



Programmation java : des Green Patterns ?

Le Green Challenge a fait émerger des patterns de développement permettant de réduire la consommation énergétique des applications Java : les "Green Patterns".

C'était un concours ouvert, lancé dans le cadre de l'Université du SI 2010, en partenariat avec GreenIT.fr. Il a mobilisé 15 équipes, soit une cinquantaine de développeurs, pendant 2 mois. Les gagnants ont remporté des iPads, iPods et des places pour l'USI.

A propos des QRcodes

Les QR Codes sont des codes-barres à 2 dimensions qui permettent à tout téléphone muni d'un appareil photo numérique d'accéder simplement au Web. En photographiant un QR Code, puis en le décodant, le téléphone peut récupérer le contenu d'une carte de visite ou se rendre sur un site Web.

Pourquoi travailler sur les QRcodes? Nous avons choisi de faire un challenge autour des QR Codes car le décodage de ces images consomme beaucoup de temps CPU. Il nous permet donc de mesurer une amélioration de la sobriété de l'application, chose que nous aurions eu du mal à quantifier avec une application peu consommatrice en CPU.

Le périmètre du Challenge

Pour identifier les "greens patterns", le challenge proposait de faire baisser la consommation sur un exemple d'application: QRDecode. L'objectif de l'application QRDecode est de décoder des QRcodes, d'afficher les coordonnées des contacts auxquels correspondent ces codes barres et de positionner ces contacts sur une carte. Une implémentation de référence était proposée, en voici une capture d'écran [fig.1].

Schématiquement, si l'on se limite aux flux, le rôle de cette application est de transformer plusieurs dizaines de QR Codes représentés par des fichiers .QRC transmis à l'application en :

- Des cartes de visites,
- Des images (la représentation visuelle du QR Code),
- Des coordonnées GPS,
- Une carte affichant ces coordonnées [fig.2].

Les différentes fonctionnalités de l'application QR Decode sont donc :

1. Le décodage des fichiers QR Code en une représentation carte de visite (en l'occurrence VCard).



2. La transformation des QR Codes en images,
3. La géolocalisation des adresses contenues dans les VCard pour obtenir des coordonnées GPS,
4. Le positionnement des points GPS sur une carte graphique,
5. L'affichage des cartes de visites et de la carte graphique.

L'implémentation de référence de l'application QR Decode a été réalisée en Java et se décompose en deux parties: une partie serveur qui s'exécute sur Google App Engine et une partie client qui s'exécute sur le navigateur. Les deux parties de l'application sont instrumentées pour récupérer le temps CPU consommé. Pour la partie serveur cela se réalise via des API spécifiques fournies avec le projet, pour la partie client cela se réalise par l'intermédiaire d'un plug-in FireFox développé pour l'occasion, GreenFox.

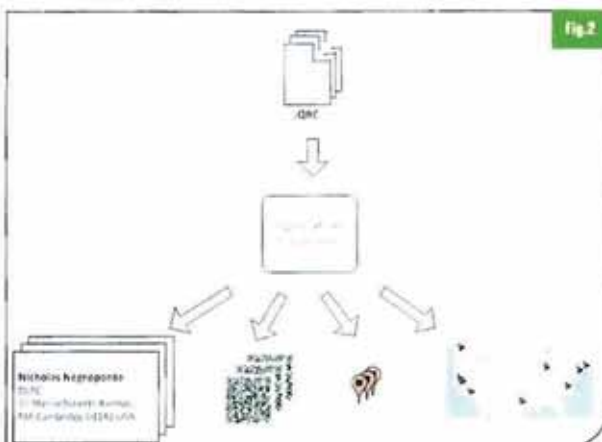
Présentation de GreenFox

GreenFox mesure le temps CPU consommé par l'utilisation de Firefox. Il fonctionne sous Firefox 3.6, pour l'instant sous Windows uniquement, à partir de Windows XP SP3. GreenFox s'installe sous forme de toolbar en bas de la fenêtre de votre navigateur. Il mesure le temps CPU consommé par le process FireFox, entre le moment où vous cliquez sur "start" et le moment où vous cliquez sur "stop". Il s'agit du temps CPU consommé par tout le process Firefox et pas uniquement de la fenêtre de votre application web. Donc il est recommandé de désactiver toutes les autres extensions afin de ne pas perturber la mesure. Chaque mesure est envoyée au serveur de collecte du Challenge [fig.3].

