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 parmi les communautés open source pour capter rapidement une communauté d'utilisateurs.

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. Afin de 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. Ces 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. [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 ou 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 de paradigmes de développement représente une prise de risque dans la poursuite du projet car ces outils sortent du cadre grand public suffisamment accessible pour favoriser l'émergence spontanée de nouveaux services.

Nous proposons un outil permettant d'éviter de forcer ce changement de paradigme en proposant une vision segmentée de programme 'standard'. 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 plus ou moins concrètement sous la forme de flux d'échange.

Nous supposons que les applications serveur sont développées dans un langage dynamique comme Javascript, et nous proposons un outil capable d'identifier les flux internes, 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 en utilisant le paradigme de fluxion que nous allons définir et qui servira au cœur de 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. Une unité est autonome quand elle peut être déplacée dynamiquement d'environnement d'exécution au cours de son activité. Déplacer une unité nécessite de déplacer avec son code d'exécution son contexte d'exécution courant, c'est à dire l'ensemble des variables de mémoire dont dépend l'unité. Dans notre approche, le contexte est capturé dans des flux spécifiques réinjectés dans l'unité à chaque exécution. Ainsi, déplacer une telle unité consiste à déplacer le code fonctionnel vers une nouvelle destination, puis à rediriger les flux d'entrées et de sorties en conséquence. L'état, les paramètres d'appel et les paramètres de retour étant capturé dans des flux, on a une estimation instantané du coup de déplacement de la fonction, proportionnel au débit des flux concernés.

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.

2.1.1 Fluxions

Une fluxion est une unité autonome de traitement de flux. Ce traitement est exprimé sous la forme d'une fonction.

Ces flux d'informations sont composés de messages, échangés d'une fluxion à l'autre. L'enchaînement des fluxions relié par des flux forme une chaîne de traitement.

Lorsque le système de messagerie délivre un message à une fluxion, celle-ci a la possibilité de :

- modifier le message, et
- envoyer un (ou plusieurs) message à la (ou les) prochaine fluxion dans la chaîne de traitement.

Les bordures du systèmes sont des fluxions qui font l'interface avec l'extérieur du système. Il existe deux types de fluxion en bordures :

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.

2.1.2 Message

Les messages sont des structures de données composant le flux d'information éphémère échangé entre les fluxion. Ils représentent à la fois :

- le signal d'invocation du traitement d'une partie du flux
- les données du flux à traiter

Un message est composé d'une adresse, et d'un corps :

```
1  {
2      addr: ma_fluxion, // adresse de destination du
           message
3      body: // corps du message, optionnel
4          {}, [], "", ...
5  }
```

Les messages sont acheminé entre les fluxions par le système de messagerie. À la réception d'un message, le système de messagerie invoque la fluxion destinataire du message. Cette fluxion renvoie à son tour un message, acheminé par le système de messagerie jusqu'à la prochaine fluxion.

2.1.3 Adressage des fluxions

Les fluxions communiquent par l'intermédiaire de flux composés de messages. Le système de méssagerie distribue ces messages aux différentes fluxions en se basant sur des adresses. Les fluxions sont enregistré dans le système de messagerie en spécifiant une adresse unique dans le système. Cette adresse permet aux fluxions de recevoir des messages.

Lorsqu'une fluxion se déplace d'un nœud à l'autre, son adresse se déplace aussi, et ce avec les flux associé à cette adresse. Les deux nœuds communiquent ce déplacement entre eux pour le valider, et auprès des autres nœuds pour les en informer. De cette manière chacun des nœuds connaît l'emplacement de chacune des fluxions sur les autres nœuds. Ainsi le système de messagerie composé de l'ensemble de ces nœuds peut acheminer n'importe quel message sur l'ensemble du système.

2.1.4 Persistance des états

Les fluxions n'ont pas d'état afin de pouvoir être transporté d'un nœud à l'autre en cours d'exécution. Comme les applications écrit dans un modèle de programmation classique reposent sur l'existence d'un tel état, nous l'avons cloisonné dans un flux spécifique à chaque fluxion. Ainsi l'état suit la mobilité d'une fluxion lors de la redirection de ses flux.

Lors de l'invocation d'une fluxion, le système de messagerie l'execute en fournissant comme contexte le contenu dde ce flux spécifique.

Nous avons appelé ce flux capturant l'état d'une fluxion un scope.

2.1.5 Système de messagerie

Le système de messagerie tiens à jours l'annuaire d'adresses des fluxions, et s'occupe d'acheminer les message d'une fluxion à l'autre. Cette dernière fonction peut être comparé au scheduler d'un système d'exploitation classique.

Le système de messagerie est présent sur chacun des nœuds de l'infrastructure, et écoute en permanence l'arrivée de messages en provenance de clients, ou d'autres nœuds. À l'arrivée d'un message, il vérifie que la fluxion associé à l'adresse

TODO détailler techniquement

2.1.6 Système de supervision

Le système de supervision rassemble des métriques sur l'écoulement des flux d'informations entre les fluxions, par l'intermédiaire du système de messagerie, et permet, grâce à ces métriques, de répartir les fluxions de manière optimale sur la topologie matériel, et de donner des indices à un opérateur humain sur la qualité de service global du système.

TODO détailler techniquement

2.2 Plateforme de fluxions

Une application fluxionnelle est composée d'un enchaînement de fluxions. Chaque fluxion présente le même comportement :

- elle est invoquée par un système de messagerie,
- elle effectue des opérations à partir du message reçu,
- elle modifie son état interne rendu persistant dans le système de messagerie
- puis elle retourne un ensemble de messages à destination d'autres fluxions.

Dans notre approche, un message est une structure clé/valeur contenant deux couples : le nom de la fluxion à invoquer *name* et le corps du message *body*, contenant l'ensemble clé/-valeur des attributs nécessaires à l'exécution de la fonction.

Le système de messagerie repose sur uniquement deux fonctions publiques :

- l'enregistrement de fluxion
  `register(<nom>, <fn>, <contexte>)`
- le démarrage d'une chaîne de traitements
  `start(<nom>,<param>)`

Les données et la logique d'une application sont cloisonnés à l'exécution, il est possible de mettre à jour une fluxion en la remplaçant dans le système, sans impacter l'exécution de l'application. De plus déplacer une fluxion ne nécessite pas de la supprimer de son nœud initial, car seul la réception de message déclenche une exécution.

La relocalisation d'une fluxion se fait de manière transparente par l'application, par le système de messagerie qui connaît la localisation des fluxions.

2.3 Architecture Web

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. 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. Le schéma **??** présente les éléments d'un système Web fluxionnel.

TODO schema

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 demi-fluxions sont lancées. La demi-fluxion '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.

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.

```
1         function(req) {
2             // traitements sur req
3             return next(req);
4         }
```

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