

Definition of a fluxionnal execution model

[Extended Abstract]

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

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 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. [biblio] 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 [Storm, MillWheel, Spark, TimeStream ...] 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 come 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, 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 are going to define in section 2, and will be at the core of our proposition of automation, described section 3. In Section 4, we evaluate our tool, and compare it to other solutions in term of performance, and development impact.

2. FLUXIONNALL 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

1. REST : **R**epresentational **S**tate **T**ransfer FIELDING et
TAYLOR 2002

```

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
10     + ' times.';
11   res.send(reply);
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 .

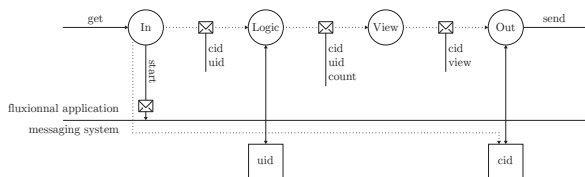


Figure 3: Count service fluxions chain

Figure 3, circles shows registered fluxions. We show exchanged messages between fluxions and data transmitted from one fluxion to the other. Finally squares stored in the messaging system hold the *execution context* for the logic and **Out** fluxionx. 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 may redirect 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 shows message virtual transmission between fluxion although they all go through the messaging system.

Listing 2 describes this counting service in our fluxionnal language. Conclusion, this new language brings a stricter segmentation than the initial code, and so allows an additional system to optimize how fluxions are organized on different physical machines according to the cost of the streams and their processing.

This high level language is not dynamic, and not typed.

You can register fluxion at any time, and the messages can contain any type or composition of types. **TODO** write a paragraph about the languages characteristics, a reserved subsection might be necessary, but maybe better placed in the next section, about transformation

```

1 use fluxion, 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   return msg
7
8 fluxion view >> output
9   msg.view = msg.uid + " connected " + msg.count + "
10     times."
11   msg.uid = undefined
12   msg.count = undefined
13   return msg
14
15 register logic, {uid: {}}
16 register view
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 services would need a more complex chain with logic distributed across multiples 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 receive this message, and format as the user will view it, and relay it the output fluxion.

Although, using a fluxionnal approach is a way to build an efficient distributed system we consider that the most important part of our work is to enable code transformation from a standard basic web approach to a flow of fluxions. We show now the main code transformation we propose.

3. PERFORMANCE EVALUATION

In this section we are going to evaluate the prototype we designed, and compare it to the performance of classical Javascript services.

We compare three distinct programs giving user the exact same service, but the implementation differs.

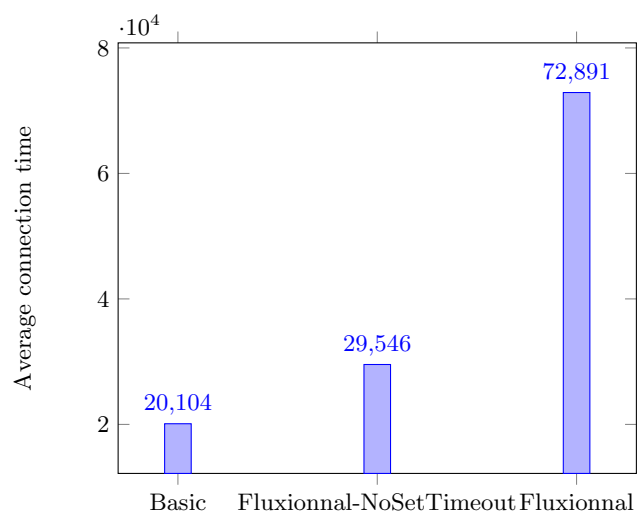
Basic The first one is a basic web service. It doesn't use fluxions.

Fluxionnal-NoSetTimeout The second one is a fluxionnal web service, except fluxions are chained without the possibilities of interleaving messages from network.

Fluxionnal The third one is a fluxionnal web service, but it probes messages from network between each fluxion execution.

On the implementation level, the third one uses **SetTimeout** while the second one don't. This difference make the second one incapable to handle fluxions chains longer than the function call stack size limit.

With these three differents programs, we want to highlight the advantages and drawbacks of the fluxionnal execution model.



■ Bench with 100 concurrent clients, connecting sequentially
100 times - Measured average response time : 20103.999100000005ms

■ Bench with 100 concurrent clients, connecting sequentially
100 times - Measured average response time : 29546.383199999997ms

■ Bench with 100 concurrent clients, connecting sequentially
100 times - Measured average response time : 72891.2149ms

APPENDIX

Références

- [1] RT FIELDING et RN TAYLOR. ■ Principled design of the modern Web architecture ■. In : *ACM Transactions on Internet Technology* (... (2002). URL : <http://dl.acm.org/citation.cfm?id=514185>.