N°166 DÉCEMBRE 2013

GNU LINUX MAGAZINE / FRANCE

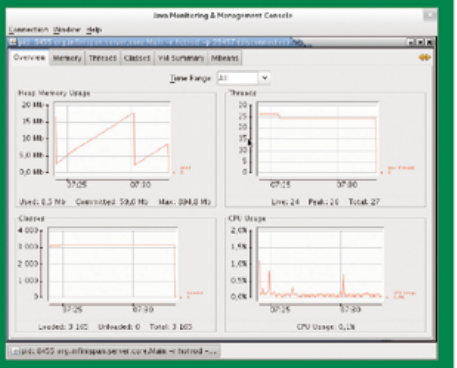
KERNEL / PEOPLE

Interview de trois principaux développeurs sur la maturité du noyau et des changements actuels p.04

ADMINISTRATION ET DÉVELOPPEMENT SUR SYSTÈMES OPEN SOURCE ET EMBARQUÉS

GRID / SCALABILITY

Réglez efficacement les problématiques de montée en charge et de passage à l'échelle grâce à InfiniSpan p.40



WEB / AJAX

Découvrez Flask, le micro-framework qui vous fera enfin aimer développer du clicka-web super intuitif p.64

PYTHON / OPENCV

Utilisez les mécanismes de reconnaissance faciale avec Python/OpenCV pour identifier vos visiteurs p.74

L 19275 - 166 - F: 7,90 € - RD

CODE / MOBILITÉ

Tirez le meilleur parti de Python et du C++ sans trop vous fatiguer avec le framework multi-plateforme Kivy p.20

ÉMULATION / IBM

Mettez un mainframe s390x dans votre laptop avec Hercules, l'émulateur d'architecture System z p.50

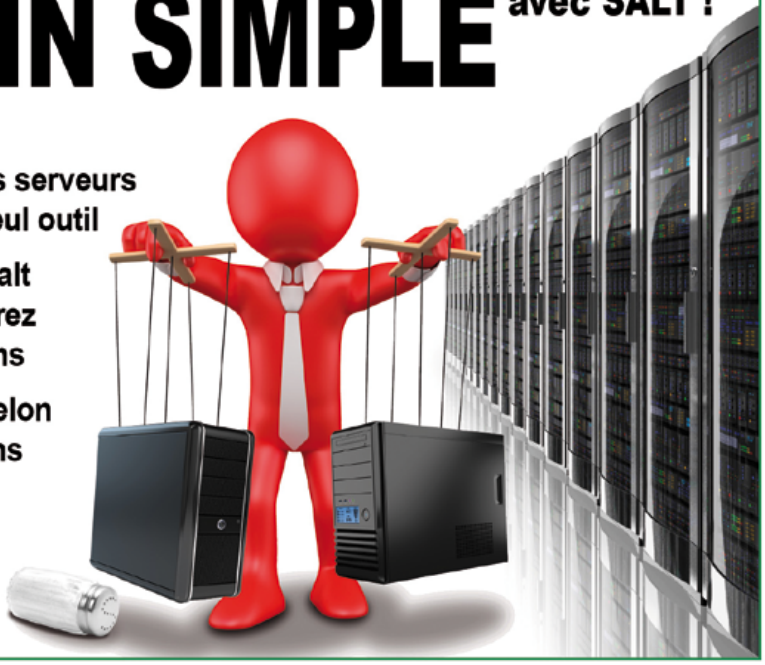
NETADMIN

Comment administrer et gérer tous vos serveurs de concert sans utiliser un monstre ?

ORCHESTRATION ENFIN SIMPLE avec SALT !

p.30

1. Pilotez vos serveurs avec un seul outil
2. Installez Salt et configurez vos minions
3. Scriptez selon vos besoins
4. Créez vos modules



ANDROID / WEBKIT

Mixez API native et contenu Webkit/HTML5 offline en utilisant une WebView dans une activité p.10

NETBSD / PXE

Créez un serveur de démarrage PXE sous NetBSD, pour démarrer l'installeur NetBSD via le réseau p.56

France METRO : 7,90 € – CH : 13 CHF – BEL/PORT.CONT : 8,90 € – DOM TOM : 8,50 € – CAN : 14 $ cad – TUNISIE : 18 TND – MAR : 100 MAD

Le CNLL

Conseil National du Logiciel Libre

et ses représentants régionaux

PRÉSENTENT

les RENCONTRES REGIONALES du LOGICIEL LIBRE & du SECTEUR PUBLIC

Du 04 Octobre 2013 au 17 Avril 2014

Paris
Lille
Metz
Strasbourg
Lyon
Rennes
Brest
Nantes
Bordeaux
Toulouse

Venez rencontrer vos pairs
avec des problématiques communes !

Venez rencontrer
les prestataires locaux de l'OpenSource !

En savoir plus sur **www.rrll.fr**

Evénement soutenu par
ADOSIS, *Agence Digitale*, spécialiste OpenSource
www.adosis.com

adosis

SOMMAIRE N°166

GNU/Linux Magazine France
est édité par Les Éditions Diamond

LES ÉDITIONS DIAMOND

B.P. 20142 – 67603 Sélestat Cedex
Tél. : 03 67 10 00 20 – **Fax** : 03 67 10 00 21
E-mail : lecteurs@gnulinuxmag.com
Service commercial : abo@gnulinuxmag.com
Sites : www.gnulinuxmag.com – boutique.ed-diamond.com

Directeur de publication : Arnaud Metzler
Rédacteur en chef : Denis Bodor
Secrétaire de rédaction : Véronique Sittler
Réalisation graphique : Jérémy Gall

Responsable publicité : Valérie Fréchard, Tél. : 03 67 10 00 27
v.frechard@ed-diamond.com
Service abonnement : Tél. : 03 67 10 00 20
Impression : pva, Druck und Medien-Dienstleistungen GmbH, Landau, Allemagne
Distribution France : (uniquement pour les dépositaires de presse)
MLP Réassort : Plate-forme de Saint-Barthélemy-d'Anjou. Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier. Tél. : 04 74 82 63 04
Service des ventes : Distri-médias : Tél. : 05 34 52 34 01
IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : À parution, N° **ISSN** : 1291-78 34
Commission paritaire : K78 976

Périodicité : Mensuel
Prix de vente : 7,90 €

FSC www.fsc.org MIXTE Papier issu de sources responsables FSC® C015136

Membre April www.april.org

ÉDITORIAL

Pourquoi faire simple quand on peut faire compliqué ?!

C'est vrai, après tout, rendre les choses complexes et « lourdes » valorise énormément à la fois le « produit » et le ou les développeurs. Dans un lointain passé (tellement lointain que le standard était MS/DOS), je m'étais même entretenu avec une personne qui m'avait avoué ouvertement ajouter dans ses applications des barres de progression « *pour bien faire comprendre à l'utilisateur que le programme travaillait durement et que cela justifiait son prix* » (non, je ne sais pas ce qu'elle est devenue aujourd'hui).

Mais la complexité est quelque chose de très relatif. Pour un utilisateur UNIX, une fenêtre pleine de menus, de cases à cocher et d'options à sélectionner est d'une complexité édifiante face à un simple fichier de configuration parfaitement limpide. L'utilisateur Windows ou Mac, quant à lui, pensera que ce simple fichier texte est, sans l'ombre d'un doute, une version informatisée du manuscrit de *Voynich*.

Malheureusement, force est de constater que la complexité va généralement croissante. Pire encore, en souhaitant simplifier les choses dans un sens, on les complique généralement sur d'autres aspects. Nos interfaces graphiques semblent suivre ce modèle à en juger par le nombre de couches et de dépendances qui ne cesse de se multiplier. Il en va de même pour les périphériques mobiles voyant à la fois leur API devenir de plus en plus riche et les environnements de développement associés de plus en plus lourds. Mais là encore, il s'agit en grande partie d'un choix, basé sur ce qui paraît être ou non complexe. Et la tendance, bien malheureuse est de considérer un Makefile plus complexe qu'un IDE basé sur Eclipse (Vous avez remarqué ? J'ai bien pris garde de ne pas mentionner les autotools), y compris dans un développement C avec pour cible une plateforme mobile utilisant comme base un ensemble de bibliothèques élégamment pensées (oui, je parle du SDK Tizen et j'attends toujours un smartphone résolument *oldschool-UNIX*).

Mais fort heureusement, il y a des exceptions. Le projet en couverture de ce GLMF en est une, car, comme vous le verrez par vous-même grâce à la plume et la verve de ce cher iMil, orchestration et gestion de configuration ne sont pas nécessairement synonymes d'usine à gaz.

KISS forever les ami(e)s ! Et puisque nous sommes en décembre, bonnes fêtes à vous !

Denis Bodor

KERNEL CORNER :

INTERVIEWS A. MORTON, G. KROAH-HARTMAN ET T. HEO

par Eric Lacombe

Nous avons le plaisir de vous présenter dans un première partie quelques pensées de Andrew Morton et de Greg Kroah-Hartman (deux figures de renom dans la communauté noyau) sur la maturité du noyau Linux et de sa communauté, sur l'importance de l'adaptation continue de Linux à son environnement, ainsi que sur des sujets liés à la maintenabilité du noyau vis-à-vis de la flexibilité que l'ABI peut apporter à l'espace utilisateur. Nous finissons par un éclairage de Tejun Heo (mainteneur des control groups) sur la refonte de l'infrastructure des control groups (en cours d'intégration depuis le 3.10 et accessible via l'option de montage « __DEVEL__sane_behavior »), où le sujet de la flexibilité est au cœur des décisions prises.

1 Interview de Andrew Morton et de Greg Kroah-Hartman

Propos recueillis par Eric Lacombe en septembre et octobre 2013.

1.1 A propos de maturité et d'évolution

[Linux Mag] : Linux ne cesse d'évoluer à un rythme effréné et des éléments majeurs de son fonctionnement sont de temps à autre refondu mais il ne devrait pas être exagéré de dire que Linux a atteint un état de maturité avancé. Toutefois, y a-t-il toujours certaines parties du noyau qui ne vous satisfont pas ?

Andrew Morton [AM] : Eh bien, performance, performance, performance. Nous aurons toujours à trouver des problèmes et à les fixer. Et parfois, cela fera empirer les choses ! Et alors il faudra de nouveau les fixer.

Greg Kroah-Hartman [GKH] : Il y a toujours quelque chose que j'ai sur ma « *todo list* » que j'aimerais voir nettoyer de différentes façons. Mais ce n'est que ma liste personnelle. Je suis sûr que tous les développeurs noyau disposent de leur propre liste pour des raisons qui leur sont propres, rendant ma liste pas plus importante que les autres.

Elle se trouve en ligne, est plutôt ennuyeuse, et probablement pas tout à fait à jour :

https://github.com/gregkh/gregkh-linux/blob/master/TODO

Aussi, merci de définir ce que tu entends par « mature » :)

Sérieusement, qu'est ce que ça veut réellement dire pour un système qui s'adapte constamment à un monde qui ne cesse de changer, à différents contextes matériels. Pour moi, « mature » dans ce contexte signifierait « obsolète », n'est-ce pas vrai ? Et je ne pense pas que Linux soit obsolète, non ?

On pourrait penser que les primitives de bases de gestion des verrous du noyau sont « mature », n'est-ce pas ? Et pourtant, elles viennent juste d'être réécrite de nouveau ces dernières semaines, allant plus vite et fonctionnant mieux dans des contextes qui n'avaient jamais été imaginés quand elles avaient été écrites pour la première fois ou même réécrites la dernière fois.

[Linux Mag] : Je ne pense pas que le terme « mature » doit être vu comme « obsolète ». Je comprends votre raisonnement, mais être mature ne signifie pas pour moi que l'on est voué à ne plus avancer. Évidemment, Linux a toujours besoin de s'adapter, mais je suppose que vous êtes plutôt satisfait aujourd'hui de la constitution de Linux. Trouver de nouvelles façons de mieux réaliser une fonction ou de résoudre un problème, ne veut pas toujours dire que la manière dont c'était effectué avant était mauvaise, n'est ce pas ? (Ce qui me vient à l'esprit comme analogie c'est la théorie de la relativité en physique qui parvient à capturer une plus grande part de la réalité que la mécanique classique. Et le passage de l'une à l'autre révèle l'évolution d'une même discipline. Mais la mécanique classique n'est pas moins mature pour autant, non ?)

[GKH] : Je ne pense pas que les modèles physiques conviennent tout à fait dans ce cas mais bien essayé :) La théorie de l'évolution en biologie est la meilleure description du noyau au cours du temps. Oui, je suis content de la constitution interne du noyau, mais il y a toujours de la place pour de l'amélioration. Il y a aussi des cas d'utilisation que nous ne connaissons pas encore et qui nécessiteront des modifications. Un exemple de ce que nous avons effectué, durant ces dix dernières années, est de permettre l'ajout ou la suppression à chaud de n'importe quel élément matériel dans un système en cours d'exécution. Mémoire, CPUs, disques, etc. C'était un changement majeur d'architecture qui couvait et que personne n'avait vu venir, et pourtant ce changement a du être réalisé afin de supporter de nouveaux systèmes et de répondre aux exigences des utilisateurs. Si tu m'avais demandé dix ans auparavant si je croyais que la structure interne du noyau était « suffisamment bonne », je l'aurais pensé mais j'aurais eu totalement tord. Et il va se produire la même chose dans les dix prochaines années. Je ne sais pas ce qu'il adviendra, mais des besoins externes à la communauté, diverses influences, feront que Linux devra changer et s'adapter à cet environnement s'il souhaite survivre.

Tout comme en biologie : adapte toi ou meurt. Mon objectif est de faire en sorte que Linux ne cesse de s'adapter. Ainsi, ce n'est pas une question d'être mature ou non, ce qui importe c'est la manière dont tu parviens à gérer le changement afin de survivre dans le futur.

[Linux Mag] : Et pour cela, la communauté doit être suffisamment mature ;) Mais c'est une autre histoire.

[GKH] : Ah, très bon point, je n'avais pas pensé à ça. Oui, notre communauté doit être mature et forte (et j'avancerais que la notre est une des plus fortes que l'on puisse trouver) afin d'être capable de gérer tous ces changements. Les BSDs sont très « mature » mais leur communauté n'est plus aussi mature ou suffisamment large et ils luttent en permanence avec un tas de choses aujourd'hui à cause de cela. Ce qui est triste, j'aime les BSDs, ils ont de fabuleux développeurs et leur code source est merveilleux, je n'ai pas envie de les voir échouer.

[Linux Mag] : Quels sont les parties du noyau qui vous captivent le plus aujourd'hui ?

[AM] : Je ne trouve pas vraiment le noyau « captivant ». C'est un projet de maintenance où la stabilité est primordiale. J'apprécie que l'on puisse changer autant de chose en si peu de temps et toujours en conservant l'ensemble raisonnablement stable et utile. Cela atteste d'un processus de développement qui a évolué de façon précautionneuse vers une forme très inhabituelle, ainsi que de ce qui est maintenant une équipe de développement hautement qualifiée.

[GKH] : Je veux seulement m'assurer que Linux fonctionne sur tous les nouveaux systèmes qui sont créés.

1.2 Entre flexibilité et maintenabilité

[Linux Mag] : Au sujet des « *control groups* », revenir à une hiérarchie unifiée *[NDLR: se référer à la section 2.1 pour plus de détails]* ne semble pas plaire à tout le monde (ex : les messages de Tim Hockin de chez Google – https://lkml.org/lkml/2013/4/22/478). Quel est votre sentiment sur ce sujet ?

[AM] : Je suis globalement dans un mode attentiste sur ce sujet. Cela serait vraiment inhabituel d'altérer une interface existante et les raisons de faire ainsi devraient être très pertinentes en effet.

[GKH] : Comme je ne suis pas un développeur des control groups, ni un utilisateur (directement), ce que j'en pense n'a pas beaucoup d'importance. Tu devrais plutôt contacter quelqu'un d'impliqué.

[Linux Mag] : Est ce qu'un compromis est toujours nécessaire entre flexibilité et maintenabilité ?

[AM] : Je deviens de plus en plus préoccupé par rapport à la complexité et à la facilité de compréhension du noyau. Plus nous le faisons complexe, et moins efficace deviens notre travail de tous les jours. Le risque le plus important est que nous introduisions des erreurs. Et cela crée un obstacle, toujours plus difficile à surmonter, pour les nouveaux développeurs noyau.

Je n'ai aucune bonne solution à cela. Je deviens de plus en plus réticent à accepter du code qui ajoute beaucoup de complexité, mais souvent le code est désirable et il est intégré de toute façon. Mon principal cheval de bataille est d'être de plus en plus exigeant sur la clarté du code, la qualité des commentaires du code et la qualité des informations fournies dans le journal des modifications. Cela ne réduit pas la complexité mais cela réduit, espérons, certains des dommages que la complexité cause.

[GKH] : Je ne pense pas qu'il y en ait un, le penses tu ?

[Linux Mag] : Mon observation est à rapprocher du précédent point sur le retour de la hiérarchie unifiée des control groups. Et il semble que dans certains cas, fournir de la flexibilité aux utilisateurs peut rendre la vie difficile à la maintenabilité du code. Ou peut-être est ce seulement qu'une bonne solution n'a pas encore été trouvée ?

[GKH] : Créer une bonne API pour l'espace utilisateur est difficile. Vraiment difficile. Tu travailles pendant un long moment, tu essaies de considérer tous les cas, tu crées l'API et ensuite tu l'utilises pendant un temps et tu réalises au final quelle est totalement à côté de la plaque. Les gens te hurlent dessus parce que tu l'as foirée. Et alors tu refais une nouvelle tentative, en itérant pendant quelques années pour tenter de l'avoir juste. C'est fait tellement rarement que ceux qui font ce travail habituellement, oublient comment ils l'ont fait la dernière fois ou bien ils ne l'ont jamais fait avant et pense que c'est facile à réaliser.

C'est même plus difficile que ça. Vraiment vraiment difficile. De nouveau, difficile.

On est juste en train de voir une partie de cette itération, c'est normal. Et si personne ne te crie dessus, alors cela signifie que ton API est mauvaise.

[Linux Mag] : Je comprends la difficulté à obtenir une bonne API, mais concernant le fil de discussion que je mentionnais (https://lkml.org/lkml/2013/4/22/478), et plus particulièrement le débat entre Tim Hockin et Tejun Heo, on peut faire il me semble le constat suivant (qui n'est pas forcément spécifique aux « control groups ») et j'apprécierai avoir ton opinion dessus :

1. Compromis nécessaire entre flexibilité et maintenabilité/complexité.

Dans le cas des « control groups », ils seront moins flexible (les deux protagonistes sont d'ailleurs d'accord sur ce point) et Tejun Heo ne veut pas croire que cela puisse poser problème (sans une argumentation étoffée et un exemple très détaillé à l'appui) même si, a priori, un cas réel (celui de Google) va être affecté négativement.

– Extrait du fil de discussion sur la Linux Kernel Mailing List (LKML) :

Tim Hockin : « Nous avons essayer avec une hiérarchie unifiée. Nous avions notre Meilleur Personnel sur le problème. Et le mieux que nous pouvions obtenir était suffisamment mauvais que nous nous sommes embarqués sur une transition [NDLR : vers une architecture à plusieurs hiérarchies] de 2 ans LITTERALLEMENT, pour améliorer la situation »

Tejun Heo : « Qu'est ce qui n'a pas fonctionné ? Quelles parties était trop mauvaises ? Je trouve cela très difficile à croire que de multiples hiérarchies orthogonales soit la seule solution possible. Aussi, merci de développer sur les problèmes que vous avez rencontrés. »

Tim Hockin : « Je suis en train de reboucler avec plus de personnel Google »

[GKH] : Le cas d'utilisation de Google est un cas difficile à justifier, étant donné qu'il est fermé, personne ne sait à quoi il ressemble, et la façon dont il est utilisé est inconnu. On pourrait argumenter en disant qu'il a été créé relativement au design actuel, et que si le design actuel est cassé (comme certains le soutiendraient), alors leur façon de l'utiliser pourrait également être « cassé ».

Mais c'est délicat, et difficile, et ce ne sont que les problèmes techniques. Jeter les gens dans la pagaille avec toutes leurs émotions et leur stress, est la recette pour aboutir à des difficultés même si tout le monde est en fait d'accord :)

Et je connais Tejun, je ne pense pas qu'il rejette quoique ce soit de façon fortuite. Je lui fais aussi confiance pour aboutir à ce qui sera la bonne solution, c'est pourquoi il est à la place qu'il est *[NDLR : mainteneur des « control groups »]*, parce que la communauté lui fait confiance.

[Linux Mag] : 2. Le manque de volonté d'accepter ou de considérer d'autres façons de faire, quand tu as décidé que c'était LA bonne façon de faire.

A ce sujet, je veux bien comprendre Tim Hockin, qui semble faire face à quelqu'un (Tejun Heo) qui ne changera pas d'avis sans une explication très détaillé de ce qui peut-être « va de travers » et qui ne cherchera pas à travailler sur une nouvelle approche (avec les gens de chez Google par exempel) qui ramènerait la précédente flexibilité.

– Extrait du fil de discussion sur la LKML :

Tim Hockin : « Ainsi, ce que tu es en train de dire est que tu ne te soucies pas du fait que cette nouvelle chose soit moins capable que l'ancienne, même s'il en ressort un réel impact. »

Tejun Heo : « En quelque sorte. Au moins jusqu'à maintenant, se débarrasser du support des hiérarchies orthogonales semble être le bon compromis. Tout dépend de la façon dont on mesure le niveau de simplification des choses par rapport au poids des « impacts réels ». Ce n'est pas comme si cela pouvait être catégorisé en blanc ou noir. Etant donné le contexte actuel, je pense que c'est la bonne façon de faire. »

[GKH] : Peut être que la flexibilité était mauvaise :)

C'est difficile de prendre une décision, il n'y a pas plus de dix personnes au monde qui comprennent vraiment ces trucs. Je ne suis certainement pas l'un d'entre eux et je n'ai pas envie d'être l'un d'eux. J'ai à m'inquiéter et à travailler sur d'autres problèmes.

Laissons Tejun et Tim et tous les autres travailler ensemble pour mettre tout cela dans le bon ordre. Je suis persuadé qu'à la fin, nous aurons quelque chose de meilleur que ce que nous avons actuellement. Et c'est le principal objectif, parce que tout le monde s'accorde à dire que ce que nous avons aujourd'hui n'est pas acceptable.

[Linux Mag] : Dans certains cas d'élégantes solutions sont trouvées et permettent de résoudre toutes les préoccupations. Vous souvenez vous d'une impasse technique que vous avez réussi à surmonter avec la communauté ?

[AM] : Hm. Pendant les 5-10 premières années de mon activité de développement noyau, nous cherchions toujours à fixer des problèmes de mauvaise utilisation de verrous et des blocages résultant. Souvent fondé sur un rapport d'utilisateur remontant un blocage noyau. L'infrastructure noyau d'auto-test « lockdep » *[NDLR : apparue dans le noyau 2.6.18, et étendu par la suite, cf. Kernel Corner 86]*, qui automatise la majorité des tests pour de tels bugs, a drastiquement réduit la fréquence à laquelle les utilisateurs font l'expérience de tels blocages noyau et remontent ce type de problèmes. Je pense que *lockdep* a été une innovation remarquable qui a permis d'obtenir une complexité plus élevée, tout en accélérant le développement et en augmentant la fiabilité du noyau.

[GKH] : Rien ne me reviens en mémoire pour l'instant. Je pense avoir besoin de boire un peu plus de vin pour me souvenir :)

[Linux Mag] : Merci beaucoup pour tout le temps que vous avez passé à répondre à mes questions.

