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. 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 memory context during its execution.

Messages are composed of the name of the recipient fluxion, a body, and are carried by a messaging system. After processing a message, the fluxion modify its context, and then terminate its execution by sending a message on its output stream. Each fluxion sends back a unique message to one or many recipient. 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 build 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.

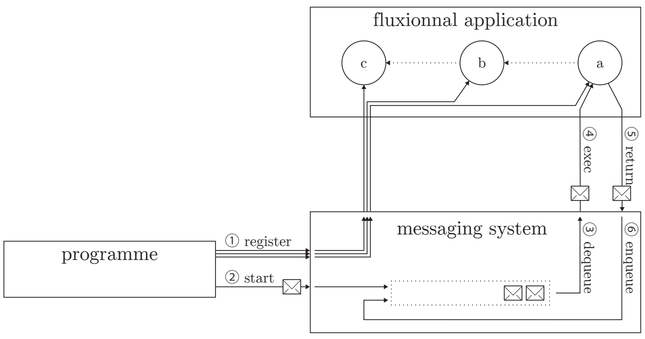

Figure 1: Messaging system details

The messaging system needs every fluxion to be registered. This registration match 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 registration is done using the function `register(<nom>, <fn>, <context>)`, step ① on figure 1.

To trigger a fluxions chain, a first `start(<msg>)` message is sent, 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 message resulting of this invocation is then enqueued, step ⑤ and ⑥. The system loops through steps ③ to ④ until the queue is empty.

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

Algorithm 1 Message queue 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[1] 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 specifics elements of the web interface inside the fluxionnal system.

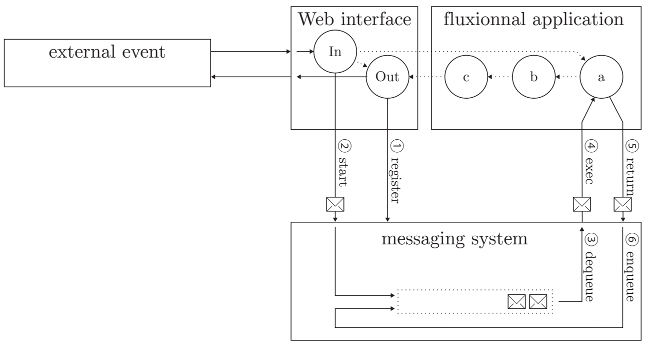

Figure 2: Fluxionnal application with web interface

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. REST : **Re**presentational **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
         + ' 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` $\rightarrow$ `reply` $\rightarrow$ `send`

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

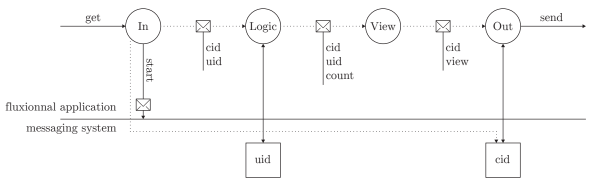

Figure 3: Count service fluxions chain

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

```
use fluxion, web

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

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

We uses this interface to develop web services using the fluxionnal execution model. We now evaluate this model against the basic one.

3. PERFORMANCE EVALUATION

We use the web service example described in the previous section, Figure 3, to compare the fluxionnal execution model and the basic Javascript implementation, listing 1. For this evaluation we developed several fluxionnal execution models using different instructions for chaining fluxions. Thus, through multiple implementation, we can compare the efficiency of the model itself, not of one implementation.

3.1 About the message queue

To implement the message queue in Javascript, we used already existing mechanisms as Javascript interpreters are oftnely packaged with an event loop. We distinguishe three different ways to queue and dequeue messages. These three implementations come the nature of the `node.js` event loop, and how Javascripts event are processed.

As shown in Figure 4 detailing the different layers of the event loop, the `node.js` event loop queues messages. These messages queues stacks of function, which queues functions. One can queue a message using `setTimeout`. Inside a message, one can queue a stack using the `node.js` instruction `process.nextTick`. Inside a stack, one can stack a function by calling directly this function.

Network messages come from I/O operations, thus, they are queued as a new message in the event loop. So to get network messages, the current message have to end, in order to get the next one, might it come from the network. However, it's well known that queueing messages in the event loop is far from efficient.

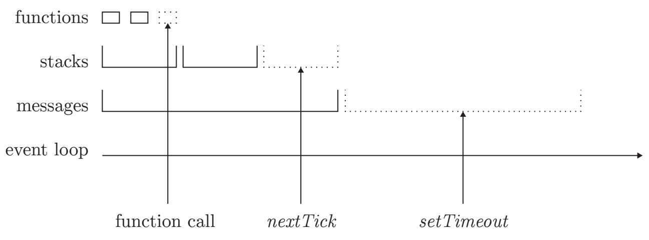

Figure 4: Javascript event loop details

3.2 Three different implementations

We tested our model with three different implementations.

Chain This implementation chains fluxions one after another by a direct function call. The whole fluxion chain is contain inside a same stack on Figure 4. It set the fluxions chain length maximum to the macimum function call stack size, and it's impossible to interleave messages from network in the middle of a fluxion chain.

NextTick This implementation uses the instruction `process.nextTick` to chain fluxions execution. This instruction add a function call at the end of the current execution. Two local fluxion processing chain could run concurrently, but it's only possible to probe network messages every n fluxions execution. By default n is set to 1000.

SetTimeout This implementation uses the instruction `setTimeout`. It probes network messages after every fluxion execution, thus networks messages can be interleaved between each local messages.

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

As we can see on Figure **??**, the difference between the basic implementation and the chained implementation is insignificant, we can conclude that splitting a web service into fluxions doesn't induce significative performance loss. And event the implementation using `nextTick` is almost as efficient as the basic implementation. However, the implementation using `setTimeout` is about 5 times less efficient than the basic implementation.

TODO 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.

4. TRANSFORMATION

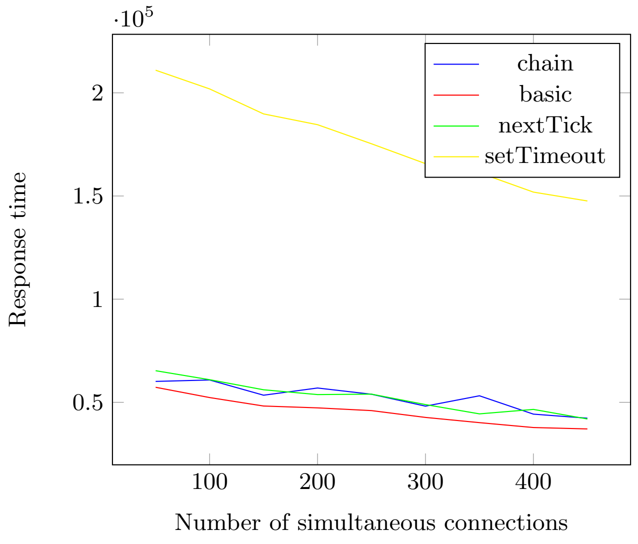

Figure 5: Response time for each implemetation in function of the number of simultaneous clients

We are detailling here, the necessary compilations steps to transform a program expressed with a classical approach, using Javascript, to the fluxionnal execution model described in the previous section.

As the fluxionnal execution model is written in Javascript, most of the initial code can be preserved. However, here is the list of the differences between the two models. These differences need a transformation from one model to another.

We consider a web service is in a subset of the classical programs, written in dynamic language. This subset implies the program to be structured in a fashion sucha as to chain processing sequentially, one after the other. TODO
The subset is badly described

4.1 Différences entre les deux modèles

4.1.1 Utilisation de la mémoire

Dans le modèle classique, la mémoire est centrale. Elle est cependant cloisonné en scopes de fonctions, et en contexte d'exécution. Une fonction n'as accès qu'à son scope, et au contexte dans lequel elle est exécutée.

Dans le modèle fluxionnel, la mémoire est décentralisé et encapsulé dans des scopes.

1. Une fonction n'utilise aucune mémoire, ou n'utilise aucune persistance entre ses invocations. -> pas de scope

```
        function(mon_argument) {
            return mon_argument + 3; // traitements
        }
```

Listing 3: Code fluxionnel

2. Une fonction utilise une closure comme scope et elle est la seule à accéder à cette closure. -> scope représentant la closure

```
        function(mon_scope) {
```

```
    // mon_scope accessible
    return function(mon_argument) {
        // mon_argument et mon_scope
            accessible
        return mon_scope + mon_argument; //
            traitements
    }
}
```

Listing 4: Code fluxionnel

3. Une fonction utilise une closure comme scope et elle n'est pas la seule à accéder à cette closure, cas de l'objet global. -> scope représentant la closure, partagé entre les fluxions.

```
var global ...

function fn1(mon_scope) {
    // global, mon_argument accessibles
    return mon_scope + global; // traitements
}

function fn2(mon_scope) {
    // global, mon_argument accessibles
    return mon_scope + global; // traitements
}
```

Listing 5: Code fluxionnel

4.1.2 Appel de fonction

Dans le modèle classique d'un service web, les fonctions de traitement sont appelé les unes après les autres en suivant un principe de chaîne de traitement.

Dans le modèle fluxionnel, le pointeur d'exécution est passé de manière événementiel, porté par le système de messagerie.

1. Une fonction appelle une autre fonction à la fin de son exécution -> la fluxion représentant la fonction appelante envoie un message à la fluxion représentant la fonction appelée.

```
function(req) {
    // traitements sur req
    return next(req);
}
```

Listing 6: Code fluxionnel

fluxion appelante -> fluxion appelée

2. Une fonction appelle une autre fonction avant la fin de son exécution. -> la fonction appelante est découpé en deux fluxions. la fluxion représentant la fonction appelée va servir d'intermédiaire entre ces deux fluxions.

fluxion appelante 1 -> fluxion appelée -> fluxion appelante 2

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.