Les équipes étaient jugées en fonction de la réduction de la consommation qu'elles apportaient à l'application de référence. Les paragraphes suivant décrivent les méthodes mises en œuvre par les différents participants.

L'architecture et répartition des traitements

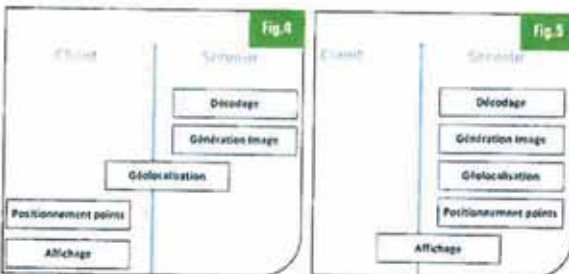
La bonne répartition des traitements entre le serveur et le client est un des éléments majeurs d'optimisation des performances





énergétiques de l'application. En effet, le code qui s'exécute côté serveur chez Google dispose de conditions énergétiques idéales (une plateforme hautement mutualisée, un PUE optimisé) alors que le code qui s'exécute sur la machine cliente, même dans les conditions minimales que nous avons choisies (un netbook de type Asus EeePC) reste celui d'un PC individuel avec une importante déperdition d'énergie. Dans l'implémentation de référence, les traitements se répartissaient comme suit : [fig.4]

La première optimisation consistait donc à décaler le plus possible de traitements côté serveur. Voici la répartition "idéale" proposée par les deux premières équipes sur le podium [fig.5].



La géolocalisation est effectuée par appel d'un traitement Google. Néanmoins le fait de le réaliser via un appel JavaScript est nettement plus coûteux que lorsqu'il est intégré dans le code serveur. De plus, il est effectué pour chaque adresse, ce qui est pénalisant. Une optimisation consiste à réaliser un appel batch sur le serveur pour réaliser la totalité du traitement en un seul appel. C'est possible via l'API Google mais cela impose des limites en nombre de requêtes lancées, un des gagnants a donc choisi d'utiliser MapQuest à la place.

Le positionnement des points sur la carte est également réalisé dans l'application de référence par l'API JavaScript de Google Maps. C'est encore une fois très coûteux en CPU. Deux stratégies ont été utilisées pour éviter cet écueil. Dans les deux cas il s'agissait de supprimer les appels JavaScript. La première stratégie consiste à réaliser une génération complète sur le serveur, cela est possible en utilisant la version statique de Google Maps qui génère une seule image intégrant la carte et tous les points [fig.6]. Une autre stratégie était l'utilisation d'un Canvas HTML 5 dans la page avec un positionnement des points en relatif sur un simple fond de carte [fig.7]. Dans les deux cas cela limite les fonctionnalités de l'application, ce qui était autorisé par le règlement.

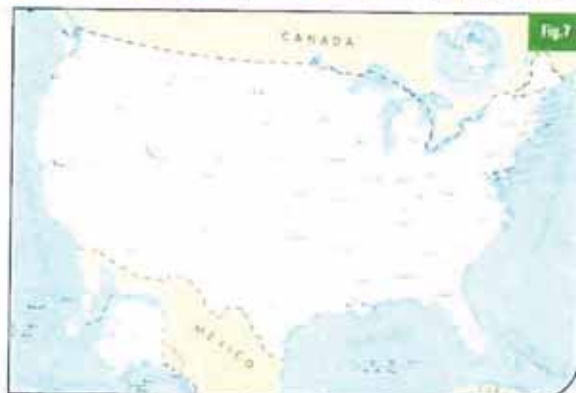


L'affichage se fait dans le navigateur et est donc obligatoirement côté client. Néanmoins, la plupart des candidats ont choisi d'alléger les traitements d'affichage en préparant le travail côté serveur. Ainsi, deux des équipes gagnantes ont directement encodé les images dans la page HTML (soit en base64 soit en passant par le data URI Scheme). Voir l'exemple ci-dessous.

```

```

L'affichage HTML des cartes de visites a également été généré côté serveur, ce qui permet d'éliminer le JavaScript qui génère le code HTML à partir de la représentation mémoire/JSON des données.