2 Interview de Tejun Heo sur la refonte de l'infrastructure des Control Groups

2.1 Bref aperçu de la refonte des Control Groups

L'infrastructure des « *control groups* », ou **cgroups** (*cf.* Kernel Corner 101 et 127), permet de partitionner l'ensemble des tâches du système en des ensembles, appelés *control groups*, sur lesquels peuvent alors s'appliquer différentes politiques de contrôle sur les ressources du système (**memcg**, **cpuset**, **blkio**, etc.). Pour chaque type de ressources, des contrôleurs spécifiques ont été développés (ils sont actuellement au nombre de 13).

Le problème est que ces contrôleurs ont souvent été construits sans implication des développeurs les plus familiers avec les sous-systèmes avec lesquels ils interagissent. Et afin de faciliter l'intégration de ces contrôleurs dans la *mainline*, la stratégie a été de ne pas être trop intrusif, ce qui n'a évidemment pas favorisé l'implication des mainteneurs des différents sous-systèmes contrôlés. Mais ces derniers montraient également de la réticence à accepter des patchs pour ajouter le support des **cgroups** (lesquels introduisent nécessairement des pertes de performance lorsque qu'ils sont activés), d'où une situation de blocage, où les contrôleurs ont évolués chacun de leur côté et pas vraiment de façon toujours cohérente ni robuste.

La nature hiérarchique des **cgroups** signifie que les utilisateurs peuvent changer les permissions des sous-répertoires et y donner accès à des utilisateurs non-privilégiés. Cela signifie donc qu'une application peut interagir directement avec le système de fichiers des **cgroups** et accéder aux commandes de contrôle noyau de ces **cgroups**. Or ces commandes sont exposées à l'API noyau sans pour autant avoir subi une revue suffisante de leur code (étant donné le manque d'implication des mainteneurs). Afin de résoudre des problèmes fondamentaux existants (comme le fait de pouvoir bloquer complètement le noyau en créant une « infinité » de sous-cgroup à partir d'un niveau donné) ainsi que d'autres problèmes latents, les développeurs noyau mettent en œuvre, entre autres, une hiérarchie unifiée au sein des cgroups et améliorent la cohérence entre les différents contrôleurs.

Aussi, la gestion de cette hiérarchie ne devrait plus pouvoir passer que par un seul gestionnaire, rendant ainsi impossible la délégation directe de la gestion de sous-hiérarchies (mais réduisant par là-même la surface d'attaque des **cgroups**, qui d'un point de vue sécurité ne peut être qu'une bonne chose). Avec **systemd** par exemple (actuellement le seul gestionnaire adapté), il faudra alors effectuer des requêtes D-Bus pour gérer cette hiérarchie.

Cette hiérarchie unifiée dispose toutefois d'une granularité qui pourra être adaptée en fonction des contrôleurs. Ainsi, la finesse des sous-groupes de processus créés pour un contrôleur donné pourra très bien être vue et donc gérée de façon plus grossière pour un autre.

L'histoire ne s'arrête pas là, la nouvelle infrastructure va subir d'autres itérations avant d'aboutir à ce qui deviendra quelque chose de plus robuste, de plus fiable, de plus maintenable et de plus sécurisé. Notons aussi, que l'ancienne API sera maintenue durant plusieurs années à partir du moment où la nouvelle entrera en jeu. Donc, pas d'inquiétude sur l'adaptation des cas d'utilisation actuels qui pourra être réalisée sereinement.

2.2 Interview de Tejun Heo

Au sujet de certaines inquiétudes sur la nouvelle infrastructure des « control groups », attisées à la lecture de l'échange entre Tim Hockin et Tejun Heo sur le LKML (https://lkml.org/lkml/2013/4/22/478), nous avons souhaité poser quelques questions à Tejun Heo, afin de mieux comprendre la porté des changements en cours.

Les propos ont été recueillis en septembre et octobre 2013.

[Linux Mag] : Mes questions se concentrent sur la refonte des control groups. Après avoir lu le débat entre toi et Tim Hockin (https://lkml.org/lkml/2013/4/22/478), j'apprécierai grandement avoir des explications plus détaillés sur certains points que je ne saisis pas complètement.

Je débute avec la citation suivante :

Tejun Heo : « C'est quelque chose de fondamentalement cassé et j'ai vraiment du mal à croire que la charge de Google est tellement différente qu'elle ne peut être catégorisée en une seule hiérarchie pour ce qui est de la distribution de ressources ».

– extrait de la LKML (https://lkml.org/lkml/2013/6/24/676)

Pourquoi la catégorisation en de multiples hiérarchies est « fondamentalement cassé » ? (les accès réseau et le temps CPU peuvent quasiment être vu comme des ressources orthogonales, non ?)

Tejun Heo [TH] : Le problème est qu'il est impossible de dire qu'une ressource donnée appartient à un cgroup car il y a de multiple liens d'appartenance. Cela introduit de fait une autre couche d'appartenance dans la gestion des ressources, dans laquelle est primordial, l'espace des permutations des appartenances à toutes les hiérarchies, dont le nombre n'est pas limité. Cela devient pénible lorsqu'il devient nécessaire de décrire ou de communiquer l'appartenance au cgroups – *[NDLR : dans l'infrastructure actuelle]* la hiérarchie entre deux ensembles d'appartenance n'est pas définie de façon indépendante de la hiérarchie considérée et l'aspect majeur est qu'elles sont variables et non bornées.

Cela rend le marquage et la gestion d'une ressource (dans le cas où elle est pertinente pour différents contrôleurs) ou encore cela rend la définition de l'API de l'espace utilisateur qui traite de l'appartenance au cgroups, très alambiqué et cela augmente la complexité à un niveau où il devient très incommode de mettre en œuvre les fonctionnalités souhaitables qui auraient été autrement possible d'introduire.

De multiples hiérarchies ont donné naissance à des utilisations qui vont bien au delà de ce qui fait parti du périmètre fondamental qui est du ressort de la gestion de ressources. Il y a des contrôleurs qui ne font parti des cgroups que pour utiliser le mécanisme de groupement des tâches. Le réseau est en fait un bon exemple car tout ce que le contrôleur fait se résume à taguer les sockets alors que cela aurait du être réalisé de l'autre façon. Ce contrôleur aurait du avoir pour rôle de configurer la pile réseau en fonction de l'appartenance au cgroup, mais étant donné qu'il n'y a pas de tel lien d'appartenance de défini, à cause des hiérarchies multiples, cela a abouti à l'utilisation (abusive) de multiples hiérarchies pour seulement taguer les sockets, qui à leur tour interfèrent avec le réel objectif de la gestion des ressources.

[Linux Mag] : Il me semble (en tant que lecteur extérieur au débat) qu'aller vers une hiérarchie unifiée risquerait d'être un fardeau pour l'espace utilisateur, en ce qui concerne la définition de certains types de politiques de contrôle de ressources, lesquelles pourraient être spécifiées bien plus facilement avec de multiples hiérarchies. Quel est votre position sur ce sujet ?

[TH] : Toutes sont étroitement liées et se nourrissent mutuellement. Parce que nous avions ce truc trop flexible, différentes utilisations s'y sont développées, qui à leur tour ont figées et amenées au niveau suivant le besoin d'une telle flexibilité. Cette évolution est une chose naturelle mais je

crois fermement qu'on en est arrivé à un point où cela ne peut plus être considéré comme salutaire.

Pour la plupart des cas d'utilisation, je pense qu'il est très improbable que la transition nécessite beaucoup trop de charge au niveau de l'espace utilisateur. Il est vrai que certains cas d'utilisation extrême requerreront de la complexité dans l'espace utilisateur. Toutefois, c'est en considérant que le schéma d'utilisation actuel ne changera pas, ce que je ne pense pas probable.

Les choses vont très certainement se restructurer autour de la nouvelle interface et tant que la nouvelle interface parvient à accomplir les fonctionnalités principales de façon bien raisonnable, je ne pense pas que la perte de cette flexibilité supplémentaire nous desservira à terme. Il est vrai qu'il y aura des points épineux à traiter mais la période de transition durera des années et le mode d'utilisation des cgroups n'en est encore qu'à ces balbutiements. Je crois que c'est le bon moment pour corriger notre trajectoire.

[Linux Mag] : Cela implique également une perte d'expressivité de la nouvelle API, car nous perdons une dimension dans « l'espace du contrôle de ressources ». Mais la question, comme tu l'expliques dans tes *mails*, est de mesurer les conséquences de cette perte par rapport aux bénéfices qui en découle sur la maintenabilité, la simplicité (peut-être sur d'autres aspects également, à mon avis la sécurité en fait parti). Ainsi, ma question est de savoir ce qui te rassure avec l'idée qu'il s'agit d'un bon compromis ? Peux tu partager certains faits qui t'ont convaincu ?

[TH] : Ce n'est pas seulement la complexité de l'implémentation. Si un aspect est poussé trop loin, il ne fait que limiter et déformer les autres, presque toujours. Par exemple, les hiérarchies multiples empêchent d'identifier un lien d'appartenance d'une façon simple ; inconvénient qui, à son tour, limite des choses qui pourraient avoir plus de valeur que l'extrême flexibilité. Comme toute chose en ce monde, il y a du sens à diminuer le « rendement » de la plupart des paramètres de conception. Les cgroups ont poussé un paramètre, la flexibilité, bien au delà du point d'équilibre et cela au détriment des autres.

Tout le travail effectué sur l'interface v2 a pour but de restaurer un bon équilibre dans l'espace de conception en contenant la flexibilité qui est allée trop loin. Un bon équilibre pour l'un, en est un mauvais pour l'autre, c'est pourquoi je ne suis pas sûr de pouvoir fournir des éléments montrant que la flexibilité est allé trop loin, mais au moins la plupart des personnes familières des cgroups et de sa base de code semble d'accord sur le fait que c'est allé trop loin.

[Linux Mag] : Et mon avant dernière question peut être introduite par cette citation :

Tejun Heo : « Je suppose que c'est ici où nous ne sommes pas d'accord. Je pense que nombre des problèmes des cgroups proviennent de trop de flexibilité. Le problème avec autant de flexibilité est qu'en plus de casser des constructions fondamentales et d'ajouter une charge additionnelle et significative à la maintenance, il empêche la réalisation de bon compromis aux bons endroits, à son tour requérant plus de 'flexibilité' pour corriger les carences introduites »

– extrait de la LKML (https://lkml.org/lkml/2013/6/27/517)

Est ce qu'un compromis est toujours nécessaire entre flexibilité et maintenabilité ?

[TH] : « Toujours » est un mot qui ne permet pas la nuance, mais dans la plupart des cas, il n'y a pas vraiment tant d'espace que ça pour manoeuvrer et pousser un paramètre à une extrémité contraint évidemment les autres. Au delà d'un certain point, il n'est même plus possible de trouver un compromis. Cela me rappelle la « squeezed light » *[NDLR : lumière pressée]*, qui augmente soit la précision de l'amplitude soit la précision de la fréquence mais au détriment de l'autre. C'est fascinant de voir que les êtres humains sont déjà en train d'utiliser cette chose et d'en avoir un usage :

http://www.squeezed-light.de/

[Linux Mag] : Enfin, est ce que la nouvelle infrastructure des cgroups sera utilisable au sein de l'infrastructure des espaces de noms ? (peut-être pas immédiatement, mais à court/moyen terme ?) Et dans ce cas peux tu partager quelques détails ?

[TH] : A terme, oui. Le noyau n'ira pas à l'encontre de cela en ne faisant qu'empêcher le `chmod` sur les sous-répertoires des cgroups [NDLR : *cf.* section 2.1]. L'implémentation en espace utilisateur requerrera du travail cependant. Je pense que l'on est toujours dans une phase de découverte. Donc il peut se passer du temps avant que cela ne se trouve dans le design final, mais je suis plutôt confiant sur le fait que tout sera en place à temps.

J'espère avoir au moins répondu à certaines des questions. Merci !

[Linux Mag] : Merci pour ton éclairage sur ces points dont les tenants et aboutissants ne sont pas toujours facile à saisir. ■

ANDROID : WEBKIT OFFLINE

par Philppe PRADOS [Consultant OCTO Technology]

HTML5 permet de gérer le mode hors-ligne lors de la consultation d'un site. Mais cela ne fonctionne que si l'utilisateur navigue sur la page une première fois et y revient lorsque le réseau est disponible. Cela permet au code Javascript de se synchroniser avec le back-end. Dans une application mixte : Android + Webkit, ce n'est pas satisfaisant.

Une application mixte est une application qui utilise les API natives pour certains écrans et des composants WebKit pour d'autres.

Cela s'applique à des fonctionnalités très volatiles comme la diffusion de publicités, l'aide en ligne, la description de processus spécifiques à une prestation, la diffusion d'offres promotionnelles de dernières minutes, etc. Pour toutes ces situations, une application native n'est pas la solution. Il est nécessaire d'avoir une souplesse maximum lors des mises à jours. Il ne faut pas modifier et republier une application mobile pour chaque mise à jour d'un écran. Qui souhaite faire cela parce qu'un produit passe de 30 % de réduction à 50 % pour un lot de deux ou que la promotion change de produit ?

Webkit est alors la solution pour marier les technologies natives et web. Webkit est le moteur HTML utilisé par Apple, Google et bien d'autres. Un composant **WebView** est ajouté à une activité. Ce dernier exploite le réseau comme le fait un simple navigateur, pour présenter les pages en perpétuelles évolutions.

Prenons l'exemple d'un site marchant qui souhaite pouvoir proposer des promotions pour écouler ses stocks. La mise en valeur de ces dernières est très variable. Parfois, c'est la combinaison de plusieurs produits, parfois l'utilisation d'un code de réduction spécifique, etc. Il n'est pas possible de concevoir une activité spéciale pour tenir compte de cette très grande variabilité. C'est pour cela qu'un site Web sera toujours plus à jour qu'une application correspondante. Utilisez fnac.com plutôt que l'application Windows 8 correspondante. Vous aurez probablement plus d'offres.

Notre application propose donc d'utiliser une **WebView** dans une activité.

Avec HTML5, il est possible d'indiquer la liste des pages à pré-charger, pour une utilisation hors ligne. C'est facile à faire. Il faut ajouter un attribut **manifest** dans le marqueur **<html/>** avec un lien vers un fichier au format **manifest**.

Attention, le type **mime** associé doit obligatoirement être **text/cache-manifest**. Vérifiez les paramètres de votre serveur HTTP si le cache n'est pas mis à jour.

```
<html manifest="cache.manifest">
...
</html>
```

Le fichier **cache.manifest** indique la liste des URL à charger dans le cache. Reportez-vous à Internet pour le détail de la syntaxe. C'est très simple : une catégorie puis une URL par ligne.

Le composant **WebView** doit alors être initialisé pour accepter l'utilisation des caches par les applications Web.

```
WebSettings settings = webView.getSettings();
settings.setJavaScriptEnabled(true);
settings.setDomStorageEnabled(true);
settings.setDatabaseEnabled(true);
settings.setSaveFormData(true);
settings.setAllowFileAccess(false);
settings.setAppCacheEnabled(true);
settings.setAppCachePath(context.getCacheDir().getAbsolutePath());
if (VERSION.SDK_INT < VERSION_CODES.JELLY_BEAN_MR2)
  settings.setAppCacheMaxSize(1024 * 1024 * 8);
```

Il est alors possible d'indiquer une stratégie pour l'exploitation du cache par le composant.

```
settings.setCacheMode(WebSettings.LOAD_CACHE_ELSE_NETWORK);
```

Et en effet, après avoir affiché l'activité une première fois, le cache de l'application est alimenté par une tâche de fond du **WebView**. Il est donc possible d'utiliser l'activité hors ligne.

1 Vraiment hors ligne ?

En fait, il y a une première difficulté. Tant que l'activité n'a pas été affichée, le cache n'est pas alimenté. Donc, si l'utilisateur n'a pas de réseau lors du premier accès à l'activité, il

reçoit une belle page d'erreur indiquant qu'il est impossible de présenter la page demandée.

Le premier réflexe dans cette situation est d'utiliser une page en dure dans l'application et de l'utiliser comme première page. C'est un poil meilleur, car on maîtrise alors le message d'erreur affiché dans la `WebView`, mais cela ne règle pas le problème. La page n'est pas affichée. Il est probable que le réseau était disponible précédemment. Il aurait été alors possible d'alimenter le cache.

Le deuxième problème à régler est le suivant : tant que la page n'est pas affichée, le cache n'est pas rafraîchi. Donc, s'il est hors ligne, l'utilisateur consulte la dernière version connu par l'application et non la dernière version publiée. C'est dommage. On utilise justement une `WebView` pour des pages volatiles, pouvant évoluer régulièrement. Une mise à jour régulière du cache serait un plus.

Un troisième problème : HTML5 permet au JavaScript de contrôler quand le réseau devient actif. Cela permet la publication vers le back-end des données maintenues dans le cache applicatif de la page. Mais si la page n'est jamais affichée par l'utilisateur ? Les données restent dans le cache ! Comment forcer leurs publications par les applications Web ?

Pour toutes ces raisons, l'utilisation des caches de HTML5 dans une activité est rarement pratiquée. Le composant ne peut fonctionner qu'avec le réseau. Et c'est un pan entier des spécifications HTML5 qui ne sert plus à rien.

Pour régler ces trois situations, il faudrait synchroniser le cache HTML d'une application, sans intervention de l'utilisateur. C'est exactement ce que je vous propose dans les lignes suivantes. Le chemin est long, pleins d'embûches, mais au final, le code est court, simple et facile à intégrer dans une application quelconque.

Les fichiers sources sont présents ici : https://github.com/pprados/android-webview-async-cache.

2 Le framework de synchronisation d'Android

Synchroniser des données n'est pas évident si on ne souhaite pas drainer la batterie.

En effet, il y a un automate à états pour la connexion radio qui fait passer le composant dans des états rapides mais consommateurs ou lents mais économes. Le passage d'un état à un autre s'effectue après le déclenchement de temporisations, différentes suivant les technologies GSM, 3G ou 4G utilisées. Si votre code n'en tient pas compte, il peut réactiver le mode rapide mais consommateur trop souvent. La batterie ne tient alors pas très longtemps. De plus, chaque transition d'état prend un certain temps qui ralenti au final l'expérience utilisateur.

Il est alors préférable d'utiliser plusieurs stratégies pour optimiser tout cela. Par exemple, il est préférable de récupérer le maximum d'informations du back-end dans la même session. Cela évite les cycles dans la gestion de la radio. De même, pour la publication des modifications, il est préférable de le faire en rafale. Il ne faut pas publier une modification immédiatement mais attendre une quinzaine de secondes avant de le faire. Il est fort probable que pendant cette période, d'autres modifications vont intervenir. Si l'application se met en « petite-veille », avec réduction de la luminosité de l'écran, il ne semble pas nécessaire de rafraîchir les données. Par contre, lorsque l'utilisateur intervient à nouveau sur l'application, il peut être judicieux d'intervenir.

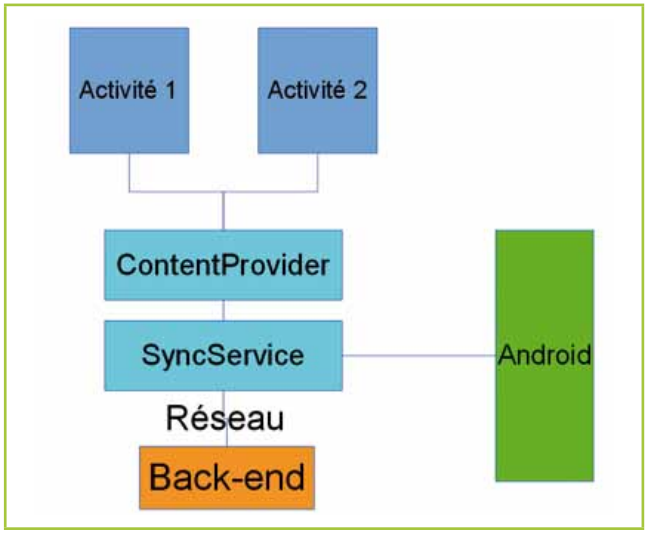
Si une connexion Wi-Fi est disponible, grouper les lectures et les écritures est également une bonne stratégie.

Gérer tous ces scénarios n'est pas facile. Cela demande une connaissance fine des automates à états des modules radios et Wi-Fi, de suivre ces états, etc.

Pour remédier à cela, Android propose un framework. Vous en avez connaissance en regardant le détail des différents comptes de votre Android. Vous pouvez cocher ou décocher les différentes synchronisations associées à un compte.

Ce framework est très bien adapté à un type d'architecture que je vous conseille d'appliquer pour tous vos projets. Google utilise ce modèle pour la plupart de ses applications. Vous pouvez le constater en regardant le nombre de service dont la synchronisation est activable, à partir d'un compte Google. Chaque service utilise ce modèle d'architecture.

Le modèle est le suivant : les applications et les activités ne s'occupent pas du réseau. Elles récupèrent des informations d'un `ContentProvider` et apportent des modifications dans ce dernier. La base de données associée au `ContentProvider` est un cache des données présentes sur le serveur.



Par exemple, le **ContentProvider** peut mémoriser des flux RSS, des mails, des contacts d'un annuaire, des flux d'un réseau social, etc.

Pour synchroniser la base de données locales, portées par le **ContentProvider**, un **AbstractThreadedSyncAdapter** se charge de cela. Il est invoqué par Android lorsqu'il estime que les meilleures conditions sont réunies.

Pour utiliser ce framework, il faut réunir trois composants :

- un **Account**. C'est un compte utilisateur comme vous pouvez les voir dans les paramètres d'Android.
- Un **ContentProvider**. C'est le composant qui porte le cache. La déclaration de la synchronisation s'effectue en associant un **Account** avec un **ContentProvider**.
- un service en charge de la synchronisation.

Ce n'est donc pas facile de réunir toutes ces conditions.

En fait, ce n'est pas si compliqué. L'**Account** n'a pas besoin d'être réel. Il ne doit pas nécessairement porter l'identité d'un utilisateur.

Ne vous inquiétez pas de la suite de l'article, il vous explique comme faire « à la main ». À la fin, je vous propose des classes permettant de simplifier toutes ces étapes.

Avant toute chose, il faut choisir trois constantes : l'**accountType** qui décrit le type de compte de manière unique dans l'OS, l'**accountName** qui décrit le nom de l'utilisateur pour ce type de compte et l'autorité d'un **ContentProvider**.

```
public static final String ACCOUNT_TYPE="fr.prados.sync"
public static final String ACCOUNT_NAME="sync"
public static final String AUTHORITY="fr.prados.webkitcache"
```

2.1 Première étape : créer un compte

La première chose à faire est de créer un compte utilisateur. Il faut publier un service spécifique et le paramétrer avec un fichier XML. Placez le fichier **authenticator.xml** suivant dans le répertoire **res/xml** de votre projet.

```
<?xml version="1.0" encoding="utf-8"?>
<account-authenticator
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:accountType="fr.prados.sync"
  android:icon="@drawable/ic_launcher"
  android:smallIcon="@drawable/ic_launcher"
  android:label="@string/app_name"
  />
```

Bien entendu, le paramètre **accountType** doit être adapté à ce que vous avez choisi.

Il faut ensuite publier un service dans le **AndroidManifest.xml**.

```
<service android:name=".SyncHTTPService"
  android:exported="false"
>
  <intent-filter>
    <action android:name="android.accounts.AccountAuthenticator" />
  </intent-filter>
  <meta-data
    android:name="android.accounts.AccountAuthenticator"
    android:resource="@xml/authenticator" />
</service>
```

Dans la classe du service, déclarez un **AbstractAccountAuthenticator** vide.

```
private AbstractAccountAuthenticator mAuthenticator;

// Dummy authenticator
private static class AccountAuthenticator
 extends AbstractAccountAuthenticator
{
  AccountAuthenticator(Context context)
  {
    super(context.getApplicationContext());
  }
  @Override
  public Bundle editProperties(
    AccountAuthenticatorResponse accountAuthenticatorResponse,
    String s)
  {
    throw new UnsupportedOperationException();
  }

  @Override
  public Bundle addAccount(
    AccountAuthenticatorResponse accountAuthenticatorResponse,
    String s,
    String s2,
    String[] strings,
    Bundle bundle)
  throws NetworkErrorException
  {
    return null;
  }
  ...
};
```

Toutes les méthodes retournent **null** ou génèrent une exception.

