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 d'accueillir 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 très 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 très 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 familiers avec ces outils 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 permettant d'éviter de forcer ce changement d'architecture en proposant d'automatiser un changement de paradigme dans les programmes, sans en modifier le code d'origine mais en y apportant une vision segmentée du programme initalement développé. 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 échangé entre fonction 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 principe du modèle d'exécution fluxionnel est d'identifier 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 cette 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 se termine en renvoyant un message sur son flux de sortie.

Les fluxions forment des chaînes de traitement lié par les flux, trapsnorté par un système de messagerie d'écrit plus bas. Les fluxions sont organisés dans un graph dirigé. (note : le graphe n'est pas acyclique)

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 permet d'acheminer les flux entre les fluxions selon leurs noms.

Les messages soumis à l'envoi sont poussés dans une file, en attendant d'être traité par le système de messagerie.

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)
        enqueue(res)
    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
```

La fonction de traitement ne sera appelé qu'avec un seul paramètre : le message reçu.

La fonction de traitement doit renvoyer un message, ou un tableau de plusieurs messages. Un message se compose du nom de la fluxion destinataire, et optionnellement d'un corps. Il se construit sous la forme suivante :

```
{
  dest: "ma_fluxion", // nom de la fluxion
       destinataire du message
  body: // corps du message, optionnel
      {}, [], "", ...
}
```

Pour exécuter un programme exprimé sous la forme de fluxions, le modèle d'exécution fluxionnel propose deux fonctions d'interface : l'enregistrement et le démarrage.

Lors de l'initialisation du programme, il est nécessaire d'enregistrer les fluxions pour qu'elles puissent ensuite recevoir des messages. L'enregistrement se fait à l'aide de la fonction `register(<nom>, <fn>, <contexte>)`. Le nom de la fluxion lui permet d'être identifié de manière unique dans le modèle d'exécution, et ainsi de recevoir des messages. Il ne peut donc pas exister plusieurs fluxions ayant le même nom. Une fluxion peut elle-même enregistrer d'autres fluxions dynamiquement.

L'exécution des fluxions enregistré dans le modèle d'exécution se fait en envoyant un message à une première fluxion, qui va elle même envoyer un message à une seconde et ainsi de suite. Pour déclencher cette chaîne un premier message est envoyé au système de messagerie en utilisant la fonction `start(<nom>,<msg>)`.

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 1 présente les éléments d'un système Web fluxionnel.

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

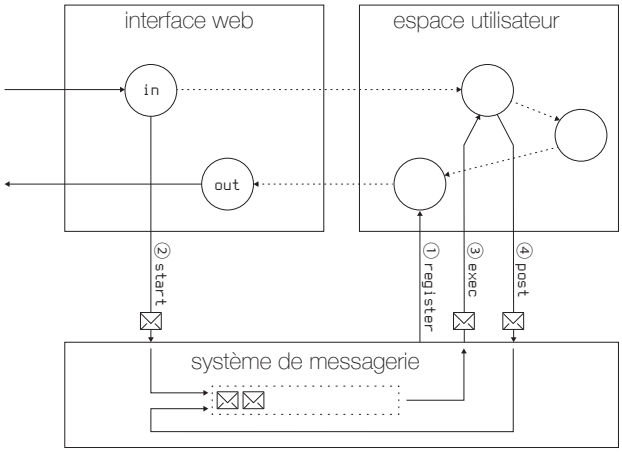

Figure 1: Schema d'un système fluxionnel avec une interface web

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 une application de comptage de visite.

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

var count = {};

app.get('/:id', function(req, res){
  res.send(req.params.id + '[' + (count[req.params.id
      ] = (count[req.params.id] + 1) || 1 ) + ']');
});

port = 8080;
app.listen(port);
console.log("Listening port: "+port);
```

```
var flx = require('./lib/flx')
  , express = require('express')
  , app = express();

flx.register("output", function(msg){
  if (msg.res) {
    this.cid[msg.cid] = msg.res;
  } else {
    this.cid[msg.cid].send(msg.view.toString());
  }
  return undefined;
}, {
  cid: {}
})

flx.register("input", function(msg){
  this.uid[msg.uid] = this.uid[msg.uid] + 1 || 1;
  msg.count = this.uid[msg.uid];
  return this.m("view", msg);
},{
  uid: {}
});

flx.register("view", function(msg) {
  msg.view = msg.uid + "[" + msg.count + "]";
  msg.uid = undefined;
  msg.count = undefined;
  return this.m("output", msg);
})

app.get('/:id', function(req, res) {
  var uid = req.params.id;
  var cid = req.client._idleStart;

  flx.start(flx.m("output", {cid: cid, res: res}));
  flx.start(flx.m("input", {uid: uid, cid: cid}));
})

app.listen(8080);
```

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