Optimisations « Green »

La plupart des équipes ont également travaillé sur l'optimisation des traitements côté serveur. Dans un premier temps, l'idée était d'effectuer un profiling de l'application afin d'identifier les lignes de code sur lequel le processeur passe le plus de temps. Plusieurs outils Java ont été utilisés pour cela : Netbeans Profiler, Java VisualVM ou JavaProfiler [Fig.8].

Voici les différentes optimisations réalisées :



Décodage des QRcodes

Le traitement de décodage des QR Codes est clairement le traitement le plus coûteux en temps processeur sur la partie serveur.

Dans l'implémentation de référence, le traitement s'appuyait sur la librairie Open Source de Yusuke Yanbe [Fig.9].

Deux des équipes sur le podium ont fait le choix de chercher et de benchmarker une autre librairie. Ils se sont donc appuyés sur la librairie Zebra Crossing qui offre plus de fonctionnalités (support d'un plus grand nombre de types de codes barres) et qui est surtout beaucoup plus performante que la librairie de départ. Une autre stratégie, plus complexe, consistait à optimiser la librairie existante. Pour cela un profiling plus fin et des optimisations sur le code Java ont été mises en œuvre. En particulier :

- Eviter les copies de blocs mémoires (voir exemple de code ci-dessous),
- Limiter le nombre d'objets à instancier et favoriser la réutilisation des instances,
- Passer en variable Static des tableaux de valeurs,
- Limiter les conversions de types,
- Limiter l'utilisation d'objet nécessitant de la synchronisation (ThreadSafe).

A noter que l'objectif de ces optimisations n'est pas de limiter l'usage mémoire (peu impactant énergétiquement) mais de limiter le nombre d'opérations pour alléger la CPU.

```
int[][] imageToIntArray(QRCodeImage image) {
    int width = image.getWidth();
    int height = image.getHeight();
    int[][] intImage = new int[width][height];
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            intImage[x][y] = image.getPixel(x, y);
        }
    }
    return intImage;
}
```

Optimisation de la génération des images

Un autre traitement coûteux était la génération de l'image graphique du QR Code à partir de l'adresse.

Une implémentation était fournie dans l'application de référence. Deux équipes ont fait le choix de la substituer par un appel à une API GoogleChart qui propose cette fonctionnalité. Cela permet en un simple appel HTTP de disposer de l'image.

```

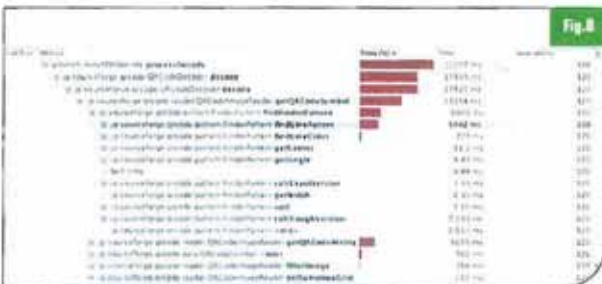
```

Même si cette option génère une économie énergétique, le jury a pris la décision de pénaliser ces deux équipes car l'appel à ce traitement n'est pas visible à travers nos instruments de mesure et pénalisait donc les autres équipes.

La plupart des autres équipes ont réalisé des optimisations sur le code de génération des images. L'optimisation la plus simple était de "rognier" la taille pour éviter le traitement de pixels inutiles. Différentes optimisations ont également été réalisées sur le traitement pour éviter de manipuler des pixels de couleurs alors que l'image du QR Code est nécessairement en noir et blanc. Enfin, l'encodage de l'image en bitmap a été privilégié en évitant de passer par les API AWT qui sont peu performantes.

Optimisation de l'IHM

L'optimisation de l'affichage consistait d'abord à éviter l'utilisation





du JSP qui provoque un overhead, surtout au premier chargement. La plupart des équipes ont également pris la décision de construire le code HTML directement en utilisant des StringBuilders (voir exemple ci-dessous).

```
public String decode(StringBuilder sbCards, StringBuilder sbAlerts,
StringBuilder sbImages, File file, int id) throws Throwable {
    ...
    // HTML of vcard
    sbCards.append("<li class = \"vcard\">");
    // preparing the HTML5 canvas
    sbCards.append("<canvas id=\"canvas\"");
    sbCards.append(id);
    sbCards.append(">");
    sbCards.append("<div style=\"width:\"");
    sbCards.append(wOut);
    sbCards.append("<div style=\"height:\"");
    sbCards.append(hOut);
    sbCards.append("></div></canvas>");
    sbCards.append("<span class=\"name\">");
    sbCards.append(sName);
    sbCards.append("</span>");
    sbCards.append("<span class=\"orga\">");
    sbCards.append(sOrga);
    sbCards.append("</span>");
    sbCards.append("<span class=\"addr\">");
    sbCards.append(sAddress);
    sbCards.append("</span>");
    sbCards.append("</li>\"+r\n");
    ...
}
```

La plupart des équipes ont aussi fait en sorte d'éviter les aller/retours qui nécessitent des traitements de connexions côté client et côté serveur. Une des équipes a fusionné dans la page HTML: le code HTML, la feuille CSS, les images et le JavaScript pour limiter les échanges à un seul aller/retour.

Le code HTML a été optimisé (suppression des espacements inutiles) par plusieurs équipes. Le code JavaScript a également été optimisé pour limiter le nombre d'appels AJAX qui impliquent des traitements (et donc de la CPU) pour réaliser les connexions. Le JavaScript a aussi été affusqué pour limiter le parsing. La meilleure stratégie était néanmoins d'éviter complètement le JavaScript !

Autres optimisations

D'autres optimisations ont été mises en œuvre. La principale concerne la gestion des traces et du code de débogage. L'idée

étant de réaliser ces traitements de manière conditionnelle pour ne pas consommer du temps d'exécution inutile. Par exemple le code de trace :

```
canvas.println("Adjust AP(" + x + ", "+ y+" to d("+dx+", "+dy+"");
```

est remplacé par:

```
if (canvas.isPrintlnEnabled())
    canvas.println("Adjust AP({}, {}) to d({}, {})", x, y, dx, dy);
```

Enfin, plusieurs équipes ont mis en œuvre les options de gestion du cache côté navigateur qui peuvent être intéressantes si l'application s'exécute plusieurs fois. Néanmoins le jury portait systématiquement d'un cache vide avant chaque mesure.

Conclusion : les Green Patterns

Le premier enseignement que l'on peut tirer de ce challenge est que l'optimisation énergétique d'une application est une réalité. Les gains obtenus entre l'application de référence et les équipes gagnantes vont de 20% pour la partie serveur à un gain de plus de 600% sur la partie cliente.

On constate ensuite que les stratégies gagnantes pour limiter la consommation sont :

- Réaliser le maximum de traitements côté serveur quitte à réduire l'ergonomie à l'essentiel côté client,
- Profiler l'application pour identifier les traitements fortement consommateurs de CPU,
- Optimiser ces traitements pour limiter le nombre d'opérations réalisées ou remplacer ces traitements par des bibliothèques plus efficaces,
- Pré-générer les affichages sur le serveur et éviter le code interprété (JavaScript ou JSP).

C'est à ce prix que l'on aura des applications "vertes" mais aussi, globalement, de meilleure qualité.

Un grand merci à tous les participants qui nous permettent d'appréhender ces bonnes pratiques.

Pour en savoir plus sur le challenge et GreenFox : <https://sites.google.com/a/octo.com/green-challenge/>

- Lionel Laske, C2S, Groupe Bouygues
- Frédéric Bordage, www.greenit.fr
- Guillaume Plouin, OCTO Technology