Pour propager le **Context**, l'instance **AccountAuthenticator** est créé dans la méthode **onCreate()** du service.

```
@Override
public void onCreate()
{
  mAuthenticator=new AccountAuthenticator(this);
}
```

Abonnez-vous !

Téléphonez au 03 67 10 00 20 ou commandez par le Web

Consultez l'ensemble de nos offres sur : boutique.ed-diamond.com !

11 Numéros de GNU/Linux Magazine

60€*

au lieu de 86,90 €* en kiosque

Économie 26,90€

Économisez plus de 30%*

* Sur le prix de vente unitaire France Métropolitaine

Nouveauté 2014
TOUS LES HORS-SÉRIES PASSENT EN GUIDE !

Un nouveau format avec une reliure de luxe pour ces Guides de référence de 128 pages à conserver dans votre bibliothèque !

Découvrez tous les Guides déjà parus sur

boutique.ed-diamond.com

*OFFRE VALABLE UNIQUEMENT EN FRANCE MÉTROPOLITAINE. Pour les tarifs hors France Métropolitaine, consultez notre site : boutique.ed-diamond.com

Les 3 bonnes raisons de vous abonner :

- Ne manquez plus aucun numéro.
- Recevez GNU/Linux Magazine chaque mois chez vous ou dans votre entreprise.
- Économisez 26,90 €/an ! (soit plus de 3 magazines offerts !)

4 façons de commander facilement :

- par courrier postal en nous renvoyant le bon ci-dessous
- par le Web, sur boutique.ed-diamond.com
- par téléphone, entre 9h-12h et 14h-18h au 03 67 10 00 20
- par fax au 03 67 10 00 21

Bon d'abonnement à découper et à renvoyer à l'adresse ci-dessous

Tournez SVP pour découvrir toutes les offres d'abonnement >>

Voici mes coordonnées postales :	
Société :	
Nom :	
Prénom :	
Adresse :	
Code Postal :	
Ville :	
Pays :	
Téléphone :	
e-mail :	

❑ Je souhaite recevoir les offres promotionnelles et newsletter des Éditions Diamond.

❑ Je souhaite recevoir les offres promotionnelles de nos partenaires.

GNU LINUX MAGAZINE / FRANCE

Édité par Les Éditions Diamond
Service des Abonnements
B.P. 20142 - 67603 Sélestat Cedex
Tél. : + 33 (0) 3 67 10 00 20
Fax : + 33 (0) 3 67 10 00 21

Vos remarques :

En envoyant ce bon de commande, je reconnais avoir pris connaissance des conditions générales de vente des Éditions Diamond à l'adresse internet suivante : **boutique.ed-diamond.com/content/3-conditions-generales-de-ventes** et reconnais que ces conditions de vente me sont opposables.

Tournez SVP pour découvrir toutes les offres d'abonnement

Abonnez-vous !

➡ **Tous les abonnements incluant GNU/Linux Magazine :**

* Tarifs France Métro (F) ** Base tarifs kiosque zone France Métro (F)

Consultez l'ensemble de nos offres d'abonnements sur :

boutique.ed-diamond.com

Vous pouvez également commander par Tél. : +33 (0)3 67 10 00 20 / Fax : +33 (0)3 67 10 00 21

➡ Nos Tarifs s'entendent TTC et en euros		F	OM1	OM2	E	RM
		France Métro	Outre-Mer		Europe	Reste du Monde
LM	Abonnement GLMF	60 €	75 €	96 €	83 €	90 €
LM+	Abonnement GLMF + GLMF HS (6 Guides)	115 €	147 €	190 €	160 €	173 €
T	Abonnement GLMF + MISC + OS + LP + LE	198 €	253 €	325 €	276 €	300 €
T+	Abonnement GLMF + GLMF HS (6 Guides) + MISC + MISC HS + OS + LP + LP HS (3 Guides) + LE	299 €	382 €	491 €	415 €	448 €

• OM1 : Guadeloupe, Guyane française, Martinique, Réunion, St Pierre et Miquelon, Mayotte

• OM2 : Nouvelle Calédonie, Polynésie française, Wallis et Futuna, Terres Australes et Antarctiques française

MA FORMULE D'ABONNEMENT :

Offre	Zone	Tarif
☐ LM		
☐ LM+		
☐ T		
☐ T+		

J'indique la somme due : (Total) €

Exemple :
Je souhaite m'abonner à l'ensemble des magazines + tous les Hors-séries/Guides et je vis en Belgique. Je coche donc l'offre **T+** (la totale avec tous les Hors-Séries/Guides) puis ma zone (E), le montant sera donc 415 euros.

Je choisis de régler par :

☐ Chèque bancaire ou postal à l'ordre des Éditions Diamond

☐ Carte bancaire n° |_|_|_|_| |_|_|_|_| |_|_|_|_| |_|_|_|_|

Expire le : |_|_|_|_|

Cryptogramme visuel : |_|_|_|

Date et signature obligatoire

Il reste à répondre à la méthode **onBind()** du **Service**.

```
@Override
public IBinder onBind(Intent intent)
{
  final String action = intent.getAction();
  if (action.equals("android.accounts.AccountAuthenticator"))
  {
    return mAuthenticator.getIBinder();
  }
  else
    return null;
}
```

Nous pouvons maintenant créer un **Account** automatiquement.

```
/**
 * Create an empty account.
 *
 * @param context The context
 * @return The account.
 */
private static Account initAccount(Context context)
{
  final AccountManager accountManager =
    (AccountManager) context.getSystemService(Context.ACCOUNT_SERVICE);
  final Account account=new Account(ACCOUNT_NAME,ACCOUNT_TYPE);
  Account[] accounts=accountManager.getAccountsByType(account.type);
  boolean find=false;
  for (int i=0;i<accounts.length;++i)
  {
    if (accounts[i].name.equals(account.name))
    {
      find=true;
      break;
    }
  }
  if (!find)
  {
    Log.d(TAG,"add new account "+account.name+
          " type "+account.type+" for synchronization");
    if (accountManager.addAccountExplicitly(account, null, null))
    {
      ...
    }
  }
  return account;
}
```

La première étape est terminée. Un compte est ajouté si ce n'est déjà fait.

2.2 Deuxième étape : publier un ContentProvider

Nous devons exposer un **ContentProvider** pour l'autorité choisie. Comme pour l'**AccountProvider**, nous déclarons une classe interne à notre service avec un **ContentProvider** vide.

```
public class SyncHTTPService extends Service
{
  // Dummy content provider for synchronize the WebKit cache.
  public static class CacheHTTPContentProvider extends ContentProvider
  {
    @Override
    public boolean onCreate()
    {
      return true;
    }

    @Override
    public int delete(Uri uri, String s, String[] as)
    {
      throw new UnsupportedOperationException("Not supported by this provider");
    }
    ...
  }
}
```

Dans le **AndroidManifest.xml**, on le déclare.

```
<!-- Provider to expose the cached pages
<provider
  android:name=".SyncHTTPService$CacheHTTPContentProvider"
  android:authorities="fr.prados.webkitcache"
 android:exported="false"
  android:syncable="true"
/>
```

Notez le paramètre **syncable** à **true**.

2.3 Troisième étape : ajouter la synchronisation

Pour s'intégrer dans le framework de synchronisation d'Android, il faut procéder de façon similaire à l'ajout d'un compte.

La première chose à faire est de publier le fichier **syncadapter.xml** dans le répertoire **res/xml**.

```
<?xml version="1.0" encoding="utf-8"?>
<sync-adapter
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:contentAuthority="fr.prados.webkitcache"
  android:accountType="fr.prados.sync"
  android:userVisible="false"
  android:supportsUploading="true"
  android:allowParallelSyncs="false"
  android:isAlwaysSyncable="true"/>
```

Le paramètre **userVisible** permet d'avoir une case à cocher dans les paramètres du compte.

En théorie, il faut déclarer un autre service avec ses paramètres, comme on vient de le faire pour la gestion du compte. Comme je suis fainéant et que j'aime bien

réduire le nombre de ligne de code d'un projet, je propose de recycler le service déjà présent pour lui faire jouer les deux rôles : la gestion du compte et la gestion de la synchronisation.

Pour cela, j'ajoute les paramètres suivant dans la déclaration du service.

```
<intent-filter>
  <action android:name="android.content.SyncAdapter" />
  </intent-filter>
<meta-data
  android:name="android.content.SyncAdapter"
  android:resource="@xml/syncadapter" />
```

J'ajoute ensuite une classe interne dérivée d'**AbstractThreadedSyncAdapter**

```
public class SyncHTTPService extends Service
{
  private class SyncAdapter extends AbstractThreadedSyncAdapter
  {
    public SyncAdapter(Context context, boolean autoInitialize)
    {
      super(context, autoInitialize);
    }

    @TargetApi(Build.VERSION_CODES.HONEYCOMB)
    public SyncAdapter(Context context,
      boolean autoInitialize,
      boolean allowParallelSyncs)
    {
     super(context, autoInitialize, allowParallelSyncs);
    }

    @Override
    public void onPerformSync(
      Account account,
      Bundle extras,
      String authority,
      ContentProviderClient provider,
      SyncResult result)
    {
      SyncHTTPService.this.onPerformSync(getContext(),
        account,
        extras,
        authority,
        provider,
        result);
    }
  };
  protected void onPerformSync(
    Context context,
    Account account,
    Bundle extras,
    String authority,
    ContentProviderClient provider,
    SyncResult result);
  {
    ...
  }
}
```

Pour simplifier l'implémentation de la méthode **onPerformSync()**, elle est propagée dans la classe externe, à savoir le service.

Il faut alors ajouter un aiguillage dans la méthode **onBind()** pour jouer sur les deux tableaux.

```
@Override
public IBinder onBind(Intent intent)
{
  final String action = intent.getAction();
  if (action.equals("android.accounts.AccountAuthenticator"))
  {
    return mAuthenticator.getIBinder();
  }
  else if (action.equals("android.content.SyncAdapter"))
  {
    return mSyncAdapter.getSyncAdapterBinder();
  }
  else
    return null;
}
```

Il reste à déclarer la synchronisation lors de la création du compte. Dans la méthode **initAccount()** précédente. Après la création du compte, nous ajoutons ces quelques lignes.

```
// Inform the system that this account supports sync
ContentResolver.setIsSyncable(account, CONTENT_AUTHORITY, 1);
// Inform the system that this account is eligible for auto sync when
the network is up
ContentResolver.setSyncAutomatically(account, CONTENT_AUTHORITY,
true);
// Recommend a schedule for automatic synchronization. The system
// may modify this based on other scheduled syncs and network
utilization.
ContentResolver.addPeriodicSync(account, CONTENT_AUTHORITY,
  new Bundle(), SYNC_FREQUENCY);
```

Et voilà. Nous avons un service qui est capable de jouer deux rôles. D'une part, il permet de gérer un compte et, d'autre part, d'être invoqué par Android lors d'une synchronisation.

2.4 Mutualisons tous cela

Dans les sources du projet et pour vous simplifier la vie, je propose la classe **AbstractSyncService** qui se charge de la tuyauterie. C'est un service qui propose un **AccountAuthenticator** et un **SyncAdapter**.

Pour l'utiliser, il faut en dériver et implémenter la méthode abstraite **onPerformSync()**. Cette classe ne propose pas de **ContentProvider**.

En indiquant votre classe dérivée dans la déclaration du service, vous avez le nécessaire pour recevoir une invocation lors d'une synchronisation de votre **ContentProvider**.

Vous pouvez alors séparer les couches de votre application. Les interfaces communiquent avec le **ContentProvider**. La communication réseau s'effectue uniquement dans la méthode **onPerformSync()**.

Ne vous inquiétez pas, il est possible de demander une synchronisation immédiate (s'il y a le réseau) ou de simplement signaler qu'il y a eu des modifications et qu'il serait bien de synchroniser les données. Regardez les méthodes de **ContentResolver**.

3 Comment utiliser cela pour synchroniser le cache WebView ?

Nous avons franchi une étape mais pas gagné la guerre. Notre objectif est d'implémenter la méthode **onPerformSync()** pour alimenter le cache WebKit de l'application. Cette méthode est invoquée dans un thread qui n'est pas l'UI thread. Normal. La synchronisation peut prendre un certain temps.

Pour alimenter le cache de WebKit, il n'y a pas trente-six moyens. Il faut créer un composant **WebKit** et lui demander de charger une page. Si cette dernière utilise le **manifest** HTML5, alors WebKit demande l'alimentation du cache en tâche de fond. C'est seulement lorsque le cache est intégralement mis à jours, que les modifications seront visibles.

Donc notre stratégie consiste à enregistrer une suite d'URL à charger dans un composant WebKit caché. Sur la réception de l'événement **onPageFinished**, l'URL suivante est demandée au composant. Lorsqu'il n'y a plus d'URL à rafraîchir, la synchronisation est terminée.

C'est bien beau mais il n'est pas possible de créer un composant WebKit si on n'est pas dans l'UI thread. Cela tombe mal, on est justement dans un autre thread. Après avoir hérité de notre classe abstraite précédente, nous créons alors un **Handler** dans l'UI thread.

```
public class SyncHTTPService extends AbstractSyncService
{
  private Handler mHandler=new Handler();
  private WebView mWebView;
  ...
}
```

Il va nous servir à déclencher notre scénario dans l'UI thread.

```
mHandler.post(new Runnable()
{
  private int mState=0;
  private String mUrl;

  @Override
  public void run()
  {
    ...
});
```

Dans la méthode **run()**, nous pouvons construire une instance **WebView** et invoquer sa méthode **loadUrl()** avec la première URL.

Attention, si la méthode **onPerformSync()** termine prématurément, le framework Android en conclus que la synchronisation est terminée. Il peut alors tuer le processus !

Nous devons alors ajouter une synchronisation entre l'automate en charge de lire les pages par le composant **WebView** et la méthode **onPerformSync()**. Ainsi, **onPerformSync()** ne termine qu'après avoir chargé toutes les URL dans la WebView.

Une dernière subtilité : lorsque tout est terminé, la **WebView** peut continuer à vivre et à animer les flashs, les vidéos, etc. Il faut invoquer la méthode **onPause()**. Malheureusement, cette dernière n'est disponible qu'avec Android Honeycomb. Pour les versions précédentes, elle est bien présente via l'introspection.

Comme nous avons un **ContentProvider**, nous pouvons notifier l'application que le cache a été synchronisé, URL par URL. Cela tombe bien, les **ContentProvider** utilisent des URL. La ligne suivante permet d'informer qui de droit :

```
getContentResolver().notifyChange(Uri.parse(mUrl), null,true);
```

Nous informons les applications s'étant enregistrées pour l'URL via un :

```
getContentResolver().registerContentObserver(url, true,mContentObserver);
```

Pour laisser un peu de temps au traitement du cache WebKit, la notification est légèrement retardée.

Je vous laisse étudier le code de la méthode **onPerformSync()**.

Dans les sources, la classe **SyncHTTPService** est immédiatement fonctionnelle. Il suffit de déclarer le service et le **ContentProvider** comme dans l'**AndroidManifest.xml** du projet.

Les méthodes **registerURL()** et **unregisterURL()** permettent de gérer la liste des URL à charger dans le cache. Si les pages correspondantes sont compatibles HTML5 et possèdent un attribut **manifest** dans le marqueur **<html/>**,

alors la suite est sous le contrôle du serveur HTTP. Je vous conseille de tester cela avec chrome ou un autre navigateur Web, pour vérifier que vous maîtrisez bien la gestion des caches HTML5.

4 Utilisation du cache

Nous pouvons alors utiliser une **WebView** en lui indiquant de n'utiliser que les données du cache.

```
mWebView.getSettings().setCacheMode(WebSettings.LOAD_CACHE_ONLY);
```

Ou bien, nous pouvons privilégier le cache et exploiter le réseau si nécessaire.

```
mWebView.getSettings().setCacheMode(WebSettings.LOAD_CACHE_ELSE_NETWORK);
```

Pourquoi utiliser le réseau si on peut s'en passer ? En condition difficile, la simple tentative de connexion au back-end pour savoir si une page du cache est toujours valide peut prendre un temps certain. Ceux qui utilisent leurs mobiles dans les trains de banlieue me comprendront.

Si la synchronisation est déjà passée, alors le cache est valide même sans le réseau. Périodiquement, le cache est rafraîchi, même si l'application ou l'activité n'est pas utilisée. Si l'application Web HTML5 à des choses à communiquer avec le back-end, elle le fera lors de la synchronisation.

Si nécessaire, elle doit s'arranger pour que la page ne soit pas considérée comme entièrement chargée, tant que les données ne sont pas toutes synchronisées.

```
<script type="text/javascript">
if (navigator.onLine)
{
  var xmlhttp=new XMLHttpRequest();
  xmlhttp.open("GET","push.html",true)
  xmlhttp.send()
}
</script>
```

Il reste un dernier problème à gérer : la toute première fois. En effet, si l'utilisateur affiche l'activité avant que la première synchronisation ait été effectuée, il se retrouve avec une page d'erreur.

Une approche simple consiste à demander une gestion du cache avec replis sur le réseau si nécessaire.

```
mWebView.getSettings().setCacheMode(WebSettings.LOAD_CACHE_ELSE_NETWORK);
```

Pour une stratégie cache seul, nous pouvons régler cela assez facilement dans l'activité. Il suffit de capturer les erreurs **ERROR_CONNECT** et **ERROR_HOST_LOOKUP** pour adapter l'affichage de la page courante et déclencher une synchronisation immédiate.

```
mWebView.setWebViewClient(new WebViewClient()
{
  @Override
  public void onReceivedError(WebView view,
    int errorCode, String description, String failingUrl)
  {
    switch (errorCode)
    {
      case ERROR_HOST_LOOKUP:
      case ERROR_CONNECT:
        ContentResolver.requestSync(
          new Account(ACCOUNT_NAME, ACCOUNT_TYPE),
          CONTENT_AUTHORITY, new Bundle());
        ConnectivityManager cm=
         ((ConnectivityManager)getSystemService(Context.CONNECTIVITY_SERVICE));
        NetworkInfo info=cm.getActiveNetworkInfo();
        boolean isConnected =(info!=null) ? info.isConnectedOrConnecting() :
false;
        view.loadUrl(isConnected
          ? "file:///android_asset/Waiting.html"
          : "file:///android_asset/NoNetwork.html");
        break;
      default:
        view.loadUrl("file:///android_asset/Error.html");
    }
  }
});
```

Un truc intéressant est qu'il est possible d'enregistrer un **ContentObserver** pour être prévenu lorsque le cache est mis à jour. Cela permet de rafraîchir immédiatement la page.

```
// Observe the HTTP cache
private final ContentObserver mContentObserver=
  new ContentObserver((new Handler()))
  {
    @Override
    public void onChange(boolean selfChange)
    {
      super.onChange(selfChange);
      onCacheUpdated(null);
    }

    @TargetApi(Build.VERSION_CODES.JELLY_BEAN)
    @Override
    public void onChange(boolean selfChange, Uri uri)
   {
      super.onChange(selfChange, uri);
      onCacheUpdated(uri);
    }
  };
```

Dans la méthode **onCacheUpdated()** de l'activité :

```
protected void onCacheUpdated(Uri uri)
{
  mWebView.reload();
}
```

Il reste à forcer la synchronisation si le cache est vide. Cela arrive lors de la première utilisation de l'application ou si l'utilisateur a cliqué sur le bouton « Vider le cache »

dans les paramètres de l'application. La présence du cache **WebView** est facile à deviner.

```
final boolean webviewCache=
  new File(context.getCacheDir(),"webviewCache").exists() ||
  new File(context.getCacheDir(),"webviewCacheChromium").exists();
```

Notez que pour effacer le cache de l'application lors des tests, je vous conseille cette commande :

```
ad shell "run-as fr.prados.webkitcache rm -R cache"
```

Pour un composant WebKit, le cache est alimenté, même si l'URL est en HTTPS. Et il n'est pas chiffré. Donc, attention. Utilisez **no-cache** si nécessaire sur certaines pages.

Vous pouvez également forcer une synchronisation à partir des paramètres du compte, via le menu.

Pour désactiver la synchronisation en tâche de fond simplement, il suffit de modifier le paramètre **android:enabled** du provider.

```
<service android:name="fr.prados.sync.SyncHTTPService"
  android:exported="false"
  android:enabled="false"
>
...
</service>
```

Faite également attention à la fraîcheur du manifest HTML5. En effet, même avec le paramètre **LOAD_NO_CACHE**, Android commence par charger ce fichier et éventuellement les suivants, si et seulement si, le contenu a été modifié ! Donc, abusez des commentaires pour forcer une modification.

Lors de la création d'un compte, s'il n'en existe pas encore, il est également pertinent de demander une synchronisation immédiate.

```
Bundle b = new Bundle();
// Disable sync backoff and ignore sync preferences. In other
// words...perform sync NOW!
b.putBoolean(ContentResolver.SYNC_EXTRAS_MANUAL, true);
b.putBoolean(ContentResolver.SYNC_EXTRAS_EXPEDITED, true);
ContentResolver.requestSync(
  account, // Sync account
  contentAuthority, // Content authority
  b); // Extras
```

Dans le code proposé, la classe **MainActivity** est un exemple d'utilisation de tout cela. Elle propose une **WebView** synchronisée en tâche de fond. Cette vue est paramétrée pour n'utiliser que les données du cache. Pour tester les différents scénarios, procédez ainsi :

- Désactivez les data 3G/4G,
- Désactivez le Wi-Fi,
- Dans les paramètres, sous l'application, videz le cache,
- et lancez la. Une page indique qu'il faut activer le réseau.
- Nous sommes dans les conditions les pires, pas de réseau, pas de cache.
- Puis activez le Wi-Fi. Après quelques secondes, les logs indiquent que le cache est en marche.
- Dès que les données sont disponibles, une notification le signale à l'activité.
- La **WebView** est rafraîchie automatiquement.
- Dorénavant, même si l'application n'est pas en fonctionnement, le cache est rafraîchi périodiquement.

Attention, suivant les versions d'Android, le composant WebKit peut garder un petit moment la page actuelle en mémoire et ne pas la rafraîchir. Il faut être patient. Après quelques minutes, la page est correctement rafraîchie.

Lors de la compilation en développement, pour faciliter le déverminage, les mises à jours sont effectuées toutes les trente secondes. Il n'est donc pas conseillé de laisser l'application en fonctionnement sur un vrai téléphone. Suivant la valeur de **BuildConfig.DEBUG**, la périodicité du rafraîchissement est modifiée.

5 Petite optimisation

Comme toujours, je ne peux résister à apporter la dernière touche dans le fichier **AndroidManifest.xml**.

Nous pouvons déclarer que le service de synchronisation et le **ContentProvideur** sont dans un processus distinct de l'application. Il suffit d'ajouter dans les deux déclarations l'attribut **android:process=":htmlcache"**. Cela permet d'alléger la mémoire lors d'une synchronisation si l'application n'est pas utilisée par l'utilisateur. Ce processus sera rapidement tué par Android. Attention, il faut alors utiliser le mode **LOAD_CACHE_ELSE_NETWORK**.

Au niveau sécurité, le service et le provider doivent être à **exported="false"**. Ainsi, seul le système Android ou l'application peut les utiliser.

Pour conclure

Nous avons résolu un problème qui me tenait à cœur depuis longtemps. Cette approche permet de clouer le bec à tous les ayatollahs du mode natif. Ils refusent le mode hybride alors que c'est la meilleure solution pour les activités très volatiles.

