑤, to be enqueud in the system, ⑥. The system loops through steps ③ and ④ until the queue is empty.

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

Algorithm 1 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
```

Algorithm 2 Message queue walking algorithm

```
function loopMessage()
    while msg presents in msgQueue do
        msg ← dequeue()                      ▷ ③
        ProcessMsg(msg)
    end while
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[8] 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 replies the result of the processing chain 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 fluxionnal system.

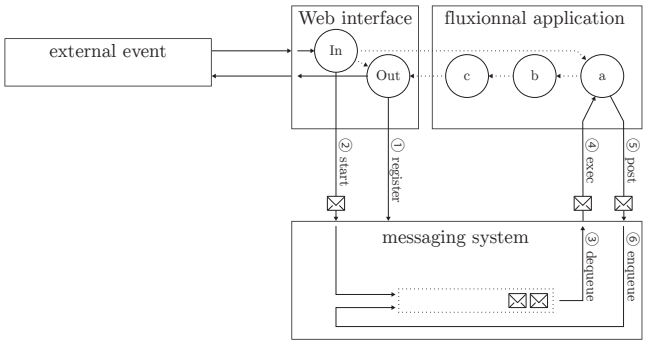

Figure 2: Fluxionnal application with web interface

2.4 Service example

In order to picture the fluxionnal 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.

