Définition d'un modèle d'execution fluxionnel

[Extended Abstract]

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

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

Fabien Cellier
Worldline
107-109 boulevard Vivier Merle
69438 Lyon Cedex 03
fabien.cellier@worldline.com

Frédéric Oblé
Worldline
107-109 boulevard Vivier Merle
69438 Lyon Cedex 03
frederic.oble@worldline.com

ABSTRACT

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

General Terms
Compilation

Keywords
Liquid IT, Flow programming, Web

1. INTRODUCTION
La croissance des plateformes du Web est dû partiellement à la capacité d'Internet à favoriser le développement de services avec une mise en production minimale très rapide. En quelques heures, il est possible de mettre en ligne un produit fonctionnel afin de rassembler une première audience. *"Release early, release often"* est souvent entendu pour capter rapidement une communauté d'utilisateurs autour d'un projet open source.

Si le service répond correctement aux attentes, l'audience va probablement grossir au fur et à mesure que le service gagne en popularité. Pour pouvoir faire face à cette croissance, la quantité de ressources utilisé par le service augmente en conséquence, et il arrive un moment dans le développement du produit où la taille des données à traiter et la quantité de ressources nécessaires, imposent l'utilisation d'un modèle de traitement plus efficace. La plupart des modèles plus efficaces passent par une segmentation des échanges entre fonctions, en utilisant différents paradigmes de communication comme les approches *three-tiers*, les événements, les messages ou les flux, afin de réduire le couplage entre les parties et pouvoir les migrer vers des environnement de plus en plus puissants. [bilbio] Une fois segmenté, les différentes parties communiquent entre elles par un principe de messagerie le plus souvent asynchrone. De nombreux outils ont été définis qui permettent d'exprimer ces différentes parties, leurs interactions, et de prendre en charge l'acheminement des messages [Storm, MillWheel, Spark, TimeStream ...]. Cependant, ces outils utilisent des interfaces et des langages particuliers. Il est nécessaire de former les équipes de développement à leur utilisation, d'engager des experts et de réécrire le code initial. Cette nouvelle architecture est globalement moins souple et souvent moins propice aux changements rapides, de ce fait le changement d'architecture présente une prise de risque dans la poursuite du projet de par les modifications impliqués sur le developpement, sans pour autant d'impact direct sur la nature du service rendu.

Nous proposons un outil visant à automatiser ce changement d'architecture, en apportant une vision segmentée du programme sans modifier le code initalement développé. Un tel outil permettrait de lever les risques décrits ci-dessus. Nous visons des applications Web dont les sollicitations proviennent des flux de requêtes utilisateurs et dont le développement initial est réalisé selon une approche web 'basique' (serveur web / traitement applicatif / data). Nous pensons qu'il est possible d'analyser cette classe d'applications dès les premières étapes d'exploitation afin de les exprimer sous la forme de flux échangés entre fonctions autonomes, relocalisables.

Nous supposons que les applications sont développées dans un langage dynamique comme Javascript, et nous proposons un outil capable d'identifier les flux internes d'échanges, de définir des unités de traitement de ces flux, et de pouvoir gérer de manière dynamique ces unités. L'outil identifie ces unités sans être intrusif dans le code existant mais en proposant une sur-expression du programme initial reposant sur le paradigme de fluxion que nous allons définir et qui servira

de cœur à notre proposition.

TODO La section 2 présente le principe de fluxion en le positionnant par rapport à l'existant. La section 3 ...

2. MODÈLE D'EXÉCUTION FLUXIONNEL

2.1 Fluxions

Le modèle d'exécution fluxionnel a pour fonction de manipuler et d'invoquer des unités d'exécutions autonomes n'ayant pour paramètre d'entrée et de sortie que des flux, c'est à dire des séquences continues et infinies de données agrégées par messages. Nous avons appelé ce type d'unité d'exécution autonome une fluxion. C'est à dire une fonction, au sens de la programmation fonctionnelle, dépendant exclusivement de flux de données. Elle est composée d'un nom unique, d'une fonction de traitement, et d'un contexte mémoire au moment de son exécution.

Les messages sont composés du nom de la fluxion destinataire et d'un corps, et acheminés par un système de messagerie. Après avoir traité un message, la fluxion de traitement modifie son contexte local, puis termine son exécution en renvoyant un message sur son flux de sortie. Chaque fluxion renvoie un message unique à destination d'une ou plusieurs fluxions. Le contexte d'exécution de la fluxion de traitement est composé de l'ensemble des variables de mémoire dont dépend la fluxion pour pouvoir reprendre son traitement entre deux messages.

Les fluxions forment des chaînes de traitement liés par les flux. L'ensemble de ces chaînes forme un graphe direct orienté.

2.2 Système de messagerie

Le système de messagerie est le cœur du modèle d'exécution fluxionnel. Il a pour fonction, à la fois d'acheminer les flux de messages, et d'invoquer les fluxions.

Il est construit autour d'une file de messages, traités les uns après les autres par invocation de la fluxion destinataire. L'utilisation d'une file de messages permet d'exécuter plusieurs chaînes de traitement parallèlement et équitablement, sans faire de différence dans l'ordonnancement entre messages locaux et messages provenant du réseau. Le cycle de vie d'une application fluxionnelle est illustré figure 1.

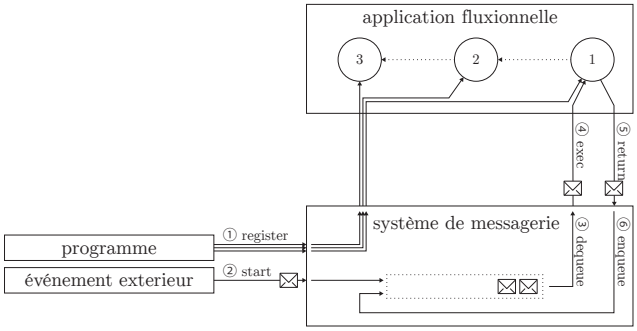

Figure 1: Schema du système de messagerie

Chaque fluxion dois être enregistrée dans le système de messagerie. Cet enregistrement associe une fonction de traitement à un nom, et un contexte d'exécution initial. Le système de messagerie achemine les flux de messages en se basant sur les noms des fluxions. C'est pourquoi il ne peut pas exister deux fluxions ayant le même nom. L'enregistrement se fait à l'aide de la fonction `register(<nom>, <fn>, <contexte>)`, étape (1) sur la figure 1.

Pour déclencher une chaîne de fluxions un premier message est envoyé à destination d'une fluxion par l'intermédiaire du système de messagerie, en utilisant la fonction `start(<msg>)`, étape (2) sur le figure 1. Cette fonction va placer un premier message dans la file. Le système dépile ce message, et exécute la fonction de traitement destinataire, étape (3) et (4) sur le figure 1. Le message résultat de cette exécution est alors empilé dans la file de message, étape (5) et (6) sur le figure 1. Le système boucle alors sur les étapes (3) à (6) jusqu'à ce qu'il n'y plus de messages dans la file.

Les algorithmes 1 et 2 formalisent précisément le comportement du système de messagerie lors de l'appel de la fonction `start`.

Algorithm 1 Algorithme de la file de messages

```
function ProcessMsg(msg)
    for dest in msg.dest do
        fluxion ← lookup(dest)
        message ← exec(fluxion, msg.body)  ▷ (4) & (5)
        enqueue(message)                   ▷ (6)
    end for
end function
```

Algorithm 2 Algorithme de parcours de la file

```
function loopMessage()
    while msg presents in msgQueue do
        msg ← dequeue()                    ▷ (3)
        ProcessMsg(msg)
    end while
end function
```

2.3 Interfaces externes

Afin de pouvoir interagir avec le monde extérieur, il faut définir des interfaces de bordures avec l'extérieur du système. Notre approche reposant principalement sur les architectures Web, l'interface visée est l'intégration des interfaces REST, qui engendre deux composants de bordures.

In permet de recevoir des connections clientes entrantes suivant le protocole REST. Pour chaque connexion entrante, le composant **In** va transmettre au composant **Out** la connexion lui permettant de répondre au client Web, et un identifiant permettant d'associer messages et connexions. Il transmet ensuite l'identifiant à la première fluxion de la chaîne de traitement en appelant la fonction `start`.

Out permet d'envoyer le résultat de la chaîne de traitement au client Web. Afin de recevoir les messages de la chaîne de traitement, le composant **Out** est enregistrée dans le système de messagerie.

Le schéma 2 présente les éléments d'un système Web fluxionnel et détaille les étapes d'acheminement d'un message par le système de messagerie.

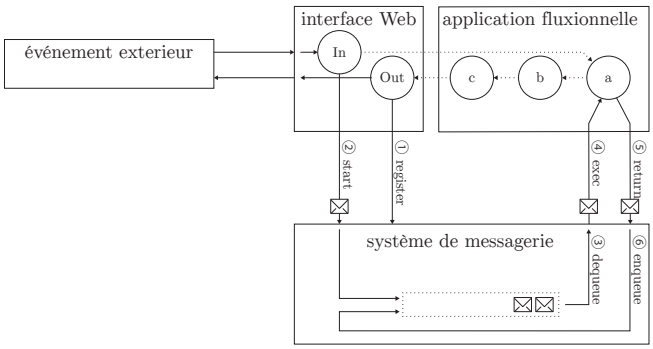

Figure 2: Schema d'un système fluxionnel avec une interface Web

2.4 Exemple de fluxion

Afin d'illustrer le modèle d'exécution fluxionnel, nous présentons ici un exemple de son utilisation à travers un simple compteur de visite.

Ce service compte le nombre de connexions HTTP de chaque utilisateur et lui renvoie ce nombre dans la réponse.

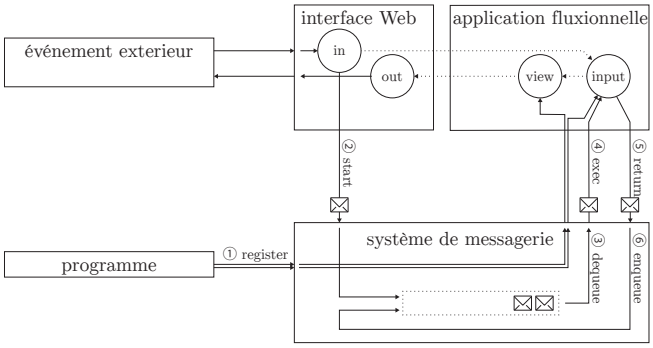

Figure 3: Schema de l'exemple d'un système fluxionnel avec une interface Web

La version initial du service pourrait représenter à l'extrait de code 4.

Dans ce code il faut distinguer 3 éléments. La variable `count` est une mémoire persistante permettant de compter le nombre de visite pour chaque identifiant d'utilisateur distinc. Elle sera donc transformée dans le notion de contexte d'exécution du système fluxionnel. La fonction `reply` réalise le traitement fluxionnel type que l'on souahite réaliser. Enfin les deux méthodes `get` et `send` permettent d'interagir avec le Web.

Ce programme minimal est transformé dans notre système selon les fluxions illustrés figure 5.

L'extrait de code 6 représente ce même service de comptage dans le modèle fluxionnel.

```
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
         || 1;
7    var visits = count[req.params.id];
8    var reply = req.params.id + '[' + visits + ']';
9    res.send(reply);
10 });
11
12 port = 8080;
13 app.listen(port);
14 console.log("Listening port: "+port);
```

Figure 4: Service initial

```
1  var flx = require('./lib/flx')
2    , web = require('./lib/external/web');
3
4  \\ 2
5  flx.register("input", function(msg){
6    this.uid[msg.uid] = this.uid[msg.uid] + 1 || 1;
7    msg.count = this.uid[msg.uid];
8    return this.m("view", msg);
9  },{
10   uid: {}
11 });
12
13 \\ 3
14 flx.register("view", function(msg) {
15   msg.view = msg.uid + "[" + msg.count + "]";
16   msg.uid = undefined;
17   msg.count = undefined;
18   return this.m("output", msg);
19 })
20
21 web.listen();
```

Figure 6: Code fluxionnel

La code classique est bien plus concis que le code fluxionnel du fait de la segmentation et de l'encapsulation impliqué par le modèle fluxionnel.

Le service à été ségmenté comme suit :

(1) Le point d'entrée **in** réagit à la connexion d'un client et démarre la chaîne de traitement en appelant `start`. Ce n'est pas une fluxion, il n'est pas enregistré dans le système de messagerie, et est lié à la machine recevant les connections entrantes.

(4) Le point de sortie **out** est enregistré dans le système comme une fluxion mais elle est liée à la machine recevant les connexions clientes afin de pouvoir y répondre.