Avec deux classes et quelques paramètres, le problème est résolu, même en hors-ligne. Je vous accorde que ces classes n'ont pas été facile à concevoir. C'est bien pour cela que je partage mes travaux avec vous dans cet article. ■

LA SOUPLESSE DE PYTHON, LES PERFORMANCES DU C++, LE TOUT SOUS ANDROID SANS TROP SE FATIGUER :

RIEN DE PLUS FACILE AVEC KIVY !

par Guillaume Saupin [Phd]

Vous êtes de ces éternels insatisfaits qui veulent toujours plus ? Vous demandez l'impossible ? Vous voulez développer pour Android, avec la souplesse de python, les performances du C++, en tirant profit du GPU ? Le tout sans trop vous fatiguer ? Vous voulez qu'on vous package rapidement votre application ? Et vous voulez faire tout ça avec un simple shell et vi ? Fallait le dire avant ! Utilisez Kivy !

1 Introduction

Kivy ce n'est pas uniquement une librairie pour développer des interfaces graphiques. Bien sûr, Kivy permet de développer de jolies interfaces exploitant au mieux les possibilités de nos tablettes tactiles. Bien sûr, Kivy est optimisé pour exploiter la puissance de calcul des GPUs qui équipent nos ordinateurs, nos téléphones, ou nos tablettes. Bien sûr, Kivy est écrit en python et permet de bénéficier de toute l'expressivité de ce langage, mais aussi son énorme écosystème. Et bien sûr, Kivy permet de faire tourner vos applications aussi bien sous Linux, que Mac OS, IOS, un système dont le nom m'échappe, et Android !

Mais ce qui est vraiment sympa, avec Kivy, ce sont les outils mis à disposition par l'équipe des développeurs pour faciliter le boulot du développeur Android. Notamment, l'outil python-for-android (à ne pas confondre avec Py4A) est tout simplement génial. Il permet en deux temps trois mouvements de créer un environnement android, et en un coup de cuillère à pot de packager son application Android dans un beau **.apk**.

Et plus fort encore, il propose un mécanisme de recipes qui autorise le déploiement de modules python facilement sur la plateforme android. Mécanisme qui, couplé avec Cython, offre la possibilité de tirer le meilleur parti de Python et du C++ : souplesse et performances !

2 Let's dive into Kivy

Voyons comment tout ça s'articule. Il va vous falloir dans un premier temps installer Kivy. Rien de plus simple.

Inutile de vous le cacher plus longtemps, Kivy utilise python, et ce n'est pas plus mal si vous l'avez installé sur votre système. De même, il n'est pas impossible qu'on soit amené à compiler un peu de C++, du coup, avoir un g++ sous la main peut-être utile. Enfin, pour récupérer la dernière version de Kivy, un outil de gestion de package de la trempe de pip ou des setuptools est le bienvenu.

Je me résume :

```
# sudo apt-get install python g++ python-pip
python-setuptools
```

2.1 Installation

2.1.1 Kivy

Kivy a quelques dépendances, notamment Cython, qui est utilisé pour optimiser une partie du code en générant du C/C++. Vous pouvez installer facilement ces dépendances si elles ne sont pas déjà présentes sur votre système avec les mêmes commandes, soit pour Cython :

```
# sudo easy_install cython
```

Ou bien encore :

```
# sudo pip install cython
```

L'affichage dans Kivy est basé sur openGL. Il vous faut donc l'installer :

```
# sudo apt-get install libgl1-mesa-dev
libgles2-mesa-dev
```

Attelons nous maintenant à la délicate installation de Kivy. A votre convenance, vous pouvez utiliser :

```
# sudo easy_install kivy
```

Ou bien encore :

```
# sudo pip install kivy
```

L'une ou l'autre de ces deux commandes devrait vous permettre de récupérer la version 1.7.2 de Kivy.

A ce stade, vous pouvez déjà créer votre première application Kivy :

```
#copier ce code dans ~/projects/Kivy/premiereAppli/main.py
from kivy.app import App
from kivy.uix.button import Button

class TestApp(App):
    def build(self):
        return Button(text='Hello World')

if __name__=='__main__':
    TestApp().run()
```

Notez que ce code, comme tout ceux présentés dans cet article, est accessible ici : https://github.com/kayhman/kivyLM.

En exécutant :

```
# python main.py
```

vous devriez voir apparaître la fenêtre suivante :

Fig. 1 : Notre première application : un simple hello world

Je ne vous fait pas l'affront de commenter le code.

Et déjà, avec ce simple exemple, un avantage de Kivy apparaît immédiatement : inutile de transférer notre applicatif sur une machine Android. Pas besoin non plus de lancer un émulateur. Tant qu'aucune fonction propre au système Android n'est lancée, un simple appel à l'interpréteur python est suffisant.

C'est particulièrement intéressant pour développer rapidement le squelette de votre application, en se concentrant uniquement sur l'interface.

2.1.2 Python for android

Grâce à Kivy, nous avons désormais un module python nous permettant de créer des interfaces graphiques basées sur openGL, et s'exécutant aussi bien sous Linux, que Windows, Mac OS, Ios, ou encore Android. Tout ce dont Kivy a besoin pour tourner sur la machine cible, c'est d'une installation de python avec les quelques dépendances citées ci-dessus.

Il nous faut donc disposer de ces dépendances sous la plateforme Android pour pouvoir distribuer un applicatif exploitant Kivy. Le projet Python-for-android proposé par l'équipe de Kivy permet justement de créer ces dépendances et de les mettre en forme pour créer simplement un applicatif android basé sur python et Kivy au format apk. Cet ensemble de dépendances est nommé une distribution.

Tournons nous à présent vers l'épineux problème de l'installation de python-for-android. Rendez vous pour cela sur la page https://github.com/Kivy/python-for-android.

Y sont donnés les liens pour télécharger le SDK et le NDK d'android. **wget** s'en sort très bien comme suit :

```
# wget http://dl.google.com/android/ndk/android-ndk-r8c-linux-x86.tar.bz2
# wget http://dl.google.com/android/android-sdk_r21.0.1-linux.tgz
```

Décompresser ensuite ces deux archives à l'endroit de votre choix. Pour ma part, j'ai choisi le répertoire **/opt/android** :

```
# cd /opt/android
# tar xjf /path/to/download/android-ndk-r8c-linux-x86.tar.bz2
# tar xzf /path/to/download/android-sdk_r21.0.1-linux.tgz
```

Vous pouvez alors renseigner votre shell sur l'emplacement de ces SDK et NDK ainsi que sur leur version en ajoutant à la fin de votre **.bashrc** :

```
#~/.bashrc
export ANDROIDSDK="/opt/android/android-sdk-linux_86"
export ANDROIDNDK="/opt/android/android-ndk-r8c"
export ANDROIDNDKVER=r8c
export ANDROIDAPI=14
```

Attention, le développement d'application android est basé sur Java. Il vous faut donc l'installer. L'openjdk fonctionne très bien :

```
# sudo apt-get install openjdk-6-jre
```

N'oubliez pas ensuite de télécharger le SDK android comme suit :

```
# cd /opt/android/android-sdk-linux/tools
# ./android
```

Sélectionner alors le SDK 4.0 correspondant à l'api 14, n'oubliez pas de sélectionner l'option « accept all » et laisser le téléchargement s'effecuter.

Vous êtes alors prêt à récupérer python-for-android :

```
# cd ~
# mkdir -p projects/Kivy
# cd projects/Kivy
# git clone git://github.com/Kivy/python-for-android
# cd python-for-android
```

Reste alors à créer la distribution android avec le module python Kivy :

```
# ./distribute.sh -m "Kivy"
```

L'option **-m** permet d'ajouter à la distribution python for android des modules python. Tous les modules python disponibles en ligne ne sont malheureusement pas installables dans la distribution python créée par python-for-android. La distribution créée se trouve alors dans le répertoire **~/projects/Kivy/python-for-android/dist/default**.

Python-for-android utilise un mécanisme de recipes qui permet l'installation de ces modules dans la distribution Android. Par défaut, une trentaine de modules est déployable pour cette architecture. Nous verrons dans un autre article comment ajouter de nouvelles recipes pour installer les modules qui pourraient vous manquer.

2.2 Kivy sous android

Nous avons désormais sous le coude une distribution de Python nous permettant de faire tourner n'importe quel code Python pur. Les recipes, nous venons de le voir, permettent d'installer d'autres modules, dont notamment Kivy. Nous sommes donc en mesure de faire tourner notre première application **main.py** sur du matériel android.

Pour ce faire, rendez vous dans le dossier :

```
# cd ~/projects/Kivy/python-for-android/dist/default
```

Par défaut, la commande **distribute** installe la distribution dans le sous-répertoire **dist/default**. Vous pouvez altérer ce comportement en passant l'option **-d autreDist**. C'est pratique pour disposer en parallèle de plusieurs distributions avec des modules différents.

Une fois dans ce répertoire, l'application android et son **.apk** se génèrent avec la commande suivante :

```
# ./build.py --package org.MyCompany.KillerApp --name killerApp --version 1.0 --dir ~/projects/Kivy/premiereAppli/ debug
```

Notez que le fichier principal de l'application **doit** s'appeler **main.py**, sans quoi votre application ne démarrera pas, faute de point d'entrée. Le répertoire **~/projects/Kivy/premiereAppli/** doit donc contenir ce fichier.

L'option **debug** construit l'application en mode debug. Il existe un mode **release** qui nécessite un compte sur le Google Play Store.

Le répertoire **~/projects/Kivy/python-for-android/dist/default/bin/** doit désormais contenir un fichier **killerApp-1.0-debug.apk**. Reste à la transférer sur votre matériel android. Trois possibilités :

1. Vous vous adressez le fichier **.apk** par mail ;) et vous installez le fichier.
2. Vous connectez votre matériel à votre ordinateur en tant que périphérique de stockage USB et vous installez le fichier.
3. Vous connectez la machine et utilisez **adb**. C'est l'alternative que je vous conseille, car elle est finalement simple et rapide. De plus, **adb** permet de logger les sorties de votre programme, ce qui est bien utile pour le débogage

La marche à suivre pour se servir d'**adb** suit :

```
# cd ~/opt/android/android-sdk-linux_86/platform-tools/
# sudo ./adb install -r ~/projects/Kivy/python-for-android/dist/default/bin/killerApp-1.0-debug.apk
```

Notez l'utilisation de l'option **-r**, inutile pour l'instant, mais qui force la ré-installation du programme si ce dernier est déjà installé.

Reste à lancer l'application depuis votre tablette ou votre smartphone. Vous devriez voir apparaître la même interface que celle que vous avez vu sur votre ordinateur.

Vous avez désormais un environnement de développement vous permettant d'écrire, de tester et de distribuer facilement une application Python avec une interface graphique sous Android. La porte est ouverte à toutes les fenêtres !

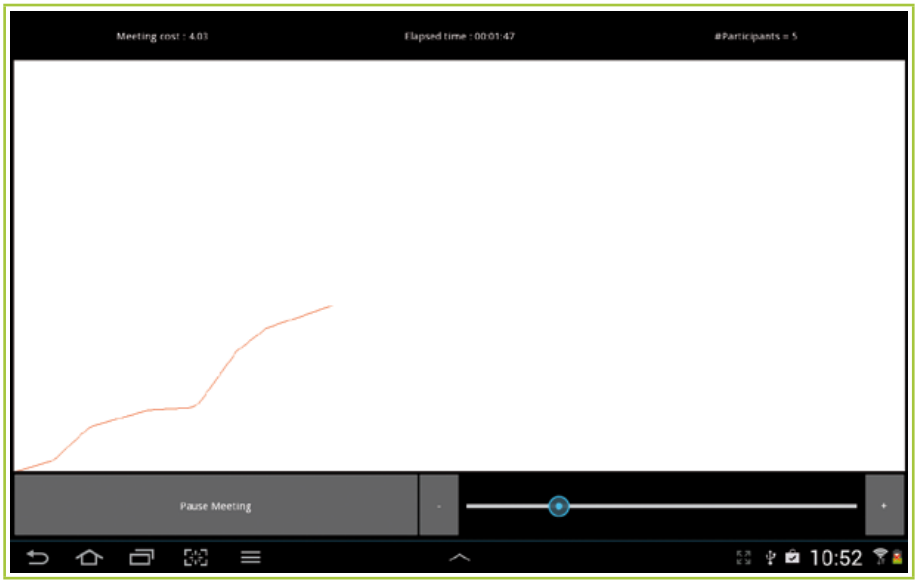


Fig. 2 : La fenêtre principale de notre application exemple MeetingCost

Complétez votre collection d'anciens numéros !

boutique.ed-diamond.com

numerique.ed-diamond.com

Fig. 3 : Le menu de MeetingCost

Pour illustrer cette seconde possibilité, nous allons nous appuyer sur une petite application qui aura pour charge de calculer le coût d'une réunion (voir Figure 2).

Son fonctionnement est simple : on indique un coût horaire moyen pour les personnes présentes à la réunion, on indique le nombre de participants, et on démarre le compteur. L'application indique alors en temps réel, à l'aide d'un graphique, le coût de la réunion (voir Figure 3).

Pour illustrer l'utilisation d'écrans multiples et la capture des « touchs » sur les boutons menu, back, home, ... l'application sera dotée d'un écran menu permettant de basculer entre l'application proprement dite, le menu et un écran de configuration fictif.

3 Pas loin de 50 widgets à disposition

La libraire Kivy a été conçue spécialement pour développer des interfaces graphiques pour les appareils dotés d'une interface tactile.

Pour découvrir l'éventail des possibilités offertes par cette sympathique librairie, je vous engage à télécharger sur le Google Play Store l'application Kivy Showcase.

Comme pour toute libraire de création d'interface graphique, la création d'une application fonctionnelle en Kivy suppose deux étapes :

- La création de l'interface à proprement parler : le positionnement des différents widgets, la définition de leur taille, de leur contenu, ...
- Le câblage de l'interface aux actions qu'elle est censée gérer: il s'agit de connecter les différents événements reconnus par Kivy (**on_touch**, **on_press**, ...) à des fonctions.

Nous allons détailler ces deux étapes ci-dessous.

3.1 Créer une interface graphique

Kivy offre deux possibilités pour créer votre interface. Soit l'interface est complètement décrite en python, en instanciant à la mano les widgets. Cette méthode est assez fastidieuse.

Soit l'interface est décrite à l'aide d'un petit langage mis au point par l'équipe Kivy : le kv.

3.1.1 Description en kv de l'interface

Nous allons utiliser le langage offert par Kivy, le kv pour décrire simplement l'architecture de votre interface. Le chargement de ce fichier se fait soit à la mano, à l'aide d'une commande du type :

```
Builder.load_file('path/to/file.kv')
```

Soit automatiquement, suivant la convention suivante : une classe nommée **MonAppliApp** charge le fichier **monappli.kv**.

La syntaxe est très simple. Un nouveau widget est créé avec un tag du type **<MonNouveauWidget>**. Sous ce tag, de manière arborescente, la description du nouveau widget est donnée. Cette description fixe les dimensions ainsi que la position du widget mais précise aussi les widgets de base dont il est constitué.

Notez que, comme en python, l'indentation est importante et identifie des sous-blocs.

Pour notre application **MeetingCosts**, voilà ce que cela donne (Reportez vous à la figure 1 et 2 pour un aperçu graphique) :

```
#copier ce code dans ~/projects/Kivy/meetingCostsKv/meetingcosts.py
#:kivy 1.0

<Graph>:
    size_hint: 1.0,1.0
    canvas:
        Rectangle:
            pos: self.pos
            size: self.size


<MenuScreen>:
```

```
    BoxLayout:
        Button:
            text: 'Cost'
            on_press: root.manager.current = 'cost'
        Button:
            text: 'Settings'
            on_press: root.manager.current = 'settings'
        Button:
            text: 'Quit'
            on_press: app.stop()

<SettingsScreen>:
    Label:
        text: 'Settings : not yet implemented'

<MeetingCost>:
    costLabel: costLabel
    elapsedLabel: elapsedLabel
    slider: slider
    participantsLabel: participantsLabel
    graph: graph
    startButton: startButton

    BoxLayout:
        orientation: 'vertical'
        padding: 5
        spacing: 5


        BoxLayout:
            size_hint_y: 0.1
            orientation: 'horizontal'
            Label:
                id: costLabel
                text: ''
            Label:
                id: elapsedLabel
                text: ''
            Label:
                id: participantsLabel
                text: ''
        Graph:
            id: graph

        BoxLayout:
            size_hint_y: 0.15
            orientation: 'horizontal'

            Button:
                id: startButton
                text: 'Start Meeting'
                on_press: root.start_stop_pressed()
                size_hint_x: 0.1

            Button:
                text: '-'
                on_press: slider.value = slider.value-1
                size_hint_x: 0.01
            Slider:
                id: slider
                min: 0
                max: 25
                size_hint_x: 0.1

            Button:
                text: '+'
                on_press: slider.value = slider.value+1
                size_hint_x: 0.01
```

Ce fichier décrit donc un premier widget **Graph** qui est composé d'un canvas, dans lequel on dessine un simple rectangle. Ce widget sera utilisé pour afficher la courbe d'évolution du coût de la réunion.

Les widgets **<SettingsScreen>** et **<MenuScreen>** sont deux widgets très simples utilisés respectivement comme écran de réglage et comme menu pour basculer entre les différents écrans de l'application.

Le widget **<MeetingCosts>** est le principal de l'application. Il réutilise le widget **<Graph>** créé en amont. Il décrit l'interface de l'application à l'aide de **BoxLayout** verticales et horizontales qui permettent d'empiler facilement les widgets.

Ainsi, une **BoxLayout** horizontale contient les trois **Labels** chargés d'afficher le coût, le temps écoulé et le nombre de participants. Chaque **Labels** a, comme tout widget, un champs **id**, qui est très important, car il est utilisé pour accèder au widget instancié. Ces champs **id** se retrouvent d'ailleurs tout de suite après la déclaration du widget **<MeetingCost>** :

```
<MeetingCost>
   costLabel: costLabel
   elapsedLabel: elapsedLabel
   slider: slider
   participantsLabel: participantsLabel
   graph: graph
   startButton: startButton
```

Une autre **BoxLayout** horizontale contient pour sa part le bouton pour démarrer le calcul du coût et un slider pour augmenter le nombre de participants. Notez la présence du champs **on_press** qui permet d'asssocier facilement la fonction **root.startStopPressed()** à un click sur ce dernier. L'objet **root** est créé automatiquement et permet d'accéder directement au widget racine du widget créé. De même, l'objet **app** pointe systématiquement vers l'application courante. Ainsi, le bouton Quit du **<menuScreen>** stoppe l'application par un appel à **app.stop()**.

Ces deux **BoxLayout** horizontales sont à leurs tours placées dans une **BoxLayout** verticale, au dessus et en dessous du widget **<Graph>** qui affiche l'évolution du coût.

3.2 Ajouter la logique à notre applicatif

A l'issu de la section précédente, nous avons décrit l'interface de notre application. Il reste à lui ajouter la logique nécessaire à son bon fonctionnement. C'est la responsabilité du fichier **main.py** :

```
import Kivy
Kivy.require('1.1.1')

from Kivy.lang import Builder
from Kivy.uix.button import Button
from Kivy.uix.widget import Widget
from Kivy.uix.screenmanager import ScreenManager, Screen
from Kivy.uix.label import Label
from Kivy.app import App
from Kivy.clock import Clock
from Kivy.properties import ObjectProperty, StringProperty
from Kivy.graphics import Color, Line, Rectangle

from datetime import datetime, timedelta
import time
```

Cette première section de code importe les modules définissant les différentes classes dont nous avons besoin.

```
class MenuScreen(Screen):
    pass
class SettingsScreen(Screen):
    pass
```

Nous déclarons ici les classes **MenuScreen** et **SettingsScreen** correspondant aux widgets **MenuScreen** et **SettingsScreen** que nous avions définis dans le fichier **.kv**. Ces deux classes héritent de la classe **Screen** afin de pouvoir être utilisées dans un **ScreenManager**, pour basculer de l'un à l'autre.

Ces widgets n'ont pas de comportement particulier, donc leur classe ne contiennent rien.

```
class Graph(Widget):
    def __init__(self, **kwargs):
        super(Graph, self).__init__(**kwargs)
        self.secondsElapsed = 0.0
        self.totalCost = 0.
        self.hourCost = 15.
        self.nbParticipants = 0
        self.maxY = 10 #dollars
        self.maxX = 5 * 60 #seconds
        self.points = []
    def update_cost(self, dt):
        self.totalCost = self.totalCost + dt / 3600.0 * self.hourCost
* self.nbParticipants
        self.secondsElapsed = self.secondsElapsed + dt
        self.points.append( (self.secondsElapsed, self.totalCost) )
        if self.totalCost > self.maxY or self.secondsElapsed > self.
maxX:
            self.rescale()
        data = reduce(lambda r, x : r + (self.pos[0] + x[0] * self.
size[0] / self.maxX, self.pos[1] + x[1] * self.size[1] / self.maxY),
self.points, ())
        with self.canvas:
            Color(1.0, 0., 0.)
            Line(points=data)

    def on_resize(self, width, height):
        self.canvas.clear()

    def rescale(self):
        self.maxY = 2.0 * self.maxY
        self.maxX = 2.0 * self.maxX
        self.canvas.clear()
        with self.canvas:
            Rectangle(pos=self.pos, size=self.size)
```

Le widget **Graph**, par contre, a un comportement bien à lui. Il est notamment responsable de l'affichage de la courbe d'évolution du coût de la réunion. C'est l'objet de la fonction **update_cost(self, dt)** qui se charge de mettre à jour le widget. Elle utilise pour ça son canvas, et trace une ligne passant par tous les points (temps, coût) à l'aide de la fonction **Line**.

Dans le cas où la courbe sortirait de l'écran, par exemple lors d'une réunion qui durerait un peu longtemps, la méthode **rescale(self)** est appelée.

```
class MeetingCost(Screen):
    costLabel = ObjectProperty()
    elapsedLabel = ObjectProperty()
    slider = ObjectProperty()
    participantsLabel = ObjectProperty()
    graph = ObjectProperty()
    startButton = ObjectProperty()
    info = StringProperty()

    def __init__(self, **kwargs):
        super(MeetingCost, self).__init__(**kwargs)
        self.stopped = True
        self.elapsedTime = 0
```

Ces quelques lignes ci-dessus sont essentielles. Elles permettent à la classe **MeetingCosts**, qui est associée automatiquement au widget **<MeetingCost>** décrit dans le fichier **.kv**, d'accéder à ses sous-widgets. Cet accès se fait simplement en utilisant la valeur des champs id spécifiés dans le **.kv**.

```
    def build(self):
        self.participantsLabel.text = "#Participants = %d"%0
        self.costLabel.text = "Meeting cost : %.2f"%0.
        self.elapsedLabel.text = "Elapsed time :
%.2d:%.2d:%.2d"%(0,0,0)
        self.slider.bind(value=self.update_participants)
```

Le widget **MeetingCost**, le cœur de notre application, est doté de 9 fonctions. La fonction **build()**, appelée à la création du widget, fixe le texte des labels et associe la fonction **update_participants()** au slider.

À DÉCOUVRIR ACTUELLEMENT !

Toutes les bonnes pratiques pour

FAIRE CONNAÎTRE ET EXPLOITER AU MIEUX... VOTRE PROJET OPEN SOURCE !

GNU/LINUX MAGAZINE HORS-SÉRIE N°69

DISPONIBLE CHEZ VOTRE MARCHAND DE JOURNAUX ET EN LIGNE SUR : boutique.ed-diamond.com

```
    def start_stop_pressed(self)  :
        if self.stopped:
            self.start_meeting()
        else:
            self.stop_meeting()

    def start_meeting(self):
        if self.stopped:
            self.startButton.text = "Pause Meeting"
            Clock.schedule_interval(self.update, 1.0)
            self.stopped = False

    def stop_meeting(self):
        if not self.stopped:
            self.startButton.text = "Start Meeting"
            Clock.unschedule(self.update)
            self.stopped = True
```

Les méthodes **start_meeting()**, **stop_meeting()** et **start_stop_pressed()** se chargent de démarrer ou stopper le calcul du coût de la réunion. Pour ce faire, la méthode **start_meeting(self)** utilise une classe bien pratique fournie par Kivy : **Clock**. Cette classe permet d'appeler de manière régulière une méthode. C'est exactement ce que nous voulons faire pour calculer le coût de la réunion : multiplier régulièrement le temps écoulé par le nombre de participants et leur coût horaire moyen !