The initial version of this service could look like listing 1.

```
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: Initial service

In listing 1, 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 fluxionnal system.
- The `reply` function, line 5 to 11, contains the logic we want to express in the fluxionnal processing chain.
- The two methods `get` and `send`, respectively line 5 and 10, interface the logic with the external interface. The hidden processing chain is : `get` → `reply` → `send`

This minimal service is transformed with our automatic tool into the Figure 3 fluxions chain.

Figure 3, circles 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

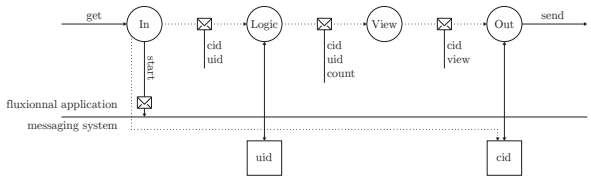

Figure 3: Count service fluxions chain

associated to the client connexion the last fluxion redirects the answer to. The `cid` tags the request and is transmitted all the way long through the flow. Each fluxion propagates the necessary values from one fluxion to the other exclusively within messages. Horizontal dashed lines show message virtual transmission between fluxion although they all go through the messaging system.

Listing 2 describes this counting service in our fluxionnal language. This new language brings a stricter segmentation than the initial code by allowing the developer only to define and register fluxions. And so, it allows an additional system to optimize how the system is organized on different physical machines according to the cost of fluxions' streams and processing. A fluxion is defined by a name, and a list of destination 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 object linked to 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: Fluxionnal sample

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

- The `logic` fluxion is the first to receive the client 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, push this count inside the message, and relay it the next fluxion.
- The `view` fluxion receives this message, formats it as the user will view it, and relay it to the output fluxion.

We use this interface to develop web services using the fluxionnal 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. COMPILER

The fluxionnal execution model is a framework to build distributed web applications, no better than so many already out. The first section of this paper describe an execution model to run application in a distribution fashion. This section describe the method we developped to transform almost any web application to be compatible with this distributed exectuion model. A distributed system run in a parallel fashion by design. We describe a compiler capable to split a program into many parallelisable parts, and to turn these parts into a network capable to run on te execution model described in the previous secion.

Promises[11] and Futures are abstractions between an imperative, synchronous programmation style, and an asynchronous execution model. They transparently transform synchronous long waiting operations into asynchronous operations allowing some parts of an application to run in parallel.

In a synchronous execution, the requested operation block the main thread until the value is computed. While in an asynchronous execution, the requested operation run along the main thread until the value is needed. The promise make this two execution paths independent, thus can run in parallel, if one need the other, it will waits for it to finish. This allows every independent parts of the applicaiton tu run in parallel, waiting for one to finish only when needed, thus leaving the computation processor to compute the values.

We call rupture points, the points where the execution flow forks in two distinct and parallelisable paths.

Javascript is a functional and dynamically typed language initially introduced to handle user interactions within Web pages. While Javascript isn't natively event-based, the DOM used in Web pages is. The latter uses an event-loop to handle events happening on the Web page, and then triggers associated functions the developer provides.

More recently, *Node.js* used the same event-loop based structure, to propose a non-blocking, event-based Javascript execution environment, specifically adapted for real-time I/O intensive applications like Web services. Because of this event-loop based architecture, the I/O API *Node.js* provides is non-blocking and asynchronous. The invocation of any function from this API returns immediately not to block the execution with time consuming I/O operations. The developer provide an handler function as argument for this asynchronous function to invoke when the operation completes. This handler function is commonly named a callback, *Node.js* uses the convention to place the callback as the last parameter. The *Node.js* event-loop receives and gathers every I/O event, waiting its turn in the loop to invoke the associated callback.

In most imperative languages the execution is synchronous by default, while special libraries handle parallelism, some using asynchronism like Promises and Futures cited above. However, Node.js imposes natively both synchronous and asynchronous paradigms. The asynchronicity of I/O Operations in *Node.js* naturally splits the execution along two execution paths. In *Node.js*, a rupture point is an asynchronous function call, defined by these two properties :

— the function is asynchronous
— the function is of higher order

The two distinct execution paths are the synchronous following instructions after the asynchronous function call, and the callback after the asynchronous operation is completed. A rupture point marks out two parallelisables parts of a web application. One of the compiler step, the pruner, spot the rupture points, and split the application along them.

With Promises and Futures, the memory is already either shared, or distributed, and left as is. In *Node.js*, the memory is shared, and the execution model expect it to be shattered into the application parts. As long as the memory is shared, the application can't be distributed on multiple nodes. The compiler need to split the shared memory into the application parts.

In Javascript, a function define its own scope, child of the function's scope the child function is defined within. A child function can access every of its parent scope up to the root scope. The root scope is called global, or window, wether you are in node.js or a browser.

In Javascript, the memory is enclosed in nested or chained scopes. Each function create a new scope containing variables local to itself. This scope is chained to the scope of the parent function, so that the child function can access variables in the scope of the parent function. Callbacks defined inside a scope can access the same scope as the calling function, allowing them to share variables.

Rupture points define application parts along scopes frontier, so a scope is never shared between two application parts. However, parent and child scopes might be separated in two application parts. If the two scopes don't share the same memory, variables from the parent are unavailable for the child. Another compiler step, the linker, understand and resolve dependenciy conflicts between the distributed functions scopes.

In the next subsections, we describe the different compilation steps. The compiler uses program from the community, they are described in the first subsection along with the first compilation step. Then, we describe the two following, important compilation steps : the *pruner* breaks a program into many independent parts and the *linker* resolves inconsistencies in the shattered functions scopes.

3.1 Common tools : parser and code generation

Parsing code and generating code back are common tasks. There exist projects to fulfill these tasks, like *Esprima* and *Acorn*, two Javascript parser. For this compiler we use a serie of tool written by Ariya Hidayat and Yusuke Suzuki for the projects *Esprima* and *Esmangle*. These tools follow the specification for an intermediate representation of the Javascript source code from the Mozilla Javascript Parser API : the Abstract Syntax Tree (AST) [2]. This structured representation breaks the source into a tree of nodes, each representing a construct from the source, like an operation or

2. https ://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Parser_API

an identifier. It can be traversed and allow easy modification of its structure, without the risk of errors involved by direct source manipulation.

An example node in the AST is :

```
CallExpression {
    type: "CallExpression";
    callee: <Expression>;
    arguments: [ <Expression> ];
}
```

Listing 3: Example of an AST node

The compiler uses *Esprima* to parse the source and generate the AST. It is the first compilation step. Thes AST can be traversed and explore with the use of *Estraverse*. *Escope* detects function scopes and variables declaration using the previously generated AST, and output an object to represent the organisation of these scopes inside the source code. One of the last compilation step is to produce a Javascript executable which uses the fluxionnal execution model. To generate this Javascript code, the compiler use Escodegen, to transform back the AST into Javascript source code.

3.2 Pruner : breaking the program

The pruner detects rupture points, breaks the application into parts along them, and map every function scope to its corresponding application part. Rupture points are indicated by calls of asynchronous function of higher order.

We distinguish specials rupture points indicated by asynchronous function handling series of external requests. Unlike basic rupture points indicating an asynchronous continuity in the execution flow, the reception of an external request indicate the start of a virtualy new execution flow. We distinguish these two types of rupture points to simplify later the dynamic analysis of system load. Each new incoming request represent an additional load for the system, and is the main unpredictable factor. While the incoming request load is taken into account, the load of every application part following in the continuity of the execution flow can be infered before run-time. The system load is only dependent at run time of the input in the system, everything else can be infered before run-time. We explain this point in details in the next section of this paper.

These two types of rupture points corresond to different asynchronous functions in the *Node.js* I/O API : the functions handling only one I/O event, or a bounded serie of I/O events, and the functions handling an unbounded serie of I/O events.

3.2.1 *One-time event*

Basic rupture points are indicated by asynchronous functions providing immediate I/O operation. Callbacks of these functions are invoked only once, and continue the execution after the completion of the I/O operation. Because of their asynchronism, these function calls mark the frontier between the current fluxion and the next one, inside a chain of fluxion. The compiler replaces the asynchronous function call by a call to a placeholder function.

3.2.2 *Series of events*

Special rupture points are indicated by asynchronous functions providing a callback for a series of future event. The handler of a network socket is called once for each incoming request. The callbacks of these functions indicate the input of a data stream in the program, and the beginning of a fluxions chain. As the callbacks mark the frontier between the current fluxion and the beginning fluxions chain, the compiler replaces the callback by a placeholder function starting the chain.

The compiler uses a dictionnary of asynchronous functions to detect rupture points during the static analysis. The compiler build this dictionnary when finding in the source, modules providing asynchronous function. Such modules are *Express* and *fs*. To find possible rupture points, the compiler tests the callee expression against this dictionnary for each callExpression node in the AST.

The compiler detects one rupture point for the following hello world web application.

```
var app = require('express');
app.get('/', function(req, res) {
  res.send("Hello World :)");
});
```

Listing 4: Hello World

There is in this program only two scopes, the global scope, and the scope of the anonymous function. These two scopes don't share any variable.

3.3 Linker : resolving dependencies

In real world application, unlike in the previous example, the previously split applicaiton parts share variables, like the following example. In this example web application, the two fluxions share a common variable : `rep`.

```
var app = require('express'),
    rep = "Hello World :)";
app.get('/', function(req, res) {
  res.send(rep);
});
```

Listing 5: Hello World with a shared variable

According to Brewer's theorem, formalized by Seth Gilbert and Nancy Lynch [**Gilbert2002**], a web application can only have two among the three options, Consistency, Availability, Partition tolerance. As Coda Hale explained in one of his blog post [3], network and node failures are unavoidable, a distributed system can't avoid to have failure. Mike Stonebraker explain in another blog post [4] that the trends is to make big data applications run on larger cluster of unreliable commodity machines. Partition tolerance can't be avoided, so the only possible trade off is between consistency and availability.

If this trends As the trend is to use larger and larger cluster of commodity machines, transactional systems trading

3. http ://codahale.com/you-cant-sacrifice-partition-tolerance/
4. http ://voltdb.com/blog/voltdb-products/clarifications-cap-theorem-and-data-related-errors/

availability for consistency, will become slower and slower. Choosing to sacrifice consistency might make more sense. It let design a lot more flexible solution, which might have a better chance to cope with this kind of highly distributed architecture.

There exist solution to minimize the impact of error due to inconsistency. Dynamo system : http ://s3.amazonaws.com/AllThingsDistributed/sosp/amazon-dynamo-sosp2007.pdf

While walking the AST, the compiler register and track every use of variables to determine which ones and what type of access the fluxion needs, and which scope they belong to. This set of variables needed by a fluxion represent its signature. For example, the signature of the fluxion created out of `asyncFn`, listing **??** include variables `b` and `c` as they are used in `callbackFn`, as well as `a` because it is used by the function call to `asyncFn`. To resolve these dependencies, during the execution, the parent function sends the signature of the next fluxion in the same message together with the result of the asynchronous operation. As the placeholder function call have the same scope than the asynchronous function call or callback it replaces, it is responsible for gathering the variables from the signature in a message along with the result of the operation and send it to the next fluxion. The placeholder function call replacing `asyncFn` after compilation of listing **??** is described in listing **??**, line **??**.

As fluxions are chained one after another, a fluxion must provide every dependency for the next one, even if some of this dependencies miss from its own scope or signature. These dependencies must be passed fluxion after fluxion from the producing fluxion, to the consuming fluxion. So, the message stream linking one fluxion to another includes the signature of the next fluxion as well as dependencies targeting downstream fluxions. The compiler has to resolve the content of these message streams beginning by the last fluxions and going upstream to the first ones. Figure 4 illustrate this principle : since fluxion C needs the variable z, fluxion B needs the variable z as well to pass it along to fluxion C.

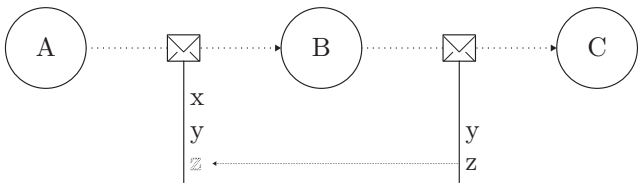

Figure 4: Fluxion C needs the variable z, so does fluxion B

3.4 Limitations

Leaving an asynchronous function call as is doesn't introduce bugs, however breaking a synchronous function by replacing its callback leads to bugs. To avoid introducing new bugs, it is important for the compiler to be able to distinguish between these synchronous and asynchronous functions.

Javascript is dynamically typed, if the index to access an array can't be resolved statically, then so do the type of the result. Some callbacks can't be resolved statically. For example, in listing 6, the function `myAsyncFn` is asynchronous and ask for a callback as parameter. The compiler would break the program along its call, however `event.type` is unresolvable statically, the compiler is unable to include the callback in the next fluxion. This structure might already be encapsulated inside a fluxion, and the callback might need variables from the scope of an upstream fluxion, but as the callback is unresolvable statically, it is impossible for the compiler to track them, and add these dependencies in the signature of the current fluxion. Even if the compiler leaves this structure as is, it introduce dependency bugs as the compiler is unable to resolve dependencies and generate accurate signatures. The compiler is currently unable to compile a program containing structures involving dynamic resolution like in listing 6.

```
1 myHandlers = [];
2 // ... definition of myHandlers
3 onEvent(function(event) {
4   myAsyncFn(myHandlers[event.type])
5 })
```

Listing 6: Example of an unresolvable callback

3.5 Futur Works

Even synchronous, the use of a callback by the `map` function indicate an independence between the callback and the main execution thread. For future improvements, we focus on studying these independences to allow the compiler to spot and break into fluxions these patterns of synchronous function call using callbacks.

For future improvements, we focus on a solution to dynamically compile fluxions and resolve dependencies, allowing to compile programs containing dynamic structures described in the last paragraph.

4. DYNAMIC ANALYSIS OF A PROGRAM

We distinguish between start and post to be able to analyze the total charge of the system, only by analyzing the flow of incoming requests.

Using fluid mechanics analogies.

5. RELATED WORKS

The first part of this work, the execution model, is partly inspired by some works on scalability for very large system, like MapReduce[5]. It also took inspiration from more recent work, like the Data Stream Management System (DSMS). Among the most known, we cited in the introduction Spark [18], MillWheel [1], Timestream [17] and Storm [12].

The idea to split a task into independent parts go back to the Actor's model[9] in 1973, and the first Functional programming Langage Lucid[3] in 1977 and all the following works on DataFlow leading up to Flow-Based programing (FBP)[13] and Functional Reactive Programming (FRP)[6]. Both FBP and FRP, recently got some attention in the Javascript community with respectively the projects *NoFlo*[14] and *Bacon.js*[15].

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 programing paradigm into a network of communicating parts known to have scalabitly advantages. There is some work on the transformation of a program into distributed parts[2], [16]. But our approach using callbacks in Javascript seems unexplored yet.

Our approach uses AST modification, as described in[10].

Obviously, our implementation is based on the work by Ryan Dahl : *Node.js*[4], as well as on one of the most known web framework available for *Node.js* : *Express*[7].

6. CONCLUSION

In this paper, we presented our work to enable a Javascript 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 stream 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 spot the asynchronous function calls and their callbacks, as indicators for a possible parallelism. This compilation tool allow to make 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 : *Proc. VLDB Endow. 6.11* (2013).

[2] M Amini. ■ Transformations de programme automatiques et source-à-source pour accélérateurs matériels de type GPU ■. In : (2012).

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

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

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

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

[7] *Express*.

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

[9] 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).

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

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

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

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

[14] *NoFlo*.

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

[16] E Petit. ■ Vers un partitionnement automatique d'applications en codelets spéculatifs pour les systèmes hétérogènes à mémoires distribuées ■. In : (2009).

[17] Z Qian, Y He, C Su, Z Wu et H Zhu. ■ Timestream : Reliable stream computation in the cloud ■. In : *Proc. 8th ACM Eur. Conf. Comput. Syst. (EuroSys '13)* (2013).

[18] M Zaharia et M Chowdhury. ■ Spark : cluster computing with working sets ■. In : *HotCloud'10 Proc. 2nd USENIX Conf. Hot Top. cloud Comput.* (2010).