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

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 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
1. flux in french

Compilation

Keywords
Flow programming, Web, Javascript

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 [23], MillWheel [1], Timestream [22] and Storm [17]. 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-
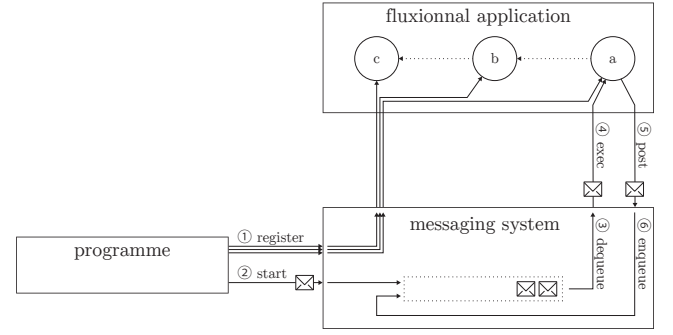

Figure 1: Messaging system details

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**
        $fluxion \leftarrow lookup(dest)$
        $message \leftarrow$ 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**
        $msg \leftarrow$ 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[9] 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.
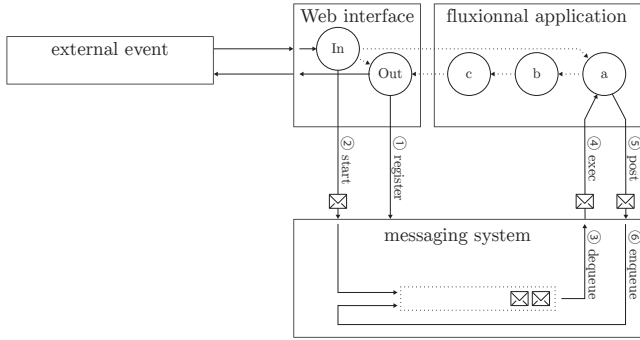

Figure 2: Fluxionnal application with web interface

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
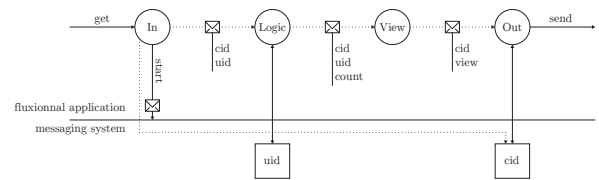

Figure 3: Count service fluxions chain

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 first section of this paper describe the fluxionnal execution model, a framework to run web application in a distributed environment. This section explains a method we developed to transform a subset of classic web application to be compliant with the execution model previously described. This transformation unveils two problems due to the differences between a web application and the execution model. In the first section, a distributed system is defined by the parallel execution of its parts, and the distribution of its memory. A classic web application is not composed of many independent parts, and relies on a central memory. The problems are to parallelize the execution of a mono-thread application into many parts, and to distribute the central memory among these independent parts. **TODO** does it need a definition of the classic web application ? if so, should be in the introduction, not here We describe a compiler as a solution to this problems, hence capable to turn a classic web application into a scaling distributed system. **TODO** scaling is a bold claim, need some background

The parallelization of a program designed for a mono thread architecture is a trending problem since the multiplication of the number of cores available on a machine. **TODO** references, see parallelization in my biblio It would allow developers to continue design application the same way they used to, while leveraging the performance of a multi-core architecture. From the sun programming guide [2], parallelism is *a condition that arises when at least two threads are executing simultaneously*, and concurrency is *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 during a communication, when a point doesn't wait for the answer to his request to continue processing an independent thread of execution. In an asynchronous execution, the requested operation run along the main thread until the value is needed, releasing the main thread from waiting the operation to complete. Promises[16] and Futures[? ?] are abstractions from an imperative, synchronous programing style, to an asynchronous execution model. They transform synchronous, long waiting operations - like remote procedure calls or input output operations - into asynchronous operations. This asynchronism make the two execution paths independent, thus they can run in parallel until they are not independent enough - when one needs results from another. We call rupture points, points where the execution flow forks in two independent and parallel paths. These points mark out the limits between independent parts.

TODO next paragraph is a draft, needs rewrite 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. **TODO** libevent, nginx : cite papers **TODO** references to

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

interruptions 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 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.The *Node.js* event-loop receives and gathers every I/O event, waiting its turn in the loop to invoke the associated callback. *Node.js* imposes natively an asynchronous paradigms along with the classic synchronous execution flow. An asynchronous functions split the execution along two concurrent execution paths, the path independent from the asynchronous request, and the path handling the answer of the request. We defined rupture points in *Node.js* as an asynchronous function call using a callback mechanism to handle the result. The two distinct execution paths are the synchronous instructions following the asynchronous function call, and the callback. A rupture point marks out two independent parts of a web application. One of the compilation step, the analyzer, spot the rupture points for the another compilation step, the mapper, to split the application along them.

TODO References of solutions to split memory into distributed parts ? Parallelism is not sufficient for an application to be distributed, because of the central memory. Promises and Futures don't transform a central memory into a distributed memory. *Node.js* provide a central memory, while the execution model expect it to be distributed into the application parts. The compiler needs to split the shared memory into the application parts for the application to be compliant with the execution model previously described.

In Javascript, scopes are nested one in the other, the parent being the global scope. 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, up to the global scope. Callbacks defined inside a scope can access the same scope as the calling function, allowing them to share variables.

Rupture points are always situated along scopes limits. A scope is never shared between two application parts. However, a child scopes separated in another application parts than its parent can't access the scopes it expects. 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 dependencies conflicts between the distributed functions scopes.

In the next subsections, we describe the compilation steps. Some part of the compilation chain are tools from the community, they are described in the first subsection along with the trivial compilation step. Then, we describe three important compilation steps relevant to the two previously described problems. The *analyzer* detects rupture points in an application, later for the *mapper* to break the application into many independent parts along the rupture points, then the *linker* resolves inconsistencies between the shattered me-

mory scopes.

3.1 Common tools : parser and code generation

The first compilation step is to parse the source code taken as input. The last compilation step is to output Javascript code either as a Javascript source code to run on the fluxionnal execution model, or as code in another high-level langage describing the fluxions and their content. Parsing code and generating code back are common tasks. There exist community 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) [3]. 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.

An example node in the AST is :

```
1 CallExpression {
2     type: "CallExpression";
3     callee: <Expression>;
4     arguments: [ <Expression> ];
5 }
```

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 organization 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 Analyzer : spotting the rupture points

The analyzer detects rupture points. A rupture point is composed of an asynchronous function, and a callback. We define in this section what a rupture point is, and how we detect them.

3.2.1 Rupture points

Rupture points are an asynchronous continuity in the execution flow, indicated by calls of asynchronous function with a callback in the parameters. We distinguish specials rupture points indicated by asynchronous functions handling series of external requests, from basic rupture points indicated by asynchronous functions handling only a one time event inside of the application. We distinguish these two types of rupture points to simplify later the dynamic analysis of the system load. The special rupture points are the entry point for the flow of request, and so is a point of choice to measure

---

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

the load during run-time. We explain this point in details in the next section of this paper. TODO *if we have the time to make the analysis* 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 series of I/O events, and the functions handling an unbounded serie of I/O events.

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 application part and the next one, inside a chain of fluxion. The rupture point is placed before the call to the asynchronous function, but after the resolution of the arguments. The javascript middleware printer replaces the asynchronous function call by a call to a placeholder function.

Series of events. Special rupture points are indicated by asynchronous functions providing a callback for a series of future event. For example, 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.

3.2.2 Detection

To detect a rupture point, it require to detect successfully the two components : the asynchronous function and the callback function.

Asynchronous functions. Asynchronous functions are detected from their call name, linked from the module exposing them. In Javascript, modules are included and stored in variables via the call to the `require` function. The name of the variable holding the module is specified by the developer, so the only constant hold onto the asynchronous function is the call to the `require` function The compiler uses a dictionnary of known asynchronous functions in modules to detect rupture points during the static analysis. Such modules are *Express* and *fs*. To find possible rupture points, the compiler tests the callee expression against the variable known for holding these modules.

To accurately detect the asynchronous function, we need to track the variables holding asynchronous functions. The more accurately we can track changes in this variables, the more rupture points we can find, the better the application can be distributed. This detection is done by traversing the AST, to find the node responsible for the assignment of modules in variables. More advanced tracking technique could be used later to improve the efficiency of the compiler.

Callback function. To detect a callback function, we track every argument of an asynchronous function call to test if it is a function. Some callback functions are declared *in situ*, and are trivially detected. For every other variable identifier, we track the declaration to recursively test the initialization value. This method is not exhaustive, but it is a first step to increase the detection of rupture points.

3.3 Mapper : correspondence between function scope and fluxion

The mapper breaks the program along rupture points to create application parts, later for these parts to be enclosed in fluxions as described in the previous section. The two types of rupture points break differently. The basic rupture point is an asynchronous continuity in the execution flow, it breaks before the asynchronous call, but after the resolution of the arguments. TODO Why does it makes sense to include the asynchronous call in the next fluxion?

The special rupture point is an entry point into the system, making the system reacts for every request. The asynchronous function of this type of rupture point is called one time, while the callback handling the event is triggered for each event. That means the application parts containing the asynchronous function is executed one time, while the parts containing the callback is executed multiple times. They cannot be in the same application parts, and be broken apart into two fluxions.

3.4 Linker : resolving dependencies

According to Brewer's theorem, formalized by Seth Gilbert and Nancy Lynch [11], a distributed application can only have two among the three options, Consistency, Availability, Partition tolerance. As Coda Hale explained in one of his blog post [4], network and node failures are unavoidable, a distributed system can't avoid to have failure. Mike Stonebraker explain in another blog post [5] 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. These two tradeoff are defined in the literature as ACID (Atomicity, Consistency, Isolation, Durability) for consistency over availability, and BASE (Basically Available, Soft state, Eventual consistency) for availability over consistency.

If this trends is verified, transactional systems trading off availability for consistency might suffer from slower and slower response time, because of the rigidity of the requirements. Choosing to sacrifice consistency might improve performance, because of the flexibility allowing to cope with problems emerging from highly distributed architecture, without durable inconsistency in the data because they are avoided by different synchronization techniques. Amazon Dynamo system[6] already demonstrated that these techniques can successfully be used to build a distributed system favoring availability without loosing consistency. In the article

---

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

presenting Amazon Dynamo system, the authors said that *experience at Amazon has shown that data stores that provide ACID guarantees tend to have poor availability. This has been widely acknowledged by both the industry and academia* This citation take reference from another paper[10] where Eric Brewer is one of the author.

From the Amazon Dynamo system paper : "Dynamo uses a synthesis of well known techniques to achieve scalability and availability : Data is partitioned and replicated using consistent hashing[14], and consistency is facilitated by object versioning[15]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs a gossip based distributed failure detection and membership protocol. Dynamo is a completely decentralized system with minimal need for manual administration. Storage nodes can be added and removed from Dynamo without requiring any manual partitioning or redistribution".

We use these papers to justify our choice for the linker to favor availability over consistency. We believe we can make use of this well-known technique in the execution model. As these techniques are already well-known, we focused our work on different problematics.

In the following section, we explain the different cases of inconsistency emerging from the partitioning of a central memory. The linker analyzes how scopes are distributed among the application parts, and which variables are distributed on multiple application parts. *Escope* gives the compiler informations about the references of every variables in the application. Among these references, we track the modifications of variables.

To resolve the dependencies in a fluxion's signature, the compiler uses different techniques.

Signature If the variable is needed read-only by a downstream fluxion, it is sent following the message chain, from the upstream to the downstream fluxion.

Scope If the variable is modified by one fluxion, the variable is placed in the own memory of the application part, called scope.

Sync If the variable is modified by at least two fluxion, the variable is synchronized between these fluxions.

3.4.1 Signature

The variables needed for read-only access by one of the scope of a fluxion and modified by another fluxion represent its signature. The signature of a fluxion is added in the message body between two fluxions. The code inside the application part is modified to shift the corresponding references to point to the message.

3.4.2 Scope

The own memory of an application parts holds the variables needed for modification, and never modified in another application part. This memory is stored in the scope of the fluxion encapsulating the application part. If one of this variables is needed for read by another fluxion downstream, this variable become part of the signature of the downstream fluxion.

3.4.3   Sync

If a variable is needed for modification by more than one fluxion, this variable needs to be synchronised between the fluxions' scopes. In this context, we use techniques described earlier to reach eventual consistency.

3.5   Printers

The last compilation step is to print the descriptive object of the applications parts into different form. We developed three different printers. One to describe exhaustively the application parts in a new high-level language we designed. One to output a javascript executable file compliant with the distributed execution model described in the previous section. Finally, one to describe concisely the applications parts and the message passing into a graph.

3.5.1   Fluxionnal high level language printer

The Fluxionnal high level language describe the application into fluxion. Each application part is preceded by its declaration : its name, the next downstream fluxion it sends message to, and the variables needed from other fluxions. Following is the javascript code of the application part.

3.5.2   Fluxionnal execution model printer

This printer encapsulate the previously prepared application parts into execution ready fluxions.

To hold heterogeneous pieces of code in a single container like a fluxion, this printer put some glue code in between. This glue code is mainly trivial.

We call the code provided modifying the execution context using the method `apply`. It execute the application parts processing function within an artificially similar context to the one expected in the original application.

There is limitations in this method. The context is not identically reproduced. For example, the `caller` variable is not reproduced. Programs relying on this variable might contains bugs after compilation.

When a variable is modified by more than one application part, the glue code synchronizes it between them. There is two part in this synchronization, the sending of the modified variable, and the reception and actualization of the modified variable.

The first part is a message sent from the fluxion modifying the variable at the end of its execution, after the processing function executed. When a fluxion receives a message updating a variable, it just update the variable, and does not execute the processing function, as there is no message to process.

3.6   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 4, 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 4.

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

Listing 4: Example of an unresolvable callback

3.7   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 [23], MillWheel [1], Timestream [22] and Storm [17].

The idea to split a task into independent parts go back to the Actor's model[12] 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)[18] and Functional Reactive Programming (FRP)[7]. Both FBP and FRP, recently got some attention in the Javascript community with respectively the projects *NoFlo*[19] and *Bacon.js*[20].

The first part of our work stands upon these thorough studies, however, we are taking a new approach on the second part of our work, to transform the sequential programing paradigm into a network of communicating parts known to have scalabilty advantages. There is some work on the transformation of a program into distributed parts[2], [21]. But our approach using callbacks in Javascript seems unexplored yet.

Our approach uses AST modification, as described in[13].

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*[8].

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] G DeCandia et D Hastorun. ■ Dynamo : amazon's highly available key-value store ■. In : *ACM SIGOPS . . .* (2007).

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

[8] *Express*.

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

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

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

[12] 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).

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

[14] D Karger, E Lehman et T Leighton. ■ Consistent hashing and random trees : Distributed caching protocols for relieving hot spots on the World Wide Web ■. In : *Proceedings of the . . .* (1997).

[15] L Lamport. ■ Time, clocks, and the ordering of events in a distributed system ■. In : *Communications of the ACM* (1978).

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

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

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

[19] *NoFlo*.

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

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

[22] 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).

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