La création de cet appel régulier se fait avec la ligne **Clock.schedule_interval(self.update, 1.0)** qui assure que **self.udpate** sera appelée toutes les secondes.

La suppression de cet appel se fait aussi simplement avec **Clock.unschedule(self.update)**.

```
    def on_resize(self, width, height):
        self.canvas.clear()

    def update_participants(self, instance, value):
        self.graph.nbParticipants = value
        self.participantsLabel.text = "#Participants = %d"%value

    def update_cost(self, dt):
        self.costLabel.text = "Meeting cost : %.2f"%self.graph.totalCost

    def update_elasped(self, dt):
        self.elapsedTime = self.elapsedTime + dt
        hours = self.elapsedTime / 3600
        minutes = (self.elapsedTime / 60 ) % 60
        secondes = self.elapsedTime % 60
        self.elapsedLabel.text = "Elapsed time : %.2d:%.2d:%.2d"%(hours,minutes
,secondes)

    def update(self, dt):
        self.graph.update_cost(dt)
        self.update_cost(dt)
        self.update_elasped(dt)
```

Ces dernières méthodes sont du python pur et se contentent de mettre à jour les divers widgets tout au long de la réunion.

```
class MeetingCostApp(App):
    def __init__(self, **kwargs):
        super(MeetingCostApp, self).__init__(**kwargs)
        self.cost = None
        self.menu = None
        self.meetingWasStopped = None
        self.sm = None
```

Pour assurer le chargement automatique du fichier **meetingcost.kv**, la classe principale, dérivant de **App**, a été nommée **MeetingCostApp**.

```
    def build(self):
        self.sm = ScreenManager()
        self.cost = MeetingCost(name="cost")
        self.menu = MenuScreen(name="menu")
        self.settings = SettingsScreen(name="settings")
        self.cost.build()
        self.sm.add_widget(self.cost)
        self.sm.add_widget(self.menu)
        self.sm.add_widget(self.settings)
        self.sm.curent = "cost"
        self.bind(on_start = self.post_build_init)
        return self.sm
```

La méthode **build()**, appelée à la construction de la classe, crée les principaux widgets et les initialise. Elle définit aussi l'onglet de notre **ScreenManager** activé par défaut, soit « **cost** ».

```
    def on_pause(self):
        self.meetingWasStopped = self.cost.stopped
        self.cost.stop_meeting()
        return True

    def on_stop(self):
        self.meetingWasStopped = self.cost.stopped
        self.cost.stop_meeting()
        return True

    def on_resume(self):
        if not self.meetingWasStopped:
            self.cost.start_meeting()
```

Sous Android, toute application qui n'est pas affichée au premier plan est par défaut arrêtée. Il faut donc gérer le cas où l'utilisateur quitte notre application pour passer par exemple à son outil de prise de note. Il faut alors sauvegarder le coût et le temps courant pour que lorsque l'utilisateur re-bascule sur notre application, le coût soit remis à jour en fonction du temps écoulé.

Dans cette optique, Kivy propose par défaut les méthodes **on_pause**, **on_stop**, **on_resume**.

```
    def _key_handler(self, key, scanCode, codePoint, modifier, args):
        if codePoint == 4:
            if self.sm.current != "menu":
                self.sm.current = "menu"
            else:
                App.get_running_app().stop()

    def post_build_init(self,ev):
        import android
        import pygame

        android.map_key(android.KEYCODE_BACK, 1001)
        win = self._app_window
        win.bind(on_keyboard=self._key_handler)

if __name__=='__main__':
    MeetingCostApp().run()
```

Enfin vient la méthode **post_build_init()**, qui est activée au lancement de l'application, grâce à la ligne :

```
        self.bind(on_start = self.post_build_init)
```

Elle nous permet de capturer la touche Retour de votre tablette / téléphone android à l'aide de **map_key**, en mappant le keycode android vers un keysym python. Cela permet de transformer un événement tactile en un événement clavier.

Ensuite, la méthode **bind** lie la méthode **_key_handler** avec les événements clavier. Cette méthode teste alors simplement si l'événement correspond à click sur la « touche » retour. Si c'est le cas, elle bascule vers le menu, si ce dernier n'est pas déjà affiché. S'il est déjà activé, l'application est alors fermée.

3.3 Création de l'application et transfert sur le matériel Android

La création de l'application se fait très simplement, comme pour le Hello World, en se rendant dans le répertoire contenant la distribution de python pour android et en construisant l'application avec la méthode build :

```
# cd ~/projects/Kivy/python-for-android/dist/default/
# ./build.py --package org.MyCompany.Meetingcost --name MeetingCost
--version 1.0 --dir ~/projects/Kivy/meetingCostsKv/ debug
```

Le transfert de l'application sur votre support android, se fait depuis le répertoire d'installation du SDK android :

```
# cd ~/opt/android/android-sdk-linux_86/platform-tools/
# sudo ./adb install -r ~/projects/Kivy/python-for-android/dist/
default/bin/MeetingCost-1.0-debug.apk
```

4 Allez plus loin

L'ensemble du code est disponible sur github : https://github.com/kayhman/KivyLM. Vous avez avec ce code un squelette d'application qui vous permet de bénéficier de la puissance de python pour mettre au point rapidement une application Android et publier rapidement sur le market android la nouvelle killer app.

Le framework Kivy présente un environnement de développement très complet pour android, mais aussi IOS, Linux, windows et MacOsX. En plus de cet aspect multiplateforme et de la possibilité qu'il offre d'utiliser python pour développer sous android, Kivy offre un autre avantage que nous n'avons pas exploité ici : il peut bénéficier facilement de Cython à l'aide de son mécanisme de recipes. Cette exploitation de Cython permet de réécrire facilement une partie du code en C/C++ et de booster drastiquement les performances de votre application.

Cet aspect de Kivy fera l'objet d'un autre article où nous verrons tout l'intérêt du couple Kivy/Cython. ■

SALT, L'AUTRE CHEF D'ORCHESTRE

par Emile <iMil> Heitor [pour GCU canal historique et la secte des serviettes de bain oranges]

« Devops », « Cloud », IaaS et j'en PaaS (oh-oh-oh la bonne blague d'entrée) et des meilleures, vous, lecteurs avides de technologies innovantes, vous n'avez pas échappé à la déferlante de buzzwords dont nous sommes victimes depuis que l'Internet est devenu une marketplace. Et finalement, si beaucoup manipulent ces buzzwords sans avoir aucune idée de ce qu'ils cachent, ils représentent une réalité tout à fait tangible qui met des parts de pizza sur la table de bon nombre d'entre nous. Véritables icônes du mouvement devops, les systèmes d'orchestration ont révolutionné la façon dont on manipule une ferme de machines; l'avènement de la virtualisation et de la transformation des serveurs physiques en « ressources » a fait naître de nouveaux besoins. Puppet et Chef font figure de pionniers dans cette sphère hyperactive mais un concurrent dont on entend beaucoup moins parler n'a pas à rougir devant ses prédécesseurs, il s'agit de Salt[1].

1 Loin de moi, bloat

Salt se distingue de ses homologues car son auteur a eu la bonne idée de ne pas construire un monstre s'appuyant sur des solutions de messagerie à la complexité douteuse mais a plutôt misé sur la librairie *ZeroMQ* **[2]**, s'assurant ainsi vélocité et simplicité. Ce choix est loin d'être anodin car, sans aucun effort supplémentaire, *Salt* offre d'emblée des fonctionnalités de gestion de configuration **et** d'orchestration. Et nous allons voir que mettre le pied à l'étrier du cheval *Salt* n'a rien de l'insurmontable. Enfin, mais c'est évidemment très personnel, je n'ai aucune affinité avec le langage *Ruby* et au contraire des solutions sus-citées, *Salt* est écrit en python, plus doux à mes yeux. Cela peut paraître dérisoire à première vue, mais devant la forte probabilité de tentation d'écriture de modules, le langage de la solution **a** son importance.

Bien que jeune (Mai 2011), le projet a d'ores et déjà le soutien de noms prestigieux tels que HP, LinkedIn ou encore PayPal, se décline déjà en version commerciale et sait adresser la quasi totalité des *Clouds* publics et privés.

Enfin, la liste des services Libres que *Salt* est en mesure de gérer est impressionnante et ne cesse de grossir, tant le développement de modules est simple avec un strict minimum de connaissances en *python* **[4]**.

2 Musique !

Disponible pour nombre de systèmes de paquets, vous n'aurez aucun mal à installer *Salt* au sein de votre système d'exploitation. Cet article se concentrera cependant sur deux plateformes, Debian GNU/Linux et NetBSD.

Le site *SaltStack* propose un dépôt de paquets binaires **[3]** à jour qu'on déclare comme d'habitude sur un système Debian :

```
# cat > /etc/apt/sources.list.d/salt.list << EOF
deb http://debian.saltstack.com/debian wheezy-saltstack main
EOF
# wget -q -O- "http://debian.saltstack.com/debian-salt-team-joehealy.
gpg.key" | apt-key add -
# apt-get update
```

Deux démons sont mis à votre disposition :

- salt-master, le serveur central où se connecteront tous les esclaves, *minions* dans la terminologie *Salt* (j'adore.)
- salt-minion, donc, des dizaines, centaines, milliers de serveurs potentiellement interrogés et contrôlés par le *master*.

Une fois n'est pas coutume, notre machine Debian sera un *minion*, aussi, nous installons le nécessaire :

```
# apt-get install salt-minion
```

Nous reviendrons à la configuration du *minion* un peu plus loin.

Vous l'aurez compris, notre *master* sera un système *NetBSD* qui dispose dans **pkgsrc/sysutils/salt** d'un paquet dont je m'assure personnellement qu'il soit à jour. Ainsi, il suffira d'invoquer **pkgin** de la sorte :

```
# pkgin in salt
```

Afin de disposer du nécessaire à la poursuite des opérations.

Comme d'habitude sous *NetBSD*, nous activons les services souhaités de la façon suivante :

```
# cat >> /etc/rc.conf << EOF
salt_master=YES
salt_minion=YES
EOF
```

Comme vous pouvez le constater, notre machine maître sera également *minion* car nous voulons pouvoir la piloter comme le reste de notre parc.

Afin d'indiquer au *minion* qui est son maître, nous éditons le fichier **/usr/pkg/etc/salt/minion** et plaçons les valeurs suivantes :

```
# hostname ou fqdn du master
master: coruscant
# identifiant du minion, par simplicité, le hostname de la machine
id: coruscant
```

Dans l'immédiat, aucune autre configuration n'est nécessaire, nous démarrons **salt-master** et **salt-minion** sans plus de manières :

```
# /etc/rc.d/salt_master start
# /etc/rc.d/salt_minion start
```

Nous configurerons notre *minion Debian* de la même façon, seul le chemin vers le fichier de configuration diffère et il faudra évidemment donner à ce dernier un **id** different :

```
# hostname ou fqdn du master
master: coruscant
# identifiant du minion
id: tatooine
```

On démarre le *minion* sur cette machine de façon classique :

```
# /etc/init.d/salt-minion start
```

Afin de contrôler notre premier minion, nous allons l'autoriser à se connecter au maître. Pour ce faire, la commande **salt-key**, lancée sur le *master*, va autoriser l'échange de clés de type AES entre notre *minion* et son maître. On liste en premier lieu les minions en attente d'acceptation :

```
$ sudo salt-key -L
Accepted Keys:
ragnos
tatooine
watto
Unaccepted Keys:
coruscant
Rejected Keys:
```

Et on accepte simplement l'échange de clés :

```
$ sudo salt-key -a coruscant
```

Ou pour accepter toutes les clés en attente :

```
$ sudo salt-key -A
```

À cet instant, la commande **salt-key -L** devrait renvoyer la liste complète des *minions* autorisés :

```
$ sudo salt-key -L
Accepted Keys:
coruscant
ragnos
tatooine
watto
Unaccepted Keys:
Rejected Keys:
```

Dès lors, il nous est immédiatement possible d'envoyer des commandes à nos *minions*. Vérifions en premier lieu ce que ces derniers nous répondent :

```
$ sudo salt '*' test.ping
coruscant:
    True
tatooine
    True
ragnos:
    True
watto:
    True
```

On pourra évidemment tester un seul *minion* en précisant son *id* :

```
$ sudo salt 'tatooine' test.ping
tatooine
    True
```

Et maintenant que nous savons que nos *minions* répondent présent, nous pouvons de ce pas leur faire exécuter des commandes :

```
$ sudo salt '*' cmd.run "uname -a"
tatooine:
    Linux tatooine 3.2.0-4-686-pae #1 SMP Debian 3.2.41-2 i686 GNU/Linux
coruscant:
    NetBSD coruscant 6.0 NetBSD 6.0 (XEN3_DOM0) amd64
watto:
    NetBSD watto.home.imil.net 6.1_RC4 NetBSD 6.1_RC4 (GENERIC) i386
ragnos:
    NetBSD ragnos 6.0 NetBSD 6.0 (RAGNOS) #2: Wed Oct 17 11:33:31 CEST
2012  root@ragnos:/usr/src/sys/arch/i386/compile/RAGNOS i386
```

Il ne s'agit là que des prémices de ce que permet l'orchestration via *Salt*, car les modules sont très nombreux. La liste complète des actions sont visibles grâce à la commande :

```
$ sudo salt '*' sys.doc
```

Testons par exemple la commande *pkg.install* sur notre minion *Debian* :

```
$ sudo salt 'tatooine' pkg.install vim
tatooine:
    ----------
    vim:
        ----------
        new:
            2:7.3.547-7
        old:
```

Bien évidemment ici, c'est **apt** qui est appelé derrière la scène, mais l'action d'installation d'un paquet via *Salt* aurait pris la même forme vers un *minion FreeBSD* ou *RedHat*.

À cela il faut ajouter une fonctionnalité donnant accès à plus de granularité dans l'interrogation des *minions*: les *grains*, justement. Cette interface permet en particulier d'interroger les *minions* fonction d'une caractéristique particulière. Par exemple, si l'on voulait *pinger* uniquement les machines dont le système d'exploitation est *NetBSD*, nous ferions :

```
$ sudo salt -G 'os:NetBSD' test.ping
watto:
    True
coruscant:
    True
ragnos:
    True
```

La liste des *grains* disponibles s'obtient via la commande :

```
$ sudo salt '*' grains.ls
```

Et leurs valeurs par l'invocation suivante :

```
$ salt '*' grains.items
```

Les *grains* sont un catalogue de données statiques, glanées au démarrage du *minion* et ces dernières s'étendent de façon très simple dans le fichier de configuration **/usr/pkg/etc/salt/minion** sous *NetBSD* et **/etc/salt/minion** sur un environnement GNU/Linux. Par exemple :

```
grains:
  roles:
    - www
    - mysql
  datacenter: equinix
  baie: A17
  chassis: 3
```

Nous reparlerons des *grains* un peu plus loin dans cet article.

3 Mes précieuuuuux fichiers...

Les suites logicielles telles que *Salt* adressent de façon élégante une problématique plus vaste que l'exécution de taches distantes : la gestion de la configuration.

Dressons le tableau, vous disposez d'un nombre conséquent de machines (possiblement virtuelles) pour lesquelles vous souhaiteriez :

- qu'elles soient toutes à jour,
- qu'elles soient manipulées comme un tout cohérent et non pas de façon artisanale,
- que la configuration de vos outils fétiches (vim, tmux, apt, bash, zsh, OpenSSH...) soit identique,
- que l'ajout, la modification ou la suppression d'un fichier soit répercutée globalement.

Ce que vous appelez de vos vœux est ce qu'on appelle la gestion de la configuration (ou gestion du changement en langage corpo-flanby). Cette fonctionnalité est inclue de base dans *Salt* et s'organise à travers un répertoire où l'on trouvera les fichiers dictant les diverses actions à mener. Pour diffuser ces informations vers les *minions*, le maître dispose d'un serveur de fichiers articulé autour de *zeromq* afin d'assurer une rapidité exemplaire du transport. Elle s'active très simplement dans le fichier **/usr/pkg/etc/salt/master** (ou **/etc/salt/master** sur un *master* GNU/Linux) :

```
file_roots:
  base:
    - /home/imil/salt
```

Nous définissons ici que le serveur de fichiers trouvera sa racine dans le repertoire **/home/imil/salt**. Les connaisseurs n'auront pas manqué de remarquer que la syntaxe présentée, comme celle de la déclaration des *grains*, est au format YAML **[5]**, interpreté par *PyYAML* qu'utilise par défaut *Salt*.

Dans ce répertoire racine du serveur de fichiers, on organisera les fichiers *states*, des recettes dans le vocable *Puppet / Chef* qui définiront les actions à mener sur nos *minions* mais aussi possiblement les fichiers à transférer. Nous verrons plus loin que d'autres méthodes sont également disponibles pour additionner à la gestion de la configuration une dimension de *versionning*.

Commençons par un exemple très simple: nous souhaitons déposer sur tous nos *minions* le fichier **foo**. Écrivons pour cela un *state* simpliste :

```
$ cat test.sls
/tmp/foo:
  file:
    - managed
    - source: salt://foo
```

Sans plus de fioritures, et comme on peut s'y attendre, ce *state* copiera le fichier **foo** présent dans **home/imil/salt** vers le répertoire **/tmp** de nos *minions*. Afin de prendre en compte ce fichier *sls* il est nécessaire d'écrire un fichier de haut niveau appelé **top.sls**. Dans ce fichier, on associe des `environnements' à des minions, puis on déclare les *states* associés. Nous avons défini plus haut dans la configuration du maître l'environnement *base*, nous nous cantonnerons à cet environnement dans le cadre de cet article.

```
$ cat top.sls
base:
  '*':
    - test
```

En clair, dans l'environnement *base* dont la racine se trouve dans **/home/imil/salt**, pour l'ensemble des *minions* (***** est un *glob* ici), nous chargerons le *state* **test.sls**. Créons donc notre fichier **foo** de la façon la plus simple qui soit :

```
$ pwd
/home/imil/salt
$ echo bar > foo
```

Et demandons à *Salt* ce qu'il compte faire, on l'interroge grâce à la commande :

```
$ sudo salt 'watto' state.show_highstate
watto:
    /tmp/foo:
        ----------
        __env__:
            base
        __sls__:
            test
        file:
            - managed
            ----------
            - source:
                salt://foo
```

On remarquera que nous avons précisé l'*id* minion, **watto**, sur lequel nous souhaitons publier puis exécuter le *state* et, comme prévu, *Salt* nous informe de l'opération à mener. Exécution :

```
$ sudo salt 'watto' state.highstate
----------
    State: - file
    Name:      /tmp/foo
    Function:  managed
        Result:    True
        Comment:   File /tmp/foo updated
        Changes:   diff: New file
```

Et de vérifier sur le *minion* :

```
(imil@watto):[~]
$ cat /tmp/foo
bar
```

Au delà des fonctions de type **file**, un nombre impressionnant de modules est disponible, leur étude exhaustive remplirait la totalité des pages de ce magazine, aussi, nous ne nous concentrerons que sur certaines d'entre elles. En particulier, l'une des utilisations les plus courantes de ce type de suite logicielle concerne l'installation et la mise à jour des paquets, cette opération est prise en charge par les fonctions de type **pkg**. Il s'agit bien évidemment du pendant des opérations réalisées plus haut en mode commande, mais cette fois au sein d'une recette ; là aussi, ces fonctions sont abstraites et appelleront en sous-main les divers outils de *packaging* de l'*OS*.

Exemple très simple: nous souhaitons nous assurer que le paquet **vim** est bien installé sur tous nos *minions*, nous renseignons un fichier *state* avec les lignes suivantes :

```
$ cat test.sls
vim:
  pkg:
    - installed
```

Comme on peut l'imaginer, l'application de ce *state* sur les *minions* aura pour effet d'installer le paquet nommé **vim** sur ces derniers.

Plus fort, nous souhaitons mettre à jour un fichier sur les systèmes distants en nous assurant que le logiciel qui l'utilisera est bien présent :

```
$ cat test.sls
/etc/vimrc:
  file:
    - managed
    - source://vim/vimrc
    - require:
      - pkg: vim
```

Ainsi, le fichier présent sur le serveur de fichiers *Salt*, **/home/imil/salt/vim/vimrc** sera copié sur les minions sous **/etc/vimrc** mais non sans avoir installé le paquet **vim** si ce dernier était absent sur la machine destination.

4 Les colonnes de sel

Nous venons à peine de gratter la surface de capacités ce de fantastique outil, car pour industrialiser plus en amont les déploiements, il est pratiquement toujours impératif de rendre plus génériques les actions. Pour ce faire, *Salt* met à disposition de l'administrateur des concepts de *templating* à mi-chemin entre la déclaration et la programmation.

Dans cette optique, *Salt* introduit la notion de *pillar*. Un fichier *pillar* définit des données relatives à un ou des *minions*. Par exemple, on pourrait y stocker une liste de fichiers, d'utilisateurs, de donnés à post-traiter...

Les *pillars* sont actifs par défaut dans *Salt* mais pour les besoins de l'expérimentation, nous allons redéfinir leur emplacement sur le système de fichiers dans la configuration du *master* :

```
pillar_roots:
  base:
    - /home/imil/salt/pillar
```

Pour tirer profit des méthodes de *templating* mises à notre disposition, nous allons dérouler le scénario suivant : nous disposons d'une poignée de *dotfiles*, fichiers de configuration de nos outils fétiches, que nous souhaiterions déployer sur l'ensemble de nos *minions*. Simple mais fastidieuse, la spécification des actions à mener pour l'intégralité

des fichiers semble intuitivement une mauvaise idée. De plus, nous allons pousser l'élégance jusqu'à l'utilisation d'un dépot *git* comme source d'information. Oui, *Salt* sait faire ça.

De façon à permettre à *Salt* de s'interfacer sur un serveur *git*, les paramètres suivants devront être activés dans la configuration du *master* :

```
fileserver_backend:
  - roots
  - git

gitfs_remotes:
  - git://github.com/iMilnb/dotfiles.git
```

Deux remarques :

- « *Order matters* », l'ordre d'apparition des *backends* a son importance, dans l'exemple ci-dessus, si un fichier est trouvé dans la racine du serveur de fichiers local, *Salt* n'ira pas chercher dans son cache de fichiers *git*.
- Il est possible de lister plusieurs dépôts *git* et, comme précédemment, l'ordre est pris en compte.

Notez que pour l'utilisation de cette fonctionnalité, l'ajout du module **python-git** ou **python-git-python** est obligatoire. D'autre part, *Salt* n'interrogera pas systématiquement votre dépôt *git* lors d'un déploiement, mais utilisera plutôt un cache local accessible également à travers le *scheme* **salt://**, comme l'est un classique fichier local au système de fichiers.

