Automatic scalability for web applications

How to abstract scalabilty constraints from the developer
Industrial Paper

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

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

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

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

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

ABSTRACT
The audience's growth of a web application is very uncertain, it can increase and decrease in a matter of hours if not minutes. This uncertain growth often leads the 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 to express 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.

1. flux in french

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

General Terms
Compilation

Keywords
Flow programming, Web, Javascript

1. INTRODUCTION
The growth of web platforms is partially caused by Internet's capacity to stimulate services development, allowing very quick releases 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. It continues until the amount of data to treat requires the development team to use a more efficient processing model. Many of the most efficient models split the system into parts to reduce their coupling and migrate them to more resourceful environment. They are also based on a cluster of commodity machines[9] to allow incremental scalability. The Staged Event-driven Architecture (SEDA)[19] and MapReduce [4] are 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 [20], MillWheel [1], Timestream [18] and Storm [14]. However these tools propose specific interfaces and languages, generally different than the tools used in the early step of a project. 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. Therfore, these modifications imply the development team to take risks without adding concrete value to the service.

We propose a tool to free the development team of a technological shift wiping the code base, while automating this shift at a lower level to enable scalability. 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. We focus on 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 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 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 carried by a distributed messaging system. They are composed of the name of the recipient fluxion and a body. 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 the other 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.

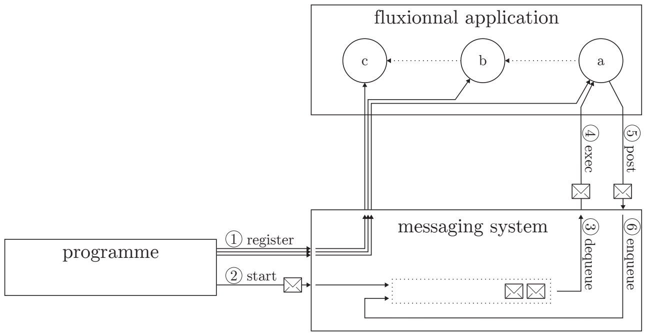

Figure 1: Messaging system details

The messaging system needs every fluxion to be registered. This registration matches 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. If two fluxions share a name, the messaging system would be in a conflicting situation. The registration is done using the function `register(<nom>, <fn>, <context>)`, step ① on figure 1.

To trigger a fluxions chain, a message is sent using `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

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 :

Algorithm 2 Message queue walking algorithm

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

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.

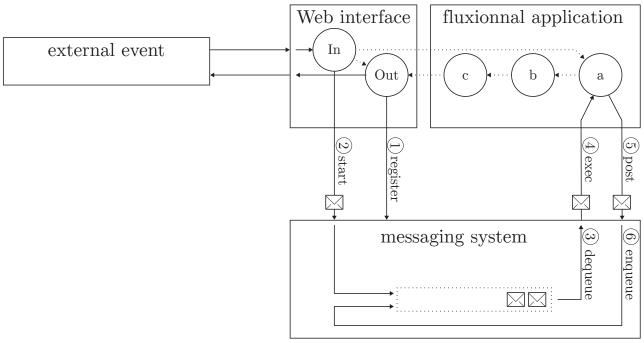

Figure 2: Fluxionnal application with web interface

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 manually into the Figure 3 fluxions chain. We expect a similar result with the compiler described in following section.

