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 4, we link our work with related works. Finally, we conclude this paper in section 5.

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-

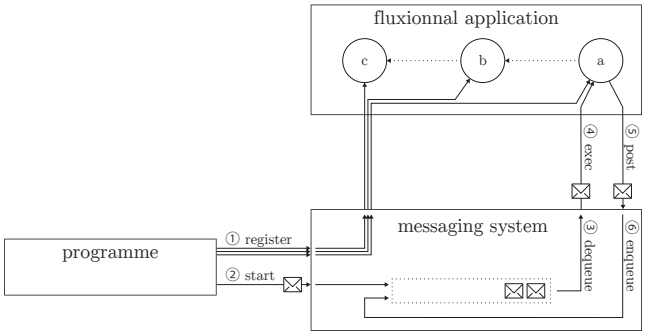

Figure 1: Messaging system details

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.

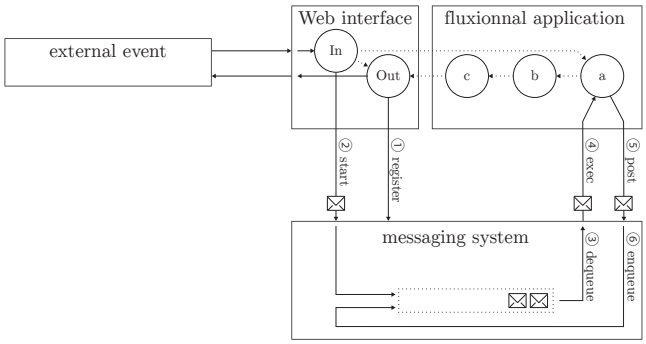

Figure 2: Fluxionnal application with web interface

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

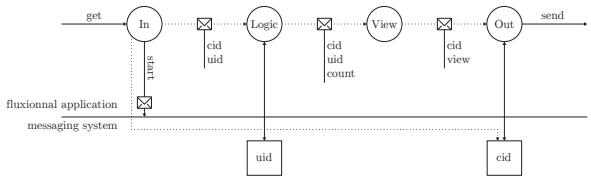

Figure 3: Count service fluxions chain

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

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. Javascript is not a pure functional language. As functions have access to the scope of their parents, multiple handlers can share a context. However, there is no possible concurrence between the handlers as there is only one execution thread.

More recently, *Node.js* used the same event-loop based structure, to propose a non-blocking, event-based 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 last argument provided by the developer to this function must be a handler function to invoke when the operation completes. This function is commonly named a callback. The *Node.js* event-loop receives and gathers every I/O event, waiting its turn in the loop to invoke the associated callback. Listing 3 illustrates the call of the asynchronous function `asyncFn` with the callback `callbackFn`.

```
1  function MyFn() {
2    var a = 1,                                          ▷ ①
3        b = 2,
4        c = 3;
5    asyncFn(a, b, function callbackFn(result) {
6      // process the result of asyncFn                  ▷ ③
7      // Use variables b, and c.
8    });
9    // result is not yet ready                          ▷ ②
10 }
```

Listing 3: Example of an asynchronous function call

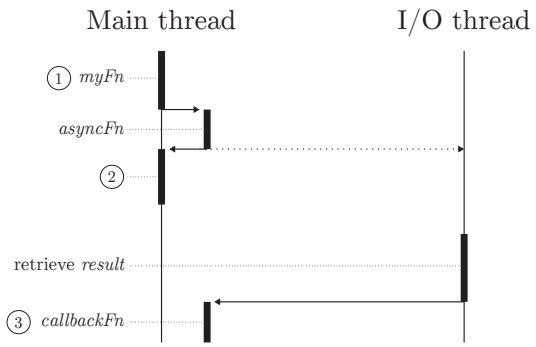

Figure 4: Illustration of listing 3

In most imperative languages the execution is synchronous by default, while special libraries handle parallelism, some using asynchronism and callbacks[11]. However, Node.js imposes natively both synchronous and asynchronous paradigms, as seen above, listing 3. The invocation of `asyncFn`, ③, and the code following the invocation, ②, are independent from each other because the execution order isn't deterministic. This invocation of an asynchronous function virtually splits the execution and create a possible parallelism between these two execution paths. Because of the parallelism implied by this independence, we think it is possible to isolate these parts enough to encapsulate them in fluxions, and migrate their execution onto different machines. Figure 5 illustrates the two isolated parts encapsulated in fluxions from listing 3.

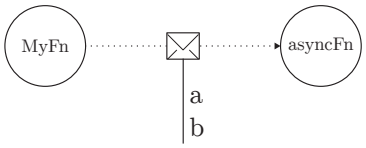

Figure 5: Illustration of listing 3 broken into fluxions

In the next subsections, we describe our work on a compiler to break a program into many independent parts and enclose them in fluxion. We then address the limitations of this method, and possible future improvements.

3.1 Breaking the program

As asynchronous functions indicates frontier along which the program is breakable into independent parts, the very first step of the compilation process is to spot them by analyze of the source code. We statically analyze the source code using an intermediate representation of it : the Mozilla Javascript 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 an identifier. It can be traversed and allow easy modification of its structure, without the risk of errors involved by direct source manipulation. In an AST, the node of a function call is of type *CallExpression*, and contains a reference to the callee and an array of the arguments. As an asynchronous function call marks the frontier between two fluxions, most modifications required to split the execution paths happen around the nodes of this type.

We distinguish two types of asynchronous functions in the *Node.js* I/O API : the functions handling a series of I/O events, and the functions handling only one I/O event. We details in the next paragraphs how the compiler modify these two types of asynchronous functions. We explain in section 3.3 why the compiler needs a list of asynchronous function with their type to distinguish between the two types.

3.1.1 Series of events

Some asynchronous functions provide a callback for a series of future event. Like the function `app.get` from listing 1, line 5. 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. This placeholder function uses the function `start(<msg)` provided by the fluxionnal execution model described section 2. The placeholder function is detailed later, section 3.2.

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

Algorithm 3 details the `StartCallExpression` function called while walking the AST to do the callback replacement. On line number 5, it creates the first fluxion of the chain to include the callback previously parsed. On line number 7, the callback is replaced with a placeholder function.

Algorithm 3 Algorithm to replace callback from start function call

```
1:  function StartCallExpression()
2:      for argument in arguments do
3:          if argument is a Function then
4:              argument.name + uniqueidentifier → name
5:              Create fluxion name parsing argument
6:              Register output to name
7:              placeholder → argument
8:          end if
9:      end for
10: end function
```

3.1.2 *One-time event*

The other type of asynchronous functions provides 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. This placeholder function uses the function `post(<msg>)` provided by the fluxionnal execution model described section 2. The placeholder function is detailed later, section 3.2.

Algorithm 4 details how the `PostCallExpression` function replace the asynchronous function call. Line number 4, the compiler parses the current node containing the *CallExpression* to create the next fluxion. It creates the next fluxion enclosing the asynchronous function call into a simple function together with the callback declaration. Callbacks declared *in situ* in the arguments are trivially spotted. However to accurately detect every callbacks, the compiler needs to track and resolve every identifier used in the program to identify their type. We address the problem of unresolvable callbacks in section 3.3. On line number 6, a placeholder function call replace the asynchronous function call. This placeholder is invoked instead of the asynchronous function and sends a message to the next fluxion.

Algorithm 4 Algorithm to replace post function call

```
1: function PostCallExpression(node)
2:     if node.callee is asynchronous then
3:         node.callee.name + uniqueidentifier → name
4:         Create fluxion name parsing node
5:         Register output to name
6:         placeholder → node
7:     end if
8: end function
```

These two compilation algorithms break the program into fluxions, and register the inputs and outputs of each to build a network map of the program. The compiler uses this map to resolve the remaining dependencies between this parts. We address in the next section this dependencies problem, as well as the placeholder function.

3.2 Scope sharing and Concurrency

In Javascript, 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 the scope of the parent. Callbacks defined inside a scope can access the same scope as the calling function, allowing them to share variables even while their executions are asynchronous. As an example, in listing 3, `callbackFn` is defined inside `MyFn`, so it can access `MyFn`'s scope containing variables `a`, `b` and `c`. By isolating a callback, we break its dependencies with its parent scope. These needs to be resolved.

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 3 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 3 is described in listing 4, line 7.

```
1 function MyFn() {
2   var a = 1,
3       b = 2,
4       c = 3;
5
6   // Placeholder for asyncFn-uid
7   flx.post(flx.m("asyncFn-uid", {a, b, c}));
8 }
```

Listing 4: Example of a placeholder function call

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 6 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.

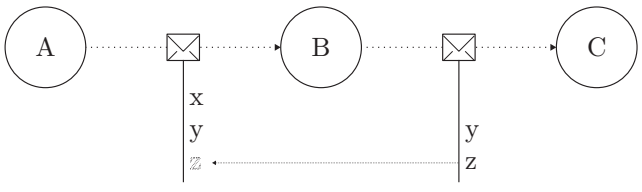

Figure 6: Fluxion C needs the variable z, so does fluxion B

3.3 Limitations

Node.js uses the convention to place the callback as the last parameter. While most of the *Node.js* API is asynchronous and uses this convention, Javascript allows to pass a function as an argument whether the callee is asynchronous or not. For example, functions from the `Array` prototype ask functions as parameters for a behavior to call on each iteration over the array. Listing 5 presents an example of this structure. This iteration is synchronous, so the function passed as an argument, ①, and the code following the function call, ②, are somehow dependent. 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.

```
1  my_modified_array = my_array.map(function(element)
       {
2    // Modifying element                          ▷ ①
3  })
4  // Following code                               ▷ ②
```

Listing 5: Example of a synchronous function using a callback

The compiler is unable to distinguish by itself between the synchronous and asynchronous functions, or between the two types of asynchronous functions described in section 3.1.1 and 3.1.2. We propose to provide for the compiler a list of every asynchronous function likely to be broken into fluxions, with their type specified. With this list, the compiler is able to modify asynchronous functions accordingly to their type, while leaving untouched every other function.

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.

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 unresolved, 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

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.

3.4 Compilation example

As of today our work on the compiler is still incomplete, there is inconsistencies between the high-level language detailed in section 2 and the compilation results. For documentation purposes, this section details the compiler current state of progress.

Listing 7 is a simplified version of the example listing 1 in section 2. We use this simplified version as a test for our compiler, as it is yet unable to handle dynamic resolution, like used with the object `count` line 6 in listing 1. The result of this compilation is listing 8. In the compilation result, the fluxion *id-1000* should hold a context containing the variable `count`, like the object `uid` in the context of fluxion *logic*, listing 2. But our compiler is yet unable to correctly resolve dependencies and fluxion contexts, so this counter service is unable to increment visits.

```
1 var app = require('express')();
2 var count = 0;
3
4 app.get("/:id", function id(req, res){
5   count = count + 1;
6   res.send(count);
7 });
8
9 app.listen(8080);
```

Listing 7: Simplified version of the initial service

```
1  // Main >> id
2    var flx = require('fluxion');
3    var app = require('express')();
4    var count = 0;
5
6    app.get("/:id"n function placeholder(req, res) {
7      flx.start(flx.message("id-1000", {
8        res: res,
9        count: count
10     }))
11   })
12
13   app.listen(8080);
14
15 // id-1000
16   flx.register("id-1000", function id-1000(msg){
17     res.send(msg.count = msg.count + 1);
18   })
```

Listing 8: Compilation result of code listing 7

4. 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].

5. 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).