Concentrons nous maintenant sur l'écriture de notre fichier *pillar*; les *pillars* ont le même format et la même extension que les *states*, nous créons donc en premier lieu un fichier **top.sls** dans le répertoire **pillar** et un fichier destiné à fournir les données que nous utiliserons lors du déploiement de nos fichiers *dotfiles* :

```
$ cat pillar/top.sls
base:
  '*':
    - dotfiles

$ cat pillar/dotfiles.sls
dotfiles:
  .tmux.conf:
  {% if grains['kernel'] == 'Linux' %}
    .tmux.conf-linux
  {% else %}
    .tmux.conf
  .profile:
    .profile
  .bashrc:
    .bashrc
  {% endif %}
  .vimrc:
    .vimrc
  .vim/colors/molokai-trans.vim:
    .vim_colors_molokai-trans.vim
  .vim/colors/molokai.vim:
    .vim_colors_molokai.vim
```

Bon oui, là comme ça, ça pique un peu. Mais lisons plus attentivement, nous déclarons ici une simple structure de données (*dotfiles*) qui contient un nom de fichier local, celui qui sera déployé sur nos *minions* et le nom du fichier distant (celui présent sur le dépôt *git*). Comme vous pouvez le constater, nous utilisons la notion de *grains*, en effet, la version de *tmux*, sur mes serveurs *Debian* est antérieure à celle proposée par **pkgsrc** sur mes machines *NetBSD* et ne supporte pas certaines options, aussi, je dispose de deux fichiers distincts. De la même façon, je ne souhaite pas écraser les fichiers **.profile** ni les **.bashrc** des machines Linux.

Dans ce fichier *pillar*, on touche une notion extrêmement puissante de la syntaxe des fichiers *sls*, il est en effet possible, grâce au système de *template Jinja* **[6]** de **programmer** nos templates en utilisant les facilités fournies par *Salt*. Ici, nous utilisons les informations des *grains* pour déterminer les fichiers déclarés en fonction du système d'exploitation du *minion*.

Munis de ces informations, nous allons pouvoir écrire un fichier *state* générique indiquant quelles opérations appliquer lors d'une demande de synchronisation :

```
{% for rcfile, remote in pillar['dotfiles'].
iteritems() %}
/home/imil/{{ rcfile }}:
  file.managed:
    - source: salt://{{ remote }}
    - user: imil
    {% if grains['os'] == 'NetBSD' %}
    - group: wheel
    {% else %}
    - group: imil
    {% endif %}
    - mode: 644
{% endfor %}

/home/imil/.vim:
  file.directory:
    - user: imil
    - recurse:
      - user
```

Là encore, le *template* utilise *Jinja* pour itérer dans la structure de données *dotfiles* disponible via *pillar*. À l'aide des *grains* on choisit le groupe propriétaire du fichier créé.

On s'assure que tous les fichiers du répertoire **.vim** appartiennent bien à l'utilisateur visé à l'aide de la fonction **file.directory** et son paramètre **recurse**.

Finalement, on applique le *state* aux *minions* par la commande :

```
$ sudo salt '*' state.highstate
```

Il sera toutefois judicieux, avant tout envoi, de valider le bon fonctionnement et la cohérence des states par la commande :

```
sudo salt '*' state.show_highstate
```

dont voici un extrait :

```
ragnos:
    ----------
    /home/imil/.bashrc:
        ----------
        __env__:
            base
        __sls__:
            imilhome
        file:
            ----------
            - source:
                salt://.bashrc
            ----------
            - user:
                imil
            ----------
            - group:
                wheel
            ----------
```

```
        - mode:
            644
        - managed
/home/imil/.profile:
    ----------
    __env__:
        base
    __sls__:
        imilhome
    file:
        ----------
        - source:
            salt://.profile
        ----------
        - user:
            imil
        ----------
        - group:
            wheel
        ----------
        - mode:
            644
        - managed
/home/imil/.tmux.conf:
    ----------
    __env__:
        base
    __sls__:
        imilhome
    file:
        ----------
        - source:
            salt://.tmux.conf
        ----------
        - user:
            imil
        ----------
        - group:
            wheel
        ----------
        - mode:
            644
        - managed
/home/imil/.vim:
    ----------
    __env__:
        base
    __sls__:
        imilhome
    file:
        ----------
        - user:
            imil
        ----------
        - recurse
            - user
        - directory
/home/imil/.vim/colors/molokai-trans.vim:
    ----------
    __env__:
        base
    __sls__:
        imilhome
```

```
    file:
        ----------
        - source:
            salt://.vim_colors_molokai-
trans.vim
        ----------
        - user:
            imil
        ----------
        - group:
            wheel
        ----------
        - mode:
            644
        - managed
/home/imil/.vim/colors/molokai.vim:
    ----------
    __env__:
        base
    __sls__:
        imilhome
    file:
        ----------
        - source:
            salt://.vim_colors_molokai.vim
        ----------
        - user:
            imil
        ----------
        - group:
            wheel
        ----------
        - mode:
            644
        - managed
/home/imil/.vimrc:
    ----------
    __env__:
        base
    __sls__:
        imilhome
    file:
        ----------
        - source:
            salt://.vimrc
        ----------
        - user:
            imil
        ----------
        - group:
            wheel
        ----------
        - mode:
            644
        - managed
/tmp/foo:
    ----------
    __env__:
        base
    __sls__:
        test
    file:
        - managed
        ----------
        - source:
            salt://foo
```

Comble du raffinement, lorsque le fichier à diffuser est déjà présent sur le *minion* visé, *Salt* affichera le différentiel entre les deux versions au format *diff*. *Like a sir*.

5 One spice to rule them all

Au sein de mon entreprise dont l'activité principale consiste à manipuler un nombre conséquent de machines virtuelles de façon industrielle toute la journée (un *cloud*, donc), de nombreuses solutions d'orchestration ont été évalué. Inégales tant sur leur courbe d'apprentissage que sur le niveau de complexité infrastructurelle, *Salt* nous est apparu comme une suite cohérente, abordable rapidement et s'intégrant parfaitement dans un éco-système régi en grande partie par *Fabric* **[7]**, lui aussi écrit en *Python*. Bien que jeune, la qualité des développement et la structuration de sa communauté, le projet *Salt* est probablement sur le point de distribuer une nouvelle donne dans le monde bouillonnant de la gestion du changement (+10 points bullshit-bingo). ■

Références

[1] http://saltstack.com/

[2] http://www.zeromq.org/

[3] http://docs.saltstack.com/topics/installation/debian.html

[4] http://docs.saltstack.com/ref/modules/

[5] http://www.yaml.org/

[6] http://jinja.pocoo.org/

[7] http://docs.fabfile.org/en/1.6/

SALT : UN PEU PLUS D'ASSAISONNEMENT

par Emile <iMil> Heitor [pour GCU canal historique et la secte des serviettes de bain oranges]

Nous avons précédemment vu comment le système de gestion de configuration Salt nous permettait de contrôler différents aspects d'un parc de machines, par exemple en s'assurant que des fichiers de configuration communs étaient tous synchronisés et que les paquets nécessaires aux services souhaités sur ces serveurs étaient bien installés et à jour. Bien que le nombre de modules fournis par défaut par Salt permette l'orchestration d'un parc relativement standard, pour une infrastructure de grande envergure ou pour des besoins spécifiques, il peut s'avérer indispensable de créer son propre module de contrôle, c'est cet aspect du logiciel que nous allons explorer ici.

1 Vogue, petit module, vogue

Un *master Salt* sait évidemment communiquer avec ses *minions* pour les interroger ou leur faire exécuter des actions et dans ce dernier cas, une panoplie de possibilités permet de façonner les *minions* avec grande souplesse.

Salt est livré muni d'une quantité remarquable de modules, présents dans **`${PREFIX}/lib/pythonX.Y/{site,dist}-packages/salt/modules`**. Mais d'emblée, nous sommes en mesure d'étendre cette liste sans polluer le contenu du répertoire placé sous le contrôle du paquet fourni par votre système. Pour ce faire, il suffit, dans le répertoire défini comme **`file_roots`** dans le fichier de configuration de *salt-master* d'ajouter un répertoire **`_modules`** (notez l'*underscore* devant le nom du répertoire). Une fois le répertoire **`_modules`** peuplé de modules maison, ces derniers seront synchronisés sur les *minions* grâce à la commande :

```
# salt '*' saltutil.sync_modules
```

Où, comme vu dans l'article de présentation de la suite *Salt*, **`'*'`** représente l'*environnement* de base (tous les *minions*) ; on aurait pu spécifier un minion en particulier ou une condition relative à une catégorie de *minions*. Par exemple, pour cibler les *minions* dont le système d'exploitation est *Debian GNU/Linux*, on aurait fait :

```
# salt -G 'os:Debian' saltutil.sync_modules
```

Utilisant ainsi les *grains*, valeurs statiques chargées au démarrage des *minions*.

Deux intérêts majeurs à la possibilité de synchroniser ainsi dynamiquement des modules :

- Utiliser des modules spécifiques à votre environnement.
- Tester des modules que vous souhaiteriez publier plus largement et pourquoi pas les soumettre au projet *Salt* pour inclusion dans le logiciel lui même.

Munis de cette facilité, nous allons pouvoir démarrer l'apprentissage de l'écriture d'un module simple et nous familiariser ainsi avec les multiples possibilités offertes par l'environnement *Salt*.

2 Hello, Salt

Nous en avons parlé dans l'article d'introduction, *Salt* est écrit en python, et tout naturellement, les modules sont également proposés dans ce langage, simplifiant grandement leur écriture. Nous allons donc écrire un simple module qui renvoie une chaîne de caractères et constater son fonctionnement sur un sous-ensemble de *minions*. Exécution :

```
$ cat ~/salt/_modules/hello.py
def hello():
        return 'Hello, Salt!
```

On synchronise ce module comme vu précédemment, je choisis ici mes machines *NetBSD* :

```
# salt -G 'os:NetBSD' saltutil.sync_modules
watto:
    - modules.hello
korriban:
    - modules.hello
coruscant:
    - modules.hello
exar:
    - modules.hello
ragnos:
    - modules.hello
```

Le maître nous informe que le module **hello** a été synchronisé sur nos *minions* dont le *grain* « os » vaut *NetBSD*. Il nous est maintenant possible de faire appel à notre nouveau module de la façon la plus simple qui soit :

```
$ sudo salt -G 'os:NetBSD' hello.hello
exar:
    Hello, Salt!
watto:
    Hello, Salt!
coruscant:
    Hello, Salt!
korriban:
    Hello, Salt!
ragnos:
    Hello, Salt!
```

Simple non ?

Le développement de modules, comme tout développement, pouvant s'avérer semé d'embûches, il n'est pas obligatoire de passer systématiquement par la machinerie *master - minion*. L'utilitaire **salt-call** permet de réaliser un appel direct :

```
# salt-call hello.hello
[INFO    ] Configuration file path: /usr/pkg/etc/salt/minion
local:
    Hello, Salt!
```

Ce petit exemple ouvre d'ores et déjà un nombre de possibilités assez vaste mais sa portée reste limitée. Héritant des capacités objet de *Python*, *Salt* permet, au sein d'un module, de faire appel à toute sa machinerie interne.

Faisons-donc appel, dans notre module trivial, à la notion de *grains* :

```
$ cat hello.py
def hello():
       if __grains__['kernelrelease'] < '6.1':
              return 'VIEUX!'
       return 'JEUNE! {0}'.format(__grains__['kernelrelease'])
```

Après avoir synchronisé cette nouvelle version, nous pouvons constater son bon fonctionnement :

```
# salt -G 'os:NetBSD' hello.hello
watto:
    JEUNE! 6.1_RC4
coruscant:
    VIEUX!
exar:
    VIEUX!
korriban:
    VIEUX!
ragnos:
    VIEUX!
```

Pour rappel, on consultera la liste des *grains* disponibles via la commande :

```
# salt 'watto' grains.items
watto:
  cpu_flags: FPU DE PSE TSC MSR PAE MCE CX8 APIC SEP MTRR PGE MCA
CMOV PAT PSE36 CFLUSH MMX FXSR SSE SSE2 SSE3 CX16 POPCNT RAZ XD LAHF
  cpu_model: Intel 686-class
  cpuarch: i386
  defaultencoding: ISO8859-15
  defaultlanguage: fr_FR
  domain: home.imil.net
  fqdn: watto.home.imil.net
  gpus:
      {'model': 'CL-GD5446', 'vendor': 'Cirrus Logic'}
  host: watto
  id: watto
  ipv4:
      127.0.0.1
      192.168.1.151
  kernel: NetBSD
  kernelrelease: 6.1_RC4
  localhost: watto.home.imil.net
  master: coruscant
  mem_total: 511
  nodename: watto.home.imil.net
  num_cpus: 2
  num_gpus: 1
  os: NetBSD
  os_family: NetBSD
  osrelease: 6.1_RC4
  path: /usr/bin:/bin:/usr/pkg/bin:/usr/local/bin:/sbin:/usr/sbin:/
usr/pkg/sbin:/usr/X11R7/bin:/home/imil/bin
  ps: ps auxwww
  pythonpath:
      /usr/pkg/bin
      /usr/pkg/lib/python27.zip
      /usr/pkg/lib/python2.7
      /usr/pkg/lib/python2.7/plat-netbsd6
      /usr/pkg/lib/python2.7/lib-tk
      /usr/pkg/lib/python2.7/lib-old
      /usr/pkg/lib/python2.7/lib-dynload
      /usr/pkg/lib/python2.7/site-packages
  pythonversion: 2.7.3.final.0
  saltpath: /usr/pkg/lib/python2.7/site-packages/salt
  saltversion: 0.15.1
  server_id: 1276300030
  shell: /usr/pkg/bin/bash
  virtual: kvm
```

Toutes les clés et valeurs sont disponibles, soit à partir de la ligne de commande à la suite du *flag* **-G**, soit directement dans le module à l'aide du **dict __grains__**.

3 Pass'pass'l'module

Au delà des *grains*, c'est à l'intégralité des fonctionnalités de *Salt* auxquelles nous avons accès dans notre module, ceci à travers le *dict* **__salt__**. Accédons par exemple aux fonctions d'exécution :

```
$ cat hello.py
def hello(directory):
        return __salt__['cmd.run']('ls {0}'.format(directory))

$ sudo salt 'coruscant' saltutil.sync_modules

$ sudo salt-call hello.hello '/usr'
[INFO    ] Configuration file path: /usr/pkg/etc/salt/minion
[INFO    ] Executing command 'ls /usr' in directory '/root'
local:
    X11R6
    X11R7
    bin
    games
    include
    lib
    libdata
    libexec
    local
    mdec
    obj
    obj-i386
    pkg
    pkgsrc
    sbin
    share
    src
    tests
    tools
    tools-i386
```

Ici, nous exécutons la commande **ls** suivie d'un paramètre variable (**directory**) en faisant appel au module **cmdmod.py**, présent dans l'installation de base de *Salt*. Ce dernier expose des fonctions d'interfaçage avec l'exécution de commandes sur le *minion* ciblé. Bien évidemment, notre module **hello** est également immédiatement disponible pour tous les autres modules chargés. Par exemple, lors de l'écriture de **hello2**, dans un fichier distinct **hello2.py**, nous sommes parfaitement autorisés à utiliser les fonctionnalités de **hello.py** :

```
$ cat hello2.py
def hello2():
        return __salt__['hello.hello']('/stand')

$ sudo salt 'coruscant' saltutil.sync_modules
coruscant:
    - modules.hello2

$ sudo salt-call hello2.hello2
[INFO    ] Configuration file path: /usr/pkg/etc/salt/minion
[INFO    ] Executing command 'ls /stand' in directory '/root'
local:
    amd64
```

La fonction **hello2** fait appel à la fonction **hello** précédemment écrite dans **hello.py** en lui passant en paramètre le répertoire **/stand**. **hello2** retourne à *Salt* la valeur de retour de la fonction **hello** que nous avons appelé via le dict **__salt__** qui donne un accès direct à cette dernière.

4 Transformation

Dans l'exemple précédent, vous aurez probablement remarqué qu'alors qu'on accède aux fonctions d'exécution via l'appel à **cmd.<fonction>**, le module se nomme en réalité **cmdmod.py**. Bien plus qu'un simple renommage, le module utilise ici une puissante fonctionnalité: les modules virtuels.

L'un des intérêts de l'orchestration réside dans la généricité des fonctions appelées, par exemple, il serait fastidieux d'appeler séparément **apt.install**, **pkgin.install** ou encore **yum.install** pour écrire nos recettes d'installation de paquets sur notre parc hétérogène. Au lieu de cela, nous appelons *Salt* de façon générique :

```
# salt '*' pkg.install vim
```

Pourtant, les fonctionnalités d'installation de logiciels tiers sont belles et bien présentes dans plusieurs modules, aux noms distincts. Quelle est cette sorcellerie ? C'est ici la notion de module virtuel qui se charge d'appeler le bon *backend* pour installer le paquet. Par exemple, dans **pkgin.py**, j'ai ajouté :

```
def _check_pkgin():
    '''
    Looks to see if pkgin is present on the system, return full path
    '''
    return salt.utils.which('pkgin')


def __virtual__():
    '''
    Set the virtual pkg module if the os is supported by pkgin
    '''
    supported = ['NetBSD', 'SunOS', 'DragonFly', 'Minix', 'Darwin']

    if __grains__['os'] in supported and _check_pkgin():
        return 'pkg'
    else:
        return False
```

Ainsi, si le système d'exploitation est supporté par **pkgin** et que **pkgin** est accessible dans l'un des **$PATH** (**salt.utils.which**), nous informons *Salt* que ce module sera appelé via son nom virtuel **pkg**. Autre avantage de cette fonctionnalité, on évitera élégamment les conflits de nommage en créant son module *Python* avec un nom différent que celui par lequel il est invoqué, exactement comme cela est réalisé pour les fonctions **cmd** inscrites dans **cmdmod.py**.

5 Une dernière pincée

Quelques astuces supplémentaires permettront à vos futurs modules de passer la barrière du *pull-up request*, participation au projet rendue extrêmement simple grâce au site *GitHub* **[1]**.

5.1 Les fonctions internes

Il n'est pas rare, dans tout type de projet, d'écrire des fonctions qui n'ont qu'une portée interne, dans notre cas, des fonctions qui ne seront pas exposées aux *minions*. Cela se réalise très simplement en préfixant le nom de la fonction par un *underscore*, par exemple :

Fonction chargée par le *minion* :

```
def foo():
return 'bar'
```

Fonction non-chargée par le *minion* :

```
def _foo():
return 'bar'
```

5.2 Do-cu-men-tez !

La méthode de documentation d'un module *Salt* n'est pas différente de celle employée pour un code *Python* classique. En renseignant proprement un *docstring* dans vos fonctions, ces dernières seront consultables par la commande `Salt sys.doc` :

```
def foo():
'''
This function returns the number of fooes.

CLI Example::

salt '*' test.foo
'''
```

6 Quoi, vous êtes encore là ?

Accordez-moi que le développement de modules *Salt* brille par sa simplicité, mais également par sa souplesse et ses concepts inhérents. De plus, l'excellente lisibilité des modules présents dans l'installation de base vous permettra de vous mettre au travail rapidement. Le projet étant développé collaborativement sur *GitHub*, il est aisé de se faire une idée assez précise des structures et modèles utilisés. Enfin, le canal `#salt` sur le réseau *IRC freenode* est peuplé de nombreux développeurs du projet dont la sympathie et l'entrain ne gâchent rien au plaisir de participer à un tel projet. ■

Référence

[1] https://github.com/saltstack/sal

TENEZ LA CHARGE À L'AIDE DE INFINISPAN

par Romain Pelisse [Middleware Consultant @ Red Hat Gmbh] & Valentine Vierne [Relecture]

Plus que jamais avec le « cloud » et la vague du NoSQL, les problématiques de montée en charge et de passage à l'échelle (scalability, dans le langage de Shakespeare) prédominent dans la conception d'application en ligne. Comment parvenir à atteindre le nombre d'utilisateurs concurrents attendu ou espéré ? Comment supporter un pic de charge, lorsque cette charge s'accroît temporairement ? Comment améliorer ou conserver son temps de réponse moyen, malgré cette charge variable et généralement grandissante ?

Sans être une solution miracle, nous allons voir, à travers cet article, comment l'ajout d'une infrastructure NoSQL telle qu'une grille de données (data grid), comme InfiniSpan [1], peut aborder ces difficultés et permet même d'envisager l'obtention d'une croissance (presque) linéaire de la tenue en charge du système. En effet, avec une telle infrastructure, cette croissance linéaire, véritable Saint Graal de la conception de système informatique, beaucoup recherché mais rarement trouvé par de nombreuses architectures logicielles (et leurs concepteurs), n'est peut être plus une chimère...

1 Introduction

1.1 Contexte

Avant de décrire dans les grandes lignes le fonctionnement interne d'une grille de données tel que InfiniSpan, nous allons d'abord expliciter un peu la problématique de la montée en charge et en quoi consiste cette **croissance linéaire** que nous avons évoquée un peu plus haut.

Si le temps des systèmes centraux (mainframes) est désormais révolu, c'est que l'on sait maintenant que, pour concevoir des applications robustes et surtout capables de supporter des charges toujours croissante, il faut que l'infrastructure de ces dernières soit **extensible**. Il n'est, en effet, pas aisé de remplacer un système monolithique qui a atteint sa limite, mais il est relativement simple d'ajouter de nouvelles machines dans un système **distribué** toujours en production.

Ainsi, plus le nombre de requêtes, en entrée du système, augmente, plus on placera de machines, chacune exécutant une (ou plusieurs) instance de l'application cible. Dans un monde parfait, on peut donc ainsi supporter n'importe quelle charge, simplement en ajoutant autant de machine que nécessaire. Chaque machine ne recevant qu'une partie de la charge pour laquelle elle est en mesure d'assurer le temps de réponse souhaité, la croissance de la charge ne réduit pas les performances du système.

On peut même théoriquement supporter une charge infinie, simplement en mettant en place une infinité de machine, et donc assurer malgré tout le même temps de réponse. On notera que ceci est bien évidemment que théorique.

En effet, cette situation n'existe pas dans la réalité; pour, au moins deux raisons :

- **partage d'état ou de données** : souvent les instances de l'application doivent partager un état ou certaines données et la synchronisation de ces derniers ralentit de plus en plus le système avec l'ajout de nouvelles machines ;
- **source de données** : les sources de données sont généralement uniques et centralisées et, tôt ou tard, elles deviennent ainsi rapidement le goulot d'étranglement du système (bottleneck).

Arrêtons-nous pour le moment sur cette relativement triste conclusion ou état de fait et voyons maintenant un peu comment fonctionne une grille de données et comment ces dernières vont apporter une solution, peut être partielle, mais néanmoins très efficace, à cette problématique.

1.2 Fonctionnement d'une grille de données

Sans rentrer dans les considérations les plus avancées et mathématiques liées aux grilles de données, on va retenir,

pour cet article, les éléments suivants, qui permettront de bien comprendre les points illustrés ici :

- une grille de données stocke les informations sous forme d'un duo clé (K), valeur (V) (du moins dans notre cas) ;
- une grille de données est composée d'un nombre N d'instances ;
- l'applicatif client de cette grille connaît le nombre d'instances (N) de sa grille ;
- à l'aide d'un algorithme « magique », à partir du nombre (N) d'éléments dans la grille et de la clé (K), le client est capable de déterminer dans quelle instance (X) parmi (N) placer la valeur (V) (soit x = f(N,V)) ;
- ainsi l'algorithme permet toujours de déterminer quelle instance devrait être le « propriétaire » (owner) de la valeur (V) associée à la clé (K).

A l'aide de ces quelques propriétés, on comprend tout de suite que quelque soit le nombre d'instances N, retrouver une information dans la grille ne coûtera qu'un **seul** appel réseau, vers l'instance désignée comme propriétaire de l'information. Ainsi, si chaque élément de la grille permet de stocker, par exemple, 4GB d'information, une grille composée de dix instances peut stocker jusqu'à 40GB d'information, sans changer le temps de réponse des applications clientes !

Avant de continuer, pour faire taire immédiatement les remarques des sceptiques, qui ont le sain réflexe de toujours penser au « pire » quand on leur présente une solution (surtout si elle est vendue comme solution « sans défaut » à tous leurs problèmes), on ajoutera que la grille de données est aussi capable de survivre à la perte d'une ou même de plusieurs de ses instances. Bien sûr les performances du systèmes se dégraderont, pendant quelques instants et, bien évidemment, les informations placées dans le ou les noeuds défaillants, seront perdues, mais la grille restera fonctionnelle.