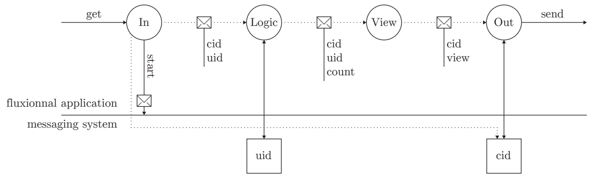

Figure 3: Count service fluxions chain

The circles in figure 3 represent registered fluxions. Envelope symbols represent exchanged messages between fluxions with the data transmitted from one fluxion to the other. Finally squares stored in the messaging system hold the *execution context* for the logic and **Out** fluxions. When a new `get` REST message is received at the **In** end point, a `start` message triggers the flow. Concurrently the **In** fluxion set a `cid` parameter to the **Out** fluxion execution context. This `cid` is associated to the client connexion 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.

```
use web

fluxion logic -> view
  this.uid[msg.uid] = this.uid[msg.uid] + 1 || 1
  msg.count = this.uid[msg.uid]
  post msg

fluxion view -> output
  msg.view = msg.uid + " connected " + msg.count + "
      times."
  msg.uid = undefined
  msg.count = undefined
  post msg

register logic, {uid: {}}
register view
```

```

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 on the developer.

3. DESIGNING A COMPILER FOR FLUXIONNAL COMPLIANCY

The section 2 of this paper describes the fluxionnal execution model, a framework to run web application in a distributed environment. This section explains the compiler we developed to transform a subset of classic web application to be compliant with the execution model previously described. Current Web applications are mostly written in Java. The langage proposes both data encapsulation and threading model, that ease the development of distributed applications. Yet, Java framework for developing efficient application are complex systems that impose new API (Servlet en référence) to the developers. Since 2009, *Node.js* provides a simple Javascript Web programming system. We focus on this promising environment for its initial simplicity and efficiency. We develop a compiler that transforms a simple javascript application into a fluxionnal system compliant to the architecture described in section 2. As javascript forbids user-space thread API, a javascript application is developed as a monothreaded application. Moreover, in Javascript the memory is hierarchical and the root scope may be accessed by any function, that leads to bad component isolation. Our compiler finds fluxions into most of Web based Javascript application. It finds component isolation through the the analyzer step and memory consistency through the linker step. We do not target all Javascript Web based application but if we are able to transform either some part of application or 50% of currently running applications without external developer help, we except a real execution gain in a cloud environment. The remaining of this section describes the two parts of the compiler. Section 3.1 explains how the *analyzer* detects rupture points in the web application to mark out the independent parts. Section 3.2 explains how the *linker* resolves the missing dependencies due to the distribution of the central memory.

3.1 Analyzer : execution parallelism

The parallelization of programs is a trending problem to leverage the multiple cores available on highly parallel architectures. 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 when a communication point continue processing an independent thread of execution while waiting for the answer to his request.

Promises[13], as well as *Node.js* callbacks, are abstractions that transform blocking synchronous operations into non-blocking asynchronous operations. This asynchronous operation run concurrently with the main thread, until the requested value is computed, and the main thread can continue the computation needing this value. This asynchronism splits the execution in two concurrent execution paths, one that needs the requested value and one that doesn't. We call rupture points, points where the execution flow forks in two concurrent paths due to asynchronism. These points mark out the limits between the independent parts of an application.

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

3.1.1 Rupture points

Rupture points represent a fork in the execution flow due to an asynchronous operation. They are composed of an asynchronous function, and a callback to handle the result of the operation. The two execution flows are the following instructions after the asynchronous function, and the instructions in the callback. Listing 3 is an example of rupture point in a simple application. A rupture point is an interface between two application parts.

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

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

There is two types of rupture points, basic and special, illustrated respectively in figure 4 and 5. In these figures, the two concurrent execution path distributed in two application parts are indicated by ① and ②.

Basic rupture points represent a simple continuity in the execution flow after a finite asynchronous operation, like reading a file in listing 3. The function calls from basic rupture points mark the interface between the current application part and the next one. This frontier is placed before the call to the asynchronous function, but after the resolution of the arguments. The result of this asynchronous operation probably being a voluminous object, this placement allow the asynchronous function call to occur in the same application

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

parts as the callback, avoiding the transfer of this voluminous result, as illustrated in figure 4.

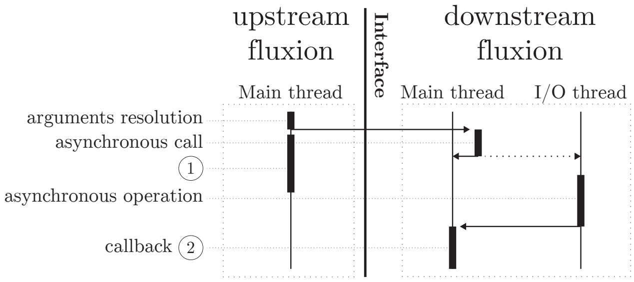

Figure 4: Basic rupture point interface. The interface is placed before the asynchronous operation, to avoid moving the result from one application part to another.

Special rupture points differ in that they are on the interface between the whole application and the outside, continuously handling incoming user requests, like `app.get()` in listing 4. The callbacks of these functions indicate the input of a data stream in the program, and the beginning of a chain of application parts following this stream. The *start* rupture points will later be used to monitor the load from incoming external requests. Because the asynchronous function is called only once, while the callback is triggered multiple times, this interface is placed between the two, as illustrated in figure 5.

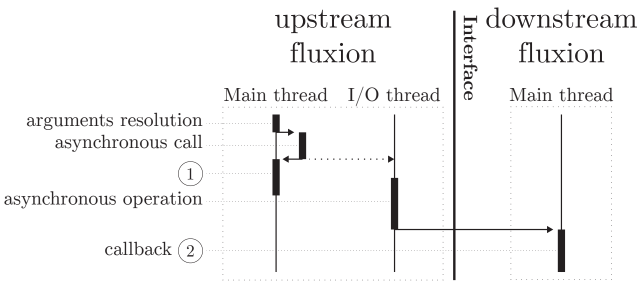

Figure 5: Special rupture point interface. The interface is placed after the asynchronous operation, for the downstream application part to be triggered for each event.

In the following, basic rupture points are called *post*, and special rupture points are called *start*

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

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

app.listen(8080);
console.log('server listening to 8080');
```

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

3.1.2 Detection

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

Asynchronous functions
The compiler is prebuilt with module names exposing asynchronous function, like the *express*, and the *fs* module in listing 4. To detect asynchronous calls, the compiler register variables holding such modules, building a dictionary of the asynchronous function call name. In listing 4, the compiler register both variables `app` and `fs`. When the compiler encounter a call expression, it compare its callee name to the dictionary to spot asynchronous functions.

Callback function
For each asynchronous call expression detected, the compiler test if an argument is a function. Some callback functions are declared *in situ*, and are trivially detected. For every other variable identifier, we track the declaration up to the initialization value to detect functions.

Variable tracking
To detect both the asynchronous function and the callback function, the compiler needs to statically track variables states. Missing rupture points by false negatives in the detection is sub-optimal, but false positives eventually produce an unstable compilation result. Therefore, the detection needs to be as accurate as possible to screen out false positives. We use a technique similar to a Program Dependency Graph (PDG)[7] to track changes in the value of variables.

3.2 Linker : memory distribution

Because of the central memory, parallelism is not sufficient for an application to be distributed. The compiler needs to distribute the memory into the application parts for the application to be compliant with the fluxionnal execution model. In Javascript, scopes are nested one in the other up to the all enclosing global scope. Each function creates a new scope containing variables local to itself, and is chained to the scope of the parent function. The child function can access variables in the scope of the parents functions, up to the global scope. Rupture points are appearing in between scopes, linking application parts in message streams. A rupture point placed between a child scopes and its parent break a chain of scope, and make the child unable to access its parent as expected, eventually leading the application to crash during execution. The linker analyzes how scopes are distributed among the application parts to detect and resolve conflicts between these scopes. Depending how the shared variable is used in the functions scopes, there are different situations to resolve. In the following, we explain the different cases of inconsistency emerging from the partitioning of a central memory.

3.2.1 Signature

The signature is a part of a message containing all the variables to send downstream. If a variable is needed for read-only access by at least one downstream application part, it is added to the signature of the rupture point. As fluxions are chained one after another, a fluxion must provide every dependency for the next one, even if some of this dependencies are not needed by the current application part. These dependencies must be passed fluxion after fluxion from the producing fluxion, to the consuming fluxion. The code inside the application part is modified for the signature's references to point to the message signature instead of the function scope.

3.2.2 Scope

The scope is the name given to the persisted memory of a fluxion. It holds the variables declared outside, but needed for modification in only one application parts. If one of this variables is needed for read by another application part downstream, this variable becomes part of the signature sent downstream. An example of such a variable is a request counter. Initialized to 0 in the global scope, the counter is incremented for each request. This counter would be in the scope of the application part handling requests reception, and sent downstream for visit metrics processing.

3.2.3 Sync

If a variable is needed for modification by more than one distributed application parts, this variable needs to be synchronized between the fluxions' scopes. Memory synchronization in a distributed system is a well-known problem. According to Brewer's theorem, formalized by Seth Gilbert and Nancy Lynch [10], an application can only have two among the three options, Consistency, Availability, Partition tolerance. Partition tolerance can't be avoided in a distributed system [3], so the only possible tradeoff is between consistency and availability. These two tradeoffs are defined in the literature as ACID (Atomicity, Consistency, Isolation, Durability) for consistency over availability, and BASE (Basically Available, Soft state, Eventual consistency)[9] for availability over consistency. The goal for this compiler is to be able to transform a subset of web application with a satisfying result. We choose not to sacrifice neither consistency nor availability, therefor we must sacrifice partition tolerance and keep these application parts together.

3.3 Compilation analyze

3.3.1 Example

```
var app = require('express')(),
    fs = require('fs');

app.get('/', function reply(req, res){
  fs.readFile(__filename, function(error, data) {
    res.send(data);
  });
});

if (!module.parent) {
    app.listen(8080);
    console.log('>> listening 8080');
}
```

Listing 5: Source of the example

```
flx anonymous-1011
-> null
  function (error, data) {
    this.res.send(data);
  }

flx reply-1010
-> anonymous-1011
  function reply(req, res) {
    (function placeholder(__filename, function (error, data) {
      this.res.send(data);
    }) {
      return flx.post(flx.m('anonymous-1011', {
        _args: arguments,
        _sign: {}
      }));
    });
  }

flx fs.readFile.js
>> reply-1010
  var app = require('express')(), fs = require('fs');
  app.get('/', function placeholder() {
    return flx.start(flx.m('reply-1010', {
      _args: arguments,
      _sign: {}
    }));
  });
  if (!module.parent) {
    app.listen(8080);
    console.log('>> listening 8080');
  }
```

Listing 6: Fluxionnal compilation result of the example

```
var flx = require('flx');
var app = require('express')(), fs = require('fs');
app.get('/', function placeholder() {
  return flx.start(flx.m('reply-1010', {
    _args: arguments,
    _sign: {}
  }));
});
if (!module.parent) {
  app.listen(8080);
  console.log('>> listening 8080');
}

// anonymous-1011 >> null

flx.register('anonymous-1011', function capsule(msg) {
  if (msg._update) {
    for (var i in msg._update) {
      this[i] = msg._update[i];
    }
  } else {
    fs.readFile(__filename, function (error, data) {
      this.res.send(data);
    }).apply(this, msg._args);
  }
}, { res: res });

// reply-1010 >> anonymous-1011

flx.register('reply-1010', function capsule(msg) {
  if (msg._update) {
    for (var i in msg._update) {
      this[i] = msg._update[i];
    }
  } else {
    (function reply(req, res) {
      (function placeholder(__filename, function (error, data) {
        this.res.send(data);
      }) {
        return flx.post(flx.m('anonymous-1011', {
          _args: arguments,
          _sign: {}
        }));
      });
    }.apply(this, msg._args));
  }
}, { fs: fs });
```

Listing 7: Execution model compliant compilation result of the example

3. http ://codahale.com/you-cant-sacrifice-partition-tolerance/

3.3.2 Limitations

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 the Staged Event-Driven Architecture (SEDA) by Matt Welsh[19] and later the MapReduce architecture[4]. It also took inspiration from more recent work following SEDA. Among the most known following works, we cited in the introduction Spark [20], MillWheel [1], Timestream [18] and Storm [14]. We cite the work on the BASE[9] and ACID data semantics, but the compiler doesn't use these works.

The idea to split a task into independent parts go back to the Actor's model[11] in 1973, and to Functional programming, like Lucid[2] in 1977 and all the following works on DataFlow leading up to Flow-Based programing (FBP)[15] and Functional Reactive Programming (FRP)[5]. Both FBP and FRP, recently got some attention in the Javascript community with respectively the projects *NoFlo*[16] and *Bacon.js*[17].

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. Promises[13] and Futures are related to our work as they are abstractions from a concurrent programing style, to an asynchronous and parallel execution model. However, our approach using Node.js asynchronicity via callbacks to automate this abstraction seems unexplored yet.

The compiler uses AST modification, as described in[12].

Obviously, our implementation is based on the work by Ryan Dahl : *Node.js*[3], as well as on one of the most known web framework available for *Node.js* : *Express*[6].

5. CONCLUSION

In this paper, we presented our work to enable a *Node.js* application to be dynamically and automatically scalable. The emerging design for an application to be scalable is to split it into parts to reduce coupling. From this insight, we designed an execution model for applications structured as a network of independent parts communicating by 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 rupture points, as indicators for a possible parallelism and memory distribution. 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] Edward A Ashcroft et William W Wadge. ■ Lucid, a nonprocedural language with iteration ■. In : *Commun. ACM* 20.7 (1977), p. 519–526.

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

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

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

[6] *Express.*

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

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

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

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

[11] 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).

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

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

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

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

[16] *NoFlo.*

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

[18] 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).

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

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