(2) La fluxion `input` est la première à recevoir le message indiquant une connexion cliente. Son traitement consiste à incrémenter le compteur de l'utilisateur présent dans son scope, et à renseigner ce compteur dans le message, avant de le renvoyer à la fluxion suivante. Elle contient l'ensemble de la logique de ce simple service. Pour des services réels, on trouverais à sa place l'ensemble des fluxions contenant la logique du service organisé en une ou plusieurs chaîne de traitement.

(3) La fluxion `view` récupère le message, et met en forme la réponse que recevra l'utilisateur, et l'envoie à la fluxion de sortie.

Ce service manipule principalement deux informations sur lesquelles qu'il est important de détailler. D'une part, les connections clientes sont partagé entre l'entrée et la sortie. Pour chaque connexion, l'entrée passe directement à la sortie un couple <id, res>. id représente simplement un identifiant permettant de retrouver res, il est envoyé à la fluxion input. res est la structure permettant de répondre au client. Ce mécanisme provoque un échange important de donnée entre le point d'entrée et le point de sortie, et implique de passer l'identifiant sur toute la chaîne de traitement, jusqu'à sa réception au point de sortie.

Les messages échangés contiennent principalement deux informations importantes : les identifiants d'utilisateurs, permettant d'incrémenter un compteur pour chaque utilisateur, et les identifiants de connexion, permettant de lier une suite de messages avec la structure contenant la connexion HTTP. Le point d'entrée, et le point de sortie du système doivent rester sur la machine où la connexion a eu lieu pour avoir accès à l'interface réseau, tandis que les autre fluxions n'ont pas cette obligation, et peuvent être migré.

Cet identifiant de connexion est nécessaire au point de sortie pour associer le résultat reçu avec la connexion vers laquelle le renvoyer à l'utilisateur. Nous passons cet identifiant pour ne pas alourdir les échanges de messages avec la structure contenant la connexion HTTP.

Ainsi, la fluxion de sortie reçois des messages provenant de deux fluxions : le point d'entrée

3. TRANSFORMATION D'UN MODÈLE À L'AUTRE

Nous développons ici les étapes de compilation necessaire pour passer d'un modèle de programmation classique utilisant Javascript, vers le modèle d'execution fluxionnel décrit dans la précédente section.

Nous listons ici l'ensemble des points de différence entre les deux modèles nécessitant une transformation lors de la compilation.

Nous considérons qu'un service web se situe dans un sous-ensemble des programmes classiques écrit dans un langage dynamique. Ce sous-ensemble implique que le programme soit structuré de manière à enchaîner des traitements séquentiellement les uns après les autres.

3.1 Différences entre les deux modèles

3.1.1 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
}
```

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
    }
}
```

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
}
```

3.1.2 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);
        }
```

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

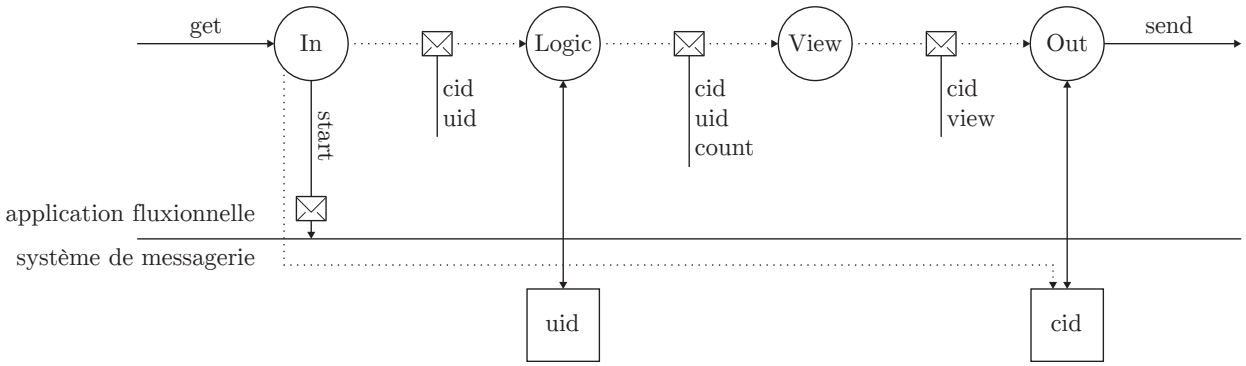

Figure 5: Suite de fluxion du compteur