Comment la grille peut elle survivre à un tel désastre ? Simplement parce que quand le ou les noeuds défaillants seront redémarrés, ou remplacés par de nouvelles instances, la grille les réintégrera naturellement et commencera un processus de synchronisation pour restaurer, si possible, leur données, ou au moins les intégrer comme des « nouveaux membres » en son sein.

En outre, pour se prémunir de la perte d'information en cas de panne, on peut aussi demander à chaque noeud de la grille de répliquer les données qu'il contient sur d'autres noeuds. Ainsi, si le noeud propriétaire d'une information est défaillant, la grille sera néanmoins en mesure de retrouver l'information sur un ou même plusieurs autres noeuds. La perte d'une instance n'aboutira donc plus à une perte définitive de données.

Assumant que les temps de réponse de tous les noeuds sont plus ou moins identiques, il est donc possible de faire grandir la grille, pour répondre à toute évolution du besoin, sans dégrader les performances de l'application. Ainsi on commence à voir comment une telle infrastructure peut répondre aux problématiques évoquées dans la première partie de cet article...

2 Cas d'utilisation d'une grille de données

Avant de passer à la partie pratique, nous allons évoquer deux cas concrets d'utilisation d'une grille de données qui donneront un peu de substance à la suite de notre article. Ces exemples permettront non seulement de mieux comprendre les motivations derrières les configurations évoquées mais aussi de bien saisir le fonctionnement d'une telle infrastructure.

Une fois ces cas d'usage évoqués, nous passerons à la partie résolument plus pratique de cet article, soit comment utiliser InfiniSpan pour implémenter de ces stratégies.

2.1 Utilisation de la grille de données comme cache distribué

2.1.1 Coût de l'accès à une base de données et de l'ORM

Les problématiques de « passage à l'échelle » évoquées plus haut sont très courantes dans les applications Web ou en ligne, où les bases de données, généralement SQL, qui servent les informations nécessaires à l'application sont rapidement débordées par les très nombreuses requêtes provenant des différentes instances de l'application.

Néanmoins, en étudiant ces requêtes, on se rend rapidement compte que la plupart de ces dernières ne sont que des accès en lecture et non en écriture et que très souvent elles concernent le même sous ensemble de données. En effet, de nombreux internautes demandent généralement accès aux mêmes informations, au même moment.

Pour illustrer ceci par un exemple parlant, imaginons un site de vente de musique. Sur ce dernier, les dernières parutions et les discographies de grands artistes (disons plutôt d'artistes qui vendent beaucoup de disques) seront très régulièrement consultées, alors que d'autres éléments du catalogue, généralement la plus grande partie, ne le seront que très rarement ou du moins dans une fréquence sans commune mesure.

Bien évidemment, les concepteurs de bases de données SQL sont bien conscients de ce problème et ont appris, depuis longtemps, à mettre en cache les informations les plus demandées, pour pouvoir les retourner rapidement. Mais il n'en reste pas moins que :

- l'application sollicite la base de données - impliquant un appel réseau et transactionnel ;
- et qu'elle doit reconstruire la réponse, retournée sous forme d'un tableau de résultat SQL, dans un format qui lui est propre (souvent sous forme d'objets) avant de pouvoir utiliser ce résultat.

Ce dernier point n'est pas négligeable, car la plupart des applications sont désormais développées à l'aide d'un langage orienté objet, ce qui implique que l'application doit souvent réaliser un lourd travail de cartographie (« mapping ») entre les données retournées de manière « brute » par la base de données et le modèle objet de l'application. Sans ce travail, l'application ne peut simplement pas utiliser les données retournées.

2.1.2 Placer les résultats dans un cache applicatif

Pour s'épargner ce travail de « Object Relation Mapping » **[XI]** et, surtout, pour diminuer la charge de la base de données, la stratégie la plus souvent retenue, et mise en place, consiste donc à utiliser un **cache applicatif** (applicatif par opposition au cache déjà en place dans les bases de données).

En effet, ce cache, qui jouera le rôle d'intermédiaire, sera consulté avant la base de données. On y placera donc le ou les résultats obtenus pour une requête mais directement sous la forme d'objet, de manière à ne plus avoir besoin de faire appel à la base de données et d'effectuer un travail de reconstruction.

En effet, dans le cas de Java par exemple, on peut facilement sérialiser, sous la forme d'un flux binaire, un objet et ses dépendances **[JavaSerialisation]**. Ainsi, on peut placer directement ce flux dans le cache, et lorsqu'elle accédera de nouveau à cette donnée à travers le cache, l'application pourra rapidement reconstruire les données associées. (Ce processus porte le nom de désérialisation).

2.1.3 Conception de la clé

Reste un point à éclaircir, comment l'application peut elle retrouver l'information placée dans le cache ? Quelle clé utilise-t'elle pour stocker le résultat d'une requête SQL ?

De manière très simple, elle peut simplement utiliser la requête SQL en elle même ! Ou, à des fins d'optimisation, et pour réduire grandement la taille de la clé, on pourra simplement utiliser une fonction de hash sur cette dernière.

Un rapide exemple de cette stratégie en Python :

```
>>> import hashlib
>>> x = hashlib.sha256("SELECT * FROM records where artist = 'Madonna'")
>>> x.hexdigest()
'08e905159601ee81d9a210a82bc1b26481fb7c4ec2f660c1187c83929e208439'
```

On notera qu'une telle stratégie peut aboutir, selon le modèle de donnée de l'application, à beaucoup d'informations dupliquées dans le cache. En effet deux requêtes différentes mais portant sur des données semblables (par exemple, un même artiste), verront potentiellement leurs résultats, ayant une partie de leurs données en commun, être tout les deux placés dans le cache. Il existe là aussi des techniques de programmation pour optimiser ceci et éliminer la redondance, mais elles dépassent un peu le cadre de cet article.

2.1.4 Éviction

A l'aide d'une telle stratégie, seule la première exécution d'une requête aboutira à un appel à la base de données. Le reste du temps, les données se trouveront simplement dans le cache. Pour s'assurer que le cache ne contienne pas l'ensemble des informations issues de la source de données, mais seulement les données les plus souvent accédées, le cache applicatif doit mettre en place une politique d'**éviction**.

Avec InfiniSpan, comme avec la plupart des technologies similaires, il est possible de configurer le système pour procéder à une éviction de certaines données dès lors qu'elles remplissent certaines conditions. Une première politique d'éviction, relativement simple à mettre en place, consiste, par exemple, à retirer une donnée dès lors qu'elle se trouve dans le cache depuis un certain temps. Ce type d'éviction se base donc sur une méta donnée, associée à l'entrée dans le cache, désignée sous le nom de « temps de vie » (Time To Live ou TTL) **[TTL]**.

Une autre stratégie très commune consiste à limiter le nombre d'entrées dans le cache et à simplement retirer la plus ancienne entrée lorsqu'il faut en ajouter une nouvelle. Cette stratégie suit donc l'algorithme connu sous le nom de FIFO (First In, First Out) **[FIFO]**, souvent étudié dans la plupart des enseignements liés à l'informatique.

Ces stratégies sont relativement simples et faciles à mettre en place mais elles ne correspondent pas tellement au scénario que nous étudions. Heureusement pour nous, il existe

LRU versus LRIS

Si cet algorithme propose une stratégie saine et appropriée à notre cas d'utilisation, il est pertinent de signaler que, parfois, ce dernier n'est pas très performant. En effet, avec LRU, les données accédées qu'une seule fois ne seront pas pour autant rapidement remplacées et d'autres qui seront peut être bientôt accédées de nouveau, sont malheureusement remplacées. Ainsi, depuis quelques années, une nouvelle approche, désignée par l'acronyme LRIS (Low Inter-reference Recency Set **[LIRS]**), est de plus en plus souvent utilisée, à la place de LRU.

une autre stratégie, beaucoup plus adaptée à notre cas, qui consiste, fort intelligemment, à limiter la taille du cache en retirant, quand c'est nécessaire, l'entrée la moins consultée. Ainsi, après un certain temps d'exécution de l'application, le cache ne contiendra qu'un sous-ensemble très pertinent de données, formées des demandes les plus fréquentes des utilisateurs de l'application.

Cette stratégie est très souvent désignée sous l'acronyme LRU, soit Least Recently Used en anglais, et elle est, de loin, la plus adaptée à notre scénario. C'est donc celle que nous allons retenir et que nous implémenterons un peu plus loin.

2.1.5 Invalidation des données dans le cache

Un dernier élément technique de ce scénario doit être étudié avant d'aller plus loin: la gestion de l'invalidation des données placées en cache. En effet, en cas de modification d'une données, par exemple la mise à jour du prix d'un produit ou l'ajout d'un nouvel album d'un artiste, l'entrée associée dans le cache ne sera plus exacte, et il faudra donc procéder à son **invalidation**.

Concrètement, l'invalidation des données est similaire à l'éviction déjà évoquée puisque la donnée est simplement retirée du cache. Néanmoins ce qui diffère grandement, c'est la cause du retrait. On n'invalide une donnée dans le cache que si, et seulement si, elle n'est plus cohérente avec la source de données, alors que l'éviction est généralement motivée par la conservation d'une taille de cache raisonnable.

Cette invalidation peut simplement être effectuée par l'application elle même, dès lors qu'elle effectue la modification d'une donnée en base. Dans le même temps, elle peut en effet notifier le cache de supprimer l'entrée associée. Néanmoins, si la modification des données est effectuée par une application tierce ou directement dans la base de données, cette stratégie n'est pas suffisante et il faut utiliser un mécanisme propre au cache pour s'assurer de l'invalidation des données.

On peut alors demander au cache, donc dans notre cas InfiniSpan, de procéder automatiquement à une invalidation des données selon certains critères. La stratégie d'invalidation la plus élémentaire, fort comparable à l'utilisation d'un TTL pour l'éviction, consiste à fixer, pour chaque entrée placée dans le cache, une durée de vie (lifespan). Une fois cette durée de vie écoulée, le cache retire donc l'élément et force, de facto, l'application à rafraîchir cette entrée, en retournant consulter la base de données.

Dans le cas, par exemple, d'une boutique en ligne, où le cache contient une partie d'un catalogue, il semble pertinent de fixer cette durée de vie à une valeur de l'ordre de la journée ou la demi-journée. Ainsi, si une personne procède à une modification de certaines données en base, on peut prédire de manière fiable que l'entrée associée dans le cache sera mise à jour dans les 24 heures qui suivent.

Avec un délai de rafraîchissement de 12h ou 24h, on est également sûr de ne pas trop diminuer les performances de l'application, en augmentant de manière trop fréquente les requêtes vers la base en vue de rafraîchir le contenu du cache. On signalera, au passage, qu'une manière simple de vérifier l'impact de ce paramétrage, est de surveiller le hit/miss ratio du cache.

En effet, ce taux indique combien de fois la donnée demandée n'a pas été trouvée dans le cache. Par construction, ce ratio devrait être proche de 1 ou de 100%. Si ce n'est pas le cas, il est important d'étudier ceci en détail, car il est probable que les données placées en cache ne sont simplement pas pertinentes pour l'usage de l'application.

2.1.6 Dimensionner le cache

Nous avons donc désormais défini quelles données seront placées dans le cache, ainsi que la clé utilisée, et la politique d'éviction (LRU) et d'invalidation de ces données (durée de vie de 12h/24h). Il ne nous reste donc plus qu'à déterminer la taille du cache et à dimensionner la grille en conséquence.

Si par exemple la base de données contient plus de 40GB de données mais que seuls 12GB sont les plus couramment accédées, on peut simplement démarrer deux instances de 8GB pour composer une grille qui sera tout à fait à même de répondre à nos besoins...

Il est à signaler que, arrivé à ce stade, la base de données de production du système existant devrait être en mesure de fournir toutes les informations nécessaires à un dimensionnement judicieux du cache. Dans le cas d'une première version de l'application, il faudra se contenter d'une approche plus empirique.

Néanmoins, l'ajout d'instances à une grille de données ne posant pas de réelle difficulté, il sera aisé d'ajouter ou d'enlever des noeuds à cette dernière si le dimensionnement original du système devait se révéler, une fois en production, inadapté.

2.2 Utilisation de la grille comme stockage temporaire de données

Voyons maintenant un cas d'utilisation un peu différent, mais utilisant les fondamentaux que nous venons de décrire. Admettons que votre compagnie propose plusieurs sites Web à ses employés et clients. Chacun propose des services différents mais vous souhaiteriez fédérer l'authentification, pour tous ces sites (Single Sign On (SSO)). Ainsi, une fois authentifiés sur un des sites, les utilisateurs pourront naviguer sur l'ensemble des sites sans saisir de nouveau leurs données d'identification.

Les données d'authentification en tant que telles sont très petites, mais le nombre d'utilisateurs est, à l'inverse,

rapidement très conséquent. Il faut donc mettre en place un service capable de gérer un très grand nombre de données tout en répondant rapidement. En effet, les différents sites devront vérifier, pratiquement à chaque accès fait par l'utilisateur, que ce dernier est bien authentifié.

On peut déjà noter que le modèle de données en lui-même s'adapte très bien à une base NoSQL de type clé/valeur telle qu'une grille de données comme InifiniSpan. En effet, la clé est naturellement l'identifiant de l'utilisateur et la valeur, les données d'authentification de ce dernier.

Lors de la première authentification, on placera donc les informations de l'utilisateur dans la grille de données et par la suite, lorsqu'il naviguera, les applications associées aux autres sites n'auront qu'à consulter la grille pour vérifier que l'utilisateur est proprement authentifié et dispose des privilèges nécessaires pour accéder aux ressources demandées.

A l'inverse du précédent scénario où le gain de performance était en partie réalisé par la suppression du travail de construction d'objet à partir du résultat, l'intérêt de l'utilisation de la grille se trouve avant tout dans sa capacité à agir comme une autorité centrale et partagée de données d'authentification. En outre, la grille permet de gérer la croissance du nombre d'utilisateurs de manière dynamique.

Si le précédent cas d'utilisation nous a permis d'évoquer la notion d'invalidation, celui-ci va nous permettre de parler d'**expiration**. En effet, rares sont les internautes consciencieux qui pensent à se déconnecter, mais il faut pourtant bien retirer, au bout d'un certain temps, les informations d'authentification de l'utilisateur - ne serait-ce que pour des raisons de sécurité.

Heureusement, notre grille de données permet de définir, comme nous l'avons vu plus haut, pour chaque élément qu'elle contient une **durée de vie** (lifespan). Ainsi, une fois un certain temps écoulé, une donnée sera naturellement retirée de la grille.

3 Mise en place pratique de la grille

Passons maintenant à la partie pratique de cet article et mettons en place, en local, une petite grille, composée de seulement deux nœuds, que nous configurerons étape par étape pour répondre aux différents besoins évoqués précédemment dans nos deux scénarios.

3.1 Quelle JVM pour InfiniSpan ?

Avant tout, pour démarrer InfiniSpan, la commande Java est nécessaire. Sur une distribution Linux à jour, on peut aisément installer la machine virtuelle Open Source OpenJDK, qui est tout aussi fiable. Pour s'en convaincre, on peut noter que Red Hat supporte l'utilisation de JBoss Data Grid **[JDG]**, la version « produit » de InfiniSpan, sur la machine virtuelle de Sun/Oracle comme sur OpenJDK. Ce qui est bien naturel, puisque Red Hat est aussi un contributeur très actif sur ce projet **[OpenJDK-RedHat]**.

Une autre solution, si vous préférez utiliser la JVM fournit par Sun/Oracle, consiste à récupérer cette dernière sur le site de téléchargement de la compagnie et d'installer, manuellement, cette machine virtuelle. Dans le cadre de cet article, nous ferons simple et utiliserons OpenJDK :

```
$ yum install -y java-1.7.0-openjdk-devel
```

Note : il est important de prendre la version `devel` du paquet, pour être sûr d'installer non seulement la commande 'java' mais aussi le compilateur qui lui est associé, soit la commande `javac`. Ceci permettra de compiler le code client un peu plus loin.

Bien évidemment, pour une installation en production, le compilateur ne sera pas nécessaire et il faudra au contraire faire attention à n'installer que la machine virtuelle :

```
$ yum install -y java-1.7.0-openjdk
```

3.2 Installation de InfiniSpan

Si certains produits venant de la communauté JBoss, comme le fameux serveur d'applications éponyme (enfin, éponyme jusqu'à il y a encore peu **[XXX]**), ont enfin trouvé leur chemin dans les paquets de distribution Linux tel que Red Hat Entreprise Server ou Fedora, InfiniSpan n'est pas encore disponible. Pour installer ce logiciel, il faut donc revenir aux « bonnes vielles méthodes » et récupérer l'archive sur le site de téléchargement du projet **[IFSP-DOWN]**.

Une fois ceci fait, il suffit simplement, sans surprise, de décompresser cette archive pour installer le logiciel. Pour faire propre, nous allons décompresser cette archive dans le répertoire `/opt/` et créer un lien symbolique pour masquer la version utilisée :

```
$ sudo unzip infinispan-5.2.0.CR1-all.zip -d /opt/
$ sudo ln -s /opt/infinispan-5.2.0.CR1-all/ /opt/infinispan
```

3.2.1 Conception du script de démarrage

Le logiciel est évidemment fourni avec son propre script de démarrage, qui se situe dans le sous-répertoire `bin` et se trouve nommé, de façon logique, `startServer.sh`. Si on l'exécute sans argument, il a le bon goût d'afficher un texte d'aide - c'est suffisamment loin d'être systématique pour être noté et apprécié.

Un extrait de cette aide est reproduit ci-dessous, pour faciliter la lecture et la compréhension de la suite de cet

article, qui fera bien évidemment appel aux arguments décrits ci dessous.

```
$ ./bin/startServer.sh
ERROR: Please indicate protocol to run with -r parameter
usage: startServer [options]
options:
    -h, --help                      Show this help message
    -V, --version                   Show version information
    --                              Stop processing options
    -p, --port=<num>                TCP port number to listen on (default:
11211 for Memcached, 11222 for Hot Rod and 8181 for WebSocket server)
    -l, --host=<host or ip>         Interface to listen on (default:
127.0.0.1, localhost)
    [---]
    -c, --cache_config=<filename>   Cache configuration file (default:
creates cache with default values)
    -r, --protocol=                 Protocol to understand by the server.
This is a mandatory option and you should choose one of these options
        [memcached|hotrod|websocket]
...
```

Notre objectif étant de construire, à partir de InfiniSpan, une grille de données de production, nous allons construire notre propre script de démarrage, à partir de ce script déjà fourni, pour faciliter la mise en place, en tant que service, de ce système.

On peut néanmoins noter les éléments suivants concernant la mise en place de notre serveur :

- le protocole utilisé sera Hot Rod, un protocole binaire, ouvert, mais « propriétaire » (on entend ici que le protocole ne suit aucun standard mais il n'est pas « closed source ») à InfiniSpan, qui est aussi la solution la plus performante et qui permet de facilement exploiter la nature distribuée de la grille de données ;
- le port utilisé sera 11222, pour rester conforme à l'usage établi de ce protocole ;
- l'hôte utilisé sera **localhost**, à des fins de test, mais notre script acceptera un argument pour redéfinir cette adresse vers une IP publique ;
- le fichier de configuration, qui sera partagé entre toutes les instances de notre grille, sera situé, pour respecter l'usage sous Unix, dans le répertoire **/etc/infini-span/** et portera le nom de **infini-span.conf** - mais ce dernier ne sera qu'un lien symbolique vers **/etc/infini-span/infini-span.xml**.

Voyons déjà une première mouture du script de démarrage de InfiniSpan :

```
#!/bin/sh
readonly HOST=${INFINI_SPAN_PUBLIC_IP:-'127.0.0.1'}
readonly PORT=11222
readonly CONFIG='/etc/infini-span/infini-span.conf'
readonly PROTOCOL='hotrod'
readonly ISPN_HOME='/opt/infini-span/'

readonly START_SCRIPT="${ISPN}/bin/startServer.sh"
readonly PIDFILE=${PIDFILE:-'/var/run/infinispan.pid'}

readonly COMMAND="${START_SCRIPT} -c ${CONFIG} -p ${PORT} -l ${HOST}
-r ${PROTOCOL}"

${COMMAND}
```

Ce script permet donc de démarrer aisément InfiniSpan, en utilisant la bonne configuration et les bons paramètres par défaut. Bien évidemment, nous allons étudier en détail la configuration du serveur. Avant ceci, nous allons d'abord, améliorer notre script, pour y inclure la configuration de la journalisation.

3.2.2 Configuration des journaux

Le script de démarrage fournit par InfiniSpan s'appuie sur la présence d'une variable d'environnement, nommée **LOG4J_CONFIG**, qui contient le chemin vers le fichier de configuration de **log4j.xml** :

```
LOG4J_CONFIG=file:///${ISPN_HOME/etc/log4j.xml
```

Nous allons donc réutiliser cet élégant mécanisme dans notre script de démarrage pour pointer vers la configuration de journalisation, que nous allons placer à coté de celle de InfiniSpan :

```
$ cp ${ISPN_HOME}/etc/log4j.xml /etc/infini-span/
```

On édite ce fichier pour changer la configuration du fichier de journalisation pour pointer vers l'usuel **/var/log/** :

```
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/" debug="false">
  <appender name="FILE" class="org.apache.log4j.FileAppender">
    ...
    <param name="File" value="/var/log/infini-span.log"/>
  </appender>
  ...
</log4j:configuration>
```

Modifions maintenant le script de démarrage pour y inclure cette amélioration :

```
#!/bin/sh
readonly HOST=${INFINI_SPAN_PUBLIC_IP:-'127.0.0.1'}
readonly PORT=11222
readonly CONFIG='/etc/infini-span/infini-span.conf'
readonly PROTOCOL='hotrod'
readonly ISPN_HOME='/opt/infini-span/'

export LOG4J_CONFIG=${LOG4J_CONFIG:-'/etc/infini-span/log4j.xml'}

readonly START_SCRIPT="${ISPN}/bin/startServer.sh"

readonly COMMAND="${START_SCRIPT} -c ${CONFIG} -p ${PORT} -l ${HOST} -r ${PROTOCOL}"

${COMMAND}
```

On remarquera qu'on permet à escient de surcharger la définition de la variable **LOG4J_CONFIG**. Ainsi, si l'on souhaite

changer la configuration de la journalisation de l'application, par exemple pour étudier un problème en production, on peut aisément surcharger cette valeur et redémarrer le serveur, sans changer la configuration du système de manière persistante.

3.2.3 Ajout des fonctions de services (start, stop, status, restart)

Nous allons maintenant étendre notre script pour lui ajouter le support des fonctionnalités généralement attendues d'un script de démarrage de service. Dans le contexte de cet article, nous nous limiterons à l'ajout des fonctionnalités **start**, **stop** et **status**.

```
...
readonly PIDFILE='/var/run/infini-span.pid'
...

start() {
  echo -n "Starting InfiniSpan..."
  ${COMMAND} & > /dev/null
  pid=$(sed -e 's/^ *\([0-9]*\) .*$/\1/)
  echo " (pid:${pid}) Done."

  echo ${pid} > ${PIDFILE}
}

stop() {
  pid=$(cat ${PIDFILE})

  kill "${pid}"
  sleep 5
  kill -9 "${pid}"
}

status() {
  pid=$(cat ${PIDFILE}

  kill -0 "${pid}"
  if [ ${?} -ne 0 ]; then
    echo "Instance ($pid) is not running."
  fi
}

case "$1" in
  start)
    start
    ;;
  stop)
     stop
     ;;
  restart)
    stop
    start
    ;;
  status)
    status
    ;;
   *)
     echo $"Usage: $0 {start|stop|status|restart}"
     exit 2
esac
```

L'implémentation de ce script est relativement simple - selon le système cible, il pourrait être amélioré, pour s'adapter au mieux à son environnement (SystemD ou Init V, par exemple). On pourrait aussi envisager de le modifier pour être exécuté par un système de contrôle de processus tel que Daemontools ou Monit. Bien que l'intégration à l'aide de ces derniers outils est intéressante, elle dépasse, malheureusement, le cadre de cet article.

3.3 Mise en place de la seconde instance

Avant d'étudier la configuration de notre serveur InfiniSpan en détail, nous allons aborder un dernier problème relativement important : comment mettre en place, sur le même serveur, une seconde (ou plus) instance de notre grille ? En plus de se présenter pour notre démonstration, la problématique peut facilement apparaître sur des systèmes en production.

En effet, pour de nombreuses raisons, un serveur, même virtuel, peut disposer de suffisamment de processeurs et de mémoire pour justifier le déploiement de plusieurs instances. En outre, il peut être plus performant d'exécuter deux machines virtuelles Java de 8GB plutôt qu'une seule de 16GB.

La configuration avancée des machines virtuelles Java exécutant une instance de InfiniSpan sort du cadre de cet article. Néanmoins, on peut, pour expliciter pourquoi deux JVMs de 8GB peuvent se révéler plus performantes qu'une seule de 16GB, indiquer que - selon l'application et l'usage fait de la grille - le temps passé à exécuter le ramasse-miette Java peut simplement devenir prohibitif si la mémoire utilisée est trop importante.

Comme évoqué plus haut, grâce à notre script de démarrage, il est déjà possible de démarrer une nouvelle instance, utilisant la même configuration, simplement en changeant le numéro du port d'écoute :

```
$ export PORT=11223
$ /etc/init.d/infini-span.sh
```

On pourrait aisément créer un nouveau script **/etc/init.d/infini-span2.sh**, invoquant le premier avec un numéro de port modifié :

```
#!/bin/sh
export PORT=11223
/etc/init.d/infini-span.sh
```

Mais cette solution manque clairement d'élégance. Nous allons donc opter pour une stratégie un peu plus satisfaisante et utiliser désormais un lien symbolique vers notre fichier de démarrage :

```
$ ln -s /etc/init.d/infini-span.sh /etc/
init.d/infini-span-0
$ ln -s /etc/init.d/infini-span.sh /etc/
init.d/infini-span-1
```

Nous allons modifier accordément notre script de démarrage pour qu'il adapte le numéro de port d'écoute du serveur, à partir du nom du lien symbolique :

```
#!/bin/sh
PORT_SHIFT=$(basename "${0}" | sed -e 's/^.*-\([0-
9*]\)$/\1/')

if [ -z "${PORT_SHIFT} ]; then
  PORT_SHIFT=0
fi
readonly PORT_BASE=${PORT_BASE:-'11222'}

readonly HOST=${INFINI_SPAN_PUBLIC_IP:-'127.0.0.1'}
readonly PORT=$(expr ${PORT_BASE} + ${PORT_SHIFT})
readonly CONFIG='/etc/infini-span/infini-span.conf'
readonly PROTOCOL='hotrod'
...
```

En démarrant désormais le service par le biais d'un de ces liens symboliques, plutôt que par l'intermédiaire du script de démarrage directement, on s'assure que les instances n'entrent pas en conflit de port.

3.3.1 Séparation des fichiers de journalisation

Si l'utilisation de liens symboliques permet de démarrer deux (ou plusieurs) instances, de manière locale, toutes ces instances partagent malheureusement le même fichier de journalisation ! Ce qui, en pratique, signifie que seule une de ces instances, la première, sera en mesure d'écrire dans le journal.

Nous allons tout d'abord modifier la configuration du framework de journalisation, log4j, en éditant le fichier **/etc/infini-span/log4j.xml**, pour ajouter une propriété au nom du journal :

```
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/" debug="false">
  <appender name="FILE" class="org.apache.log4j.FileAppender">
    ...
    <param name="File" value="/var/log/infini-span-${instanceId}.log"/>
  </appender>
  ...
</log4j:configuration>
```

Pour valoriser cette propriété lors de l'exécution, il suffit d'ajouter la définition d'une propriété au démarrage de la machine virtuelle Java. Fidèle aux bonnes pratiques dans ce domaine, le script de démarrage du serveur, fourni avec InfiniSpan, a prévu ce genre de manipulation et offre l'utilisation d'une variable d'environnement, **JVM_PARAMS**, pour ajouter ce genre de paramètre au démarrage de la JVM.

Il ne reste donc plus qu'à mettre à jour, un nouvelle fois, notre script :

```
#!/bin/sh
PORT_SHIFT=$(basename "${0}" | sed -e 's/^.*-\([0-9*]\)$/\1/')

if [ -z "${PORT_SHIFT} ]; then
  PORT_SHIFT=0
fi
readonly PORT_BASE=${PORT_BASE:-'11222'}
export JVM_PARAMS="-DinstanceId=${PORT_SHIFT}"

readonly HOST=${INFINI_SPAN_PUBLIC_IP:-'127.0.0.1'}
readonly PORT=$(expr ${PORT_BASE} + ${PORT_SHIFT})
readonly CONFIG='/etc/infini-span/infini-span.conf'
readonly PROTOCOL='hotrod'
...
```

Comparaison avec la mise en place de JBoss Data Grid [JDG]

Comme l'a illustré cette section de l'article, la mise en place d'un serveur InfiniSpan est quelque peu fastidieuse et nécessite la conception d'un script, certes fourni ici mais toujours spécifique. Les équipes gérant cette nouvelle infrastructure devront apprendre à utiliser le serveur fourni par la communauté, sans pouvoir mettre à profit leurs connaissances acquises ailleurs, sur d'autres produits Java ou JEE.

C'est entre autres pour ces raisons que la version supportée par Red Hat de InfiniSpan, JBoss Data Grid [JDG], n'utilise pas ce serveur mais se déploit à partir d'une instance de JBoss AS. La mise en place de ces instances peut donc suivre les usages et les pratiques en vigueur dans une société déjà famillière avec ce produi, et l'utilisation de la grille de données est beaucoup plus aisée aux équipes déjà formées sur JBoss AS.

On peut noter, en outre, que ce mode de déploiement facilite la mise en place de supervision autour des instances de la grille, à l'aide de solutions déjà habituées à communiquer et surveiller des instances de JBoss AS, tel que RHQ **[RHQ]**.

Le script fournit avec le projet est naturellement bien plus adapté à l'usage des projets Open Source, où l'on souhaite pouvoir rapidement tester et « jouer » avec le produit ;)

3.3.2 Configuration des instances

Après ce long, mais nécessaire, travail d'intégration avec le système, nous allons enfin pouvoir étudier de près la configuration même de notre grille et voir comment nous allons, concrètement, implémenter nos caches pour répondre aux deux usages évoqués plus haut.

3.4 Accès aux statistiques JMX

Commençons par mettre en place le paramétrage général de notre grille de données :

```
<infinispan
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="urn:infinispan:config:5.2 http://www.infinispan.org/schemas/
infinispan-config-5.2.xsd"
      xmlns="urn:infinispan:config:5.2">
   <global>
      <globalJmxStatistics
            enabled="true"
            jmxDomain="org.infinispan"
            cacheManagerName="SampleCacheManager"/>
      <transport
          clusterName="infinispan-cluster"/>
  </global>
  ...
</infinispan>
```

La première section du fichier, contenu dans l'élément **`global`**, assure la mise en place des objets JMX nécessaires pour exposer les statistiques aux outils de supervision. La plupart des outils de supervision moderne peuvent facilement recueillir et analyser des données exposées par le protocole JMX.

En effet, des outils aussi élémentaires que la JConsole **[jconsole]**, fourni avec le JDK, ou aussi élaborés que RHQ **[RHQ]**, peuvent exploiter facilement ces données. En outre, et c'est un point non négligeable, les données par JMX peuvent aussi être exploitées au sein de simple script, à l'aide d'un client JMX, ce qui permet d'envisager une intégration dans à peu près n'importe quelle solution de supervision.

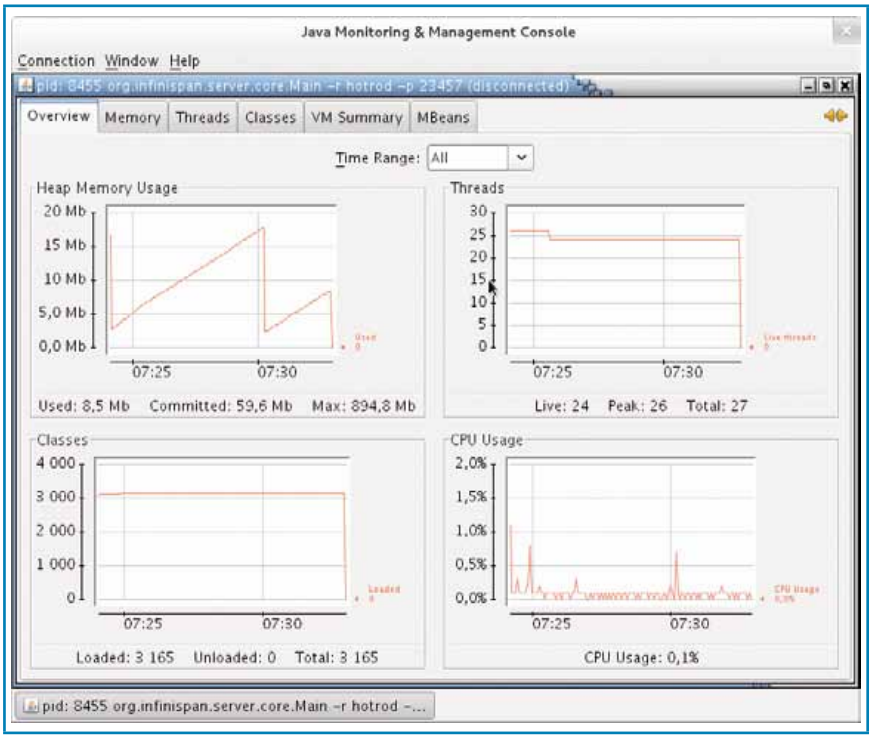


Fig. 1 : Jolokia [Jolokia]

Pour réaliser des scripts utilisant JMX, il est malheureusement nécessaire d'utiliser un client JMX, généralement écrit en Java, ce qui n'est pas très pratique. Il existe fort heureusement une solution très élégante à cette problématique : le projet Jolokia **[Jolokia]**.

Ce projet Open Source permet en effet le déploiement d'un agent, au sein de la machine virtuelle Java exécutant l'instance, qui va exposer les données et méthodes accessibles par JMX à travers une simple API HTTP/Rest. A partir de cette dernière, on peut aisément automatiser des tâches ou effectuer des mesures, à des fins de surveillance, au sein de scripts, sans nécessiter l'ajout ou le développement d'un client JMX.

Le monitoring de service HTTP et l'utilisation d'API Rest étant assez communs et familiers dans le monde de l'administration de systèmes, ce genre d'outil est donc fort appréciable.

Juste en dessous, on définit le nom du cluster utilisé par InfiniSpan. Cette définition assure que cette instance ne communiquera qu'avec des instances placées dans le même cluster, assurant ainsi que d'autres logiciels, tel que JBoss AS, utilisant aussi une technologie similaire, n'interféreront pas avec le fonctionnement de notre grille de données.

3.4.1 Paramétrage par défaut des caches

Avant de réaliser la configuration de nos deux caches, nous allons définir un certain nombre de paramètres par défaut, communs à ces deux caches, dans la section intitulée **`default`** du fichier de configuration :

```
...
  <default>
     <storeAsBinary />
     <jmxStatistics enabled="true"/>
     <clustering mode="distribution"/>
  </default>
...
```

Les trois paramétrages qui s'appliqueront à tout cache disponible dans la grille sont donc :

- **persistance au format binaire** : si la grille doit persister des données qu'elles contient, cette sauvegarde se fera dans un format binaire, qui est généralement plus performant et moins consommateur d'espace disque - ce format interdit l'exploitation directe des données sauvegardées, mais est un comportement par défaut ;
- **activation des statistiques JMX** : on assure que, par défaut, les statistiques associées à un cache sont aussi accessibles par JMX ;
- **distribution** : cette configuration correspond au fonctionnement « naturel » de la grille, où le client détermine lui-même dans quelle instance se place ou se trouve les informations associées à une clé.

On notera ici que tous ces paramètres disposent, bien évidemment, de nombreux attributs et sous-éléments, permettant d'effectuer, si nécessaire, des réglages plus fins. Pour notre démonstration, ces éléments suffissent amplement.

3.5 Les caches

Nous allons donc maintenant enfin définir deux **caches**, qui vont correspondre aux deux scénarios évoqués plus haut, soit :

- le cache **`catalog`**, qui contient une portion du catalogue des produits en vente sur la boutique en ligne de notre compagnie ;
- le cache **`sso_ref`**, qui est destiné à héberger les données d'authentification, qui doivent être retirées de la grille après 30 minutes.

On notera au passage que si l'on parle ici de **`cache`**, c'est pour rester cohérent avec la sémantique du fichier de configuration. Dans les faits, il s'agit juste d'un nom logique, définissant un espace nommé où l'on peut placer des données, qui seront régies par une certaine configuration.

```
...
   <namedCache name="catalog">
     <!--
        - 100000 éléments corresponds à 20% de notre catalogue
        - LRU évicte les éléments les moins utilisés
     -->
     <eviction maxEntries="100000" strategy="LRU"/>
     <!-- 43200000 correspond à une durée de vie de 12h -->
     <expiration lifespan="43200000"/>
   </namedCache>

   <namedCache name="sso_ref">
      <!-- 1800000 correspond à une durée de vie de 30 minutes -->
     <expiration lifespan="1800000"/>
   </namedCache>
...
```

Avec cette dernière touche, la configuration de notre cache est finie, notre nouvel élément d'infrastructure, une grille de données, est donc enfin prêt à l'emploi. Il ne reste plus qu'à adapter le code des applications pour s'en servir.

Conclusion

Cet article aura permis de faire un tour, loin d'être exhaustif, de l'utilisation d'une grille de données comme élément d'infrastructure logicielle, dans le but d'implémenter un cache distribué ou un mécanisme de stockage NoSQL performant. Ces deux scénarios nous ont permis d'évoquer différents aspects et fonctionnalités de ce type de logiciel et la mise en pratique aura permis d'entrevoir les caractéristiques de l'implémentation proposée par le projet InfiniSpan.

Bien évidemment, il ne s'agit là que d'un aperçu - il existe de nombreux autres aspects liés à l'utilisation de ce genre d'outil, qui s'ajoute à la complexité naturelle de leur cas d'utilisation. A titre d'exemple, dans le cas de InfiniSpan, cet article ne couvre pas - à escient, dans le but avoué de rester succinct, la possibilité d'utiliser la grille comme un système de fichier **[GridFileSystem]**, ainsi que ses capacités d'exécution de requêtes **[QueryISPN]** ou encore l'utilisation de Map Reduce **[MapReduce]**.

Comme tout article, il ne s'agit pas, loin de là, d'un ouvrage exhaustif ou même d'un inventaire complet mais simplement d'une vue d'ensemble de ce sujet, riche et passionnant, qui, espérons le, vous permettra de mieux réaliser le potentiel de tels outils et peut être d'en considérer l'utilisation dans vos projets. ■

Références

[TTL] http://fr.wikipedia.org/wiki/Time_to_Live

[LRIS] http://en.wikipedia.org/wiki/LIRS_caching_algorithm

[GridFileSystem] https://docs.jboss.org/author/display/ISPN/Grid+File+System

[QueryISPN] https://docs.jboss.org/author/display/ISPN/Querying+Infinispan

[MapReduce] https://docs.jboss.org/author/display/ISPN/Map+Reduce+framework

METTEZ UN MAINFRAME DANS VOTRE LAPTOP !

par Guillaume « GuiGui2 » Lasmayous [IT Specialist, Linux on System z, IBM Montpellier - Développeur NetBSD à mes heures perdues - Membre du GCU, canal historique]

Levez la main ! Combien d'entre vous ont la moindre idée du type de matériel correspondant à l'architecture s390x, qui fait partie des nouvelles architectures disponibles avec Debian 7.0 'Wheezy' ? Il y a quelques années, je savais à peine épeler s390. Et je n'avais pas la moindre idée de ce à quoi ça ressemblait et ce qu'on pouvait bien faire avec. Depuis 2006, j'ai appris :) Cet article va vous guider dans l'installation de Debian, dans sa version s390x, ce qui est rendu possible grâce à Hercules, un émulateur de l'architecture z.

07 Avril 1964. C'est à cette date que naissait l'IBM System 360, ancêtre des machines que l'on connaît aujourd'hui sous le nom de IBM System z, ou encore « Mainframe ». Depuis 1964, cette machine n'a cessé d'évoluer pour s'adapter aux demandes des applications d'aujourd'hui. L'incarnation actuelle du System z, connue sous le nom de IBM System zEC12 (EC pour Enterprise Class) est une machine qui possède de 1 à 101 processeurs cadencés à 5.5GHz, jusqu'à 3TB de RAIM (Redundant Array of Independant Memory - voyez ça comme du RAID pour la mémoire) et un certain nombre de 'devices' : que ce soit des disques, des cartes réseaux ou cryptographiques, l'architecture actuelle permet de connecter par exemple 320 cartes d'entrée/sortie disques (dites cartes FICON 8S) et 96 cartes réseaux 'OSA Express 4S'.

Je vous renvoie à la spec officielle **[1]** et aux excellents articles Wikipédia sur le sujet **[2][3][4]** pour plus de détails.

1 Terminologie

Le mainframe est un monde a part entière, avec sa communauté, ses groupes d'utilisateurs, ses salons et son jargon. La première fois que vous lirez de la littérature spécialisée mainframe, vous vous demanderez : WTF ? De quoi est-ce qu'il veut bien parler ?

Cet article sera bien trop court pour détailler toute l'architecture du mainframe mais voilà quelques termes qui nous servirons dans la suite. Si vous souhaitez connaître plus de détails sur l'architecture z, vous pouvez consulter le « z/Architecture Principle of Operations » **[5]**.

Sur un mainframe, tous les devices sont identifiés par une adresse, configurée par l'administrateur de la machine, unique, sur 4 digits. Dans l'exemple de l'article, j'ai arbitrairement décidé que les disques seraient à l'adresse 0100 et 0101 et que la carte réseau utiliseraient les adresses 0A00 et 0A01.

- **DASD (Direct Access Storage Device)** : ce sont les unités de disques, aussi connues sous le nom de 3390, correspondant au numéro de série des unités de disques de l'époque. Ils sont aujourd'hui encore référencés de la sorte, suivi par un code à 2 chiffres correspondant à la taille du volume. Ainsi, un 3390-09 (3390 model 09) correspond à un volume d'environ 9G non formaté, soit 6.3G une fois formaté sous Linux. On trouve également le terme ECKD - pour Extended Count Key Data - en référence à ce type de disque.
- **CTC (Channel-To-Channel)** : correspond à une connexion réseau de type point à point. Hercules supporte 2 types de CTC, les CTCI (qui permettent une connexion au host par l'intermédiaire du driver tun) et CTCT (pour interconnecter des émulateurs Hercules entre eux).

Enfin, on parle d'IPL - Initial Program Loading - pour signifier que l'on boote un système d'exploitation.

2 Hercules, System z emulator

Autant l'avouer de suite, accéder à du matériel du type mainframe n'est pas donné à toutes les bourses, même s'il arrive parfois d'en croiser sur eBay. Alors, comment tester ? C'est là qu'entre en jeu le spectaculaire émulateur

Hercules, fruit du travail initié par Roger Bowler en 1999, ensuite rejoint par une communauté qui s'est créée autour du logiciel.

2.1 Préparation

Première chose à faire, installer Hercules. Sur une distribution basée sur Debian, le package s'appelle, sans grandes surprises, hercules, qu'il suffit d'installer à l'aide de la commande :

```
$ sudo apt-get install hercules
```

à adapter bien évidemment à votre distribution.

Notez qu'il existe une interface graphique, du nom de herculesstudio, que je ne détaillerais pas ici.

Afin d'installer Debian sur l'émulateur Hercules, il va également nous falloir une distribution Debian. Thank you Mr Obvious ! En l'occurence, j'ai téléchargé la dernière version - à la date d'écriture de cet article - de la distribution, Debian 7.0/s390x sur le mirroir le plus proche de chez moi.

En réalité, seuls trois fichiers sont vraiment nécessaires pour initier l'installation et lancer debian-installer. Ceci implique l'utilisation d'un lecteur de cartes perforées (virtuelles, évidemment), mais je vais vous faire grâce de cette étape. En pratique, le setup est infiniment plus simple en utilisant l'iso, dont acte. Une fois téléchargée, montez l'iso localement pour pouvoir accéder à son contenu.

```
$ sudo mount -o loop,ro Downloads/debian-7.0.0-s390x-CD-1.iso /mnt
```

Il nous faut également préparer des fichiers qui seront utilisés comme des disques par Hercules. Pour préparer les disques avec une structure logique convenant à un mainframe, Hercules fournit une commande dédiée, dasdinit. On va donc créer 2 volumes, qui seront respectivement utilisés pour le root filesystem et le swap dans notre Debian s390x :

```
$ mkdir hercules
$ dasdinit -lfs -linux dasd.linux.0100 3390-9 LX0100
HHCDU044I Creating 3390 volume LX0100: 10017 cyls, 15 trks/cyl, 56832
bytes/track
HHCDU041I 10017 cylinders successfully written to file dasd.linux.0100
HHCDI001I DASD initialization successfully completed.
$ dasdinit -lfs -linux dasd.linux.0101 3390-9 LX0101
HHCDU044I Creating 3390 volume LX0101: 10017 cyls, 15 trks/cyl, 56832
bytes/track
HHCDU041I 10017 cylinders successfully written to file dasd.linux.0101
HHCDI001I DASD initialization successfully completed.
$ ls -al
total 16678340
drwxrwxr-x  2 guigui guigui       4096 juin   2 17:24 .
drwxr-xr-x 63 guigui guigui      12288 juin   2 16:45 ..
-rw-r-----  1 guigui guigui 8539292672 juin   2 17:24 dasd.linux.0100
-rw-r-----  1 guigui guigui 8539292672 juin   2 17:26 dasd.linux.0101
```

Nous avons donc maintenant 2 volumes pour installer notre Debian, ce qui est plus que largement suffisant. Enfin, on va permettre à notre utilisateur, qui fait partie du groupe **sudo**, de créer le *device* utilisé pour la connexion réseau, comme indiqué dans le fichier **README.Debian** :

```
dpkg-statoverride --add --update root <trusted group> 04750 /usr/bin/
hercifc
```

en remplacant **<trusted group>** par la valeur qui convient à votre distribution.

2.2 Configuration

Un fichier de configuration minimal pour hercules ressemble à celui-là :

```
$ cat hercules/hercules.cnf
CPUSERIAL 000069       # CPU serial number
CPUMODEL  2817
OSTAILOR  LINUX        # OS tailoring
ARCHMODE  z/Arch       # Architecture mode ESA/390 or ESAME
MAINSIZE  1024         # Main storage size in megabytes
NUMCPUS   1 # Number of CPUs

0100 3390 ./dasd.linux.0100
0101 3390 ./dasd.linux.0101
0A00,0A01  CTCI -n /dev/net/tun -t 1500 10.1.1.2 10.1.1.1
```

Explications : Les premières lignes décrivent le type de matériel que nous souhaitons émuler. Dans le cas présent, nous émulerons une machine de type System z196 (dont le CPUMODEL est 2817) dont le CPU a un numéro d'identification 000069. Cette valeur n'a d'ailleurs que peu d'importance pour le cas qui nous intéresse ici, l'installation de Linux. Nous précisons ensuite le type d'architecture : z/Arch ou z/Architecture. Le mainframe a la particularité de supporter différentes architectures, en fonction du type de système d'exploitation que l'on souhaite faire tourner dessus. Hