Définition d'un modèle d'execution fluxionnel

[Extended Abstract]

Etienne Brodu
addr line 1
addr line 2
addr line 3
etienne.brodu@insa-lyon.fr

Stéphane Frénot
addr line 1
addr line 2
addr line 3
stephane.frenot@insa-lyon.fr

Fabien Cellier
addr line 1
addr line 2
addr line 3
fabien.cellier@worldline.com

Frédéric Oblé
addr line 1
addr line 2
addr line 3
frederic.oble@worldline.com

ABSTRACT

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

ACM proceedings, LaTeX, text tagging

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 de l'audience, celle-ci 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 l'utilisation de ces nouveaux outils, d'engager des experts et de réécrire le service initial en utilisant ces nouveaux outils. Cette nouvelle architecture est globalement moins souple et moins propice aux changements rapides. // TODO à vérifier et documenter [biblio] Ce changement d'architecture présente une prise de risque dans la poursuite du projet de par les modifications qu'ils impliquent sur le developpement.

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 décrit 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 'classique' (serveur web / traitement applicatif / data). Nous pensons qu'il est possible d'analyser cette classe d'applications dès les premières étapes d'exploitations afin de les re-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 à pour fonction de manipuler et d'invoquer des unités d'exécution autonomes n'ayant pour paramètre d'entrée et de sortie que des flux, c'est à dire des séquences continues et infinies de données agrégés 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 entrées et sorties d'une fluxion sont des flux. Un flux est un ensemble de messages à destination d'une fluxion. Les messages sont composés du nom de la fluxion destinataire et d'un corps. Ils représentent à la fois le signal d'invocation, et les données nécessaires à cette invocation. Après avoir traité un message, la fonction de traitement modifie son contexte local, puis termine son exécution en renvoyant un message sur son flux de sortie.

Les fluxions forment des chaînes de traitement liés par les flux, eux-même acheminés par un système de messagerie d'écrit plus bas. Les fluxions sont organisés en un graphe dirigé. Chaque fluxion renvoie un message unique à destination d'une ou plusieurs fluxions.

Le contexte d'exécution de la fonction de traitement est l'ensemble des variables de mémoire dont dépend la fluxion pour poursuivre un traitement entre deux exécutions.

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 entres les fluxions, et d'invoquer les fluxions.

Il est construit autour d'une file de message traité les uns après les autres. L'utilisation d'une file de messages en attentes permet d'exécuter plusieurs chaîne de traitement en parallèle de manière équitable. Le cycle de vie du système de messagerie est illustré dans la figure 1.

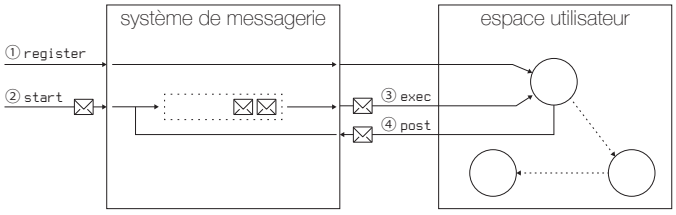

Figure 1: Schema du système de messagerie

Chaque fluxion dois être enregistré dans le système de messagerie. Cet enregistrement associe une fonction de traitement à un nom, et un contexte d'exécution propre. 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 plusieurs fluxions ayant le même nom. Lors de cet enregistrement, il est possible d'initialiser le contexte d'exécution. L'enregistrement se fait à l'aide de la fonction `register(<nom>, <fn>, <contexte>)`. Une fluxion peut elle-même enregistrer d'autres fluxions dynamiquement.

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 schema 1. Cette fonction va placer un premier message dans la file. Le système exécute la fonction de traitement destinataire de ce premier message, étape (3) sur le schema 1. Le message résultat de cette exécution est alors empilé dans la file de message, étape (4) sur le schema 1.

L'algorithme 1 explique précisément le traitement d'un message, et l'invocation de la fluxion destinataire décrit ci-dessus. Tandis que l'algorithme 2 explique le parcours de la file de messages.

Algorithm 1 Algorithme de traitement de la file de messages

```
function ProcessMsg(msg)
    for dest in msg.dest do
        flx ← lookup(dest)
        res ← exec(flx, msg.body)        ▷ étape (3)
        enqueue(res)                     ▷ étape (4)
    end for
end function
```

Algorithm 2 Algorithme de parcours de la file

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

2.3 Interface

Le système fluxionnel ne manipule que des fluxion par l'intermédiaire d'un système de messagerie. Afin de pouvoir interagir avec le monde extérieur, il faut définir des interfaces de bordure. Les bordures du systèmes sont des fluxions qui font l'interface avec l'extérieur du système.

Notre approche repose sur une espérance de gain technologique principalement sur les architectures Web. Le premier point d'entré visé est l'intégration des interfaces REST, mais tout autre point d'entrée est valable tant qu'une interface de bordure en flux peut être défini.

Dans notre approche, il existe deux types de fluxion de bordures web :

les **entrées** permettent de recevoir des connections client entrantes suivant le protocole HTTP. C'est donc le premier maillon de la chaîne de traitement. Pour chaque

connexion entrante, l'entrée va générer une bordure de sortie permettant de répondre au client.

les **sorties** permettent d'envoyer le résultat de la chaîne de traitement au client. C'est donc le dernier maillon de la chaîne de traitement.

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.

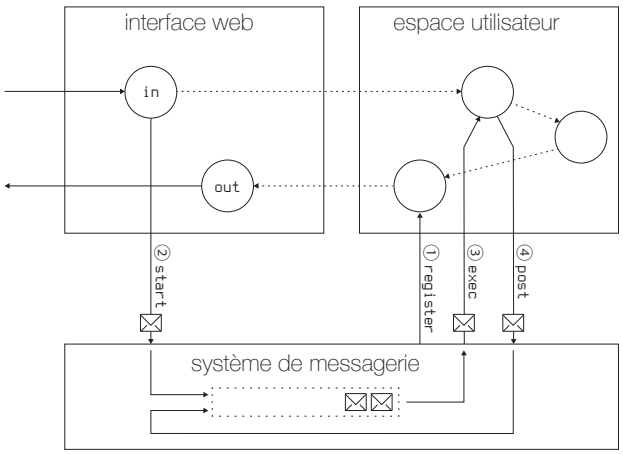

Figure 2: Schema d'un système fluxionnel avec une interface web

Le système Web est donc le déclencheur d'une chaîne de traitement de requêtes à chaque nouvelle requête d'un utilisateur un appel à la fonction start ('/', <param>) est réalisé dans le système de messagerie. Au démarrage du système Web, deux fluxions de bordure sont lancées. La fluxion de bordure 'in' n'est pas enregistré dans le système de messagerie. Elle prend les paramètres de la requête Web, place l'identifiant de la connexion client dans le contexte de la demi-fluxion de sortie, puis lance le traitement de la requête en invoquant la fonction 'start' du système de messagerie.

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 service de comptage de visite.

Ce service permet de compter le nombre de connexions HTTP de chaque utilisateur sur ce service et renvoie le résultat sous forme d'une réponse HTTP.

Le code 3 représente le code de ce service de comptage dans un modèle classique.

Le code 4 représente le code de ce même service de comptage dans le modèle fluxionnel.

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

Le service à été ségmenté comme suit :

```
1  var app = require('express')();
2
3  var count = {};
4
5  app.get('/:id', function(req, res){
6    res.send(req.params.id + '[' + (count[req.params.
         id] = (count[req.params.id] + 1) || 1 ) + ']'
         );
7  });
8
9  port = 8080;
10 app.listen(port);
11 console.log("Listening port: "+port);
```

Figure 3: Code classique

```
1  var flx = require('./lib/flx')
2    , express = require('express')
3    , app = express();
4
5  flx.register("output", function(msg){
6    if (msg.res) {
7      this.cid[msg.cid] = msg.res;
8    } else {
9      this.cid[msg.cid].send(msg.view.toString());
10   }
11   return undefined;
12 }, {
13   cid: {}
14 })
15
16 flx.register("input", function(msg){
17   this.uid[msg.uid] = this.uid[msg.uid] + 1 || 1;
18   msg.count = this.uid[msg.uid];
19   return this.m("view", msg);
20 },{
21   uid: {}
22 });
23
24 flx.register("view", function(msg) {
25   msg.view = msg.uid + "[" + msg.count + "]";
26   msg.uid = undefined;
27   msg.count = undefined;
28   return this.m("output", msg);
29 })
30
31 app.get('/:id', function(req, res) {
32   var uid = req.params.id;
33   var cid = req.client._idleStart;
34
35   flx.start(flx.m("output", {cid: cid, res: res}));
36   flx.start(flx.m("input", {uid: uid, cid: cid}));
37 })
38
39 app.listen(8080);
```

Figure 4: Code fluxionnel

— Le point d'entrée `app.get` réagit à la connection d'un client et démarre la chaîne de traitement. Ce n'est pas une fluxion et il n'est pas enregistré dans le système de messagerie.
— Le point de sortie `output` est une fluxion mais elle est liée à la machine sur laquelle arrive la connexion afin de pouvoir y répondre.
— 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.
— La fluxion `view` récupère le message, et met en forme la réponse que recevra l'utilisateur, et l'envoie à la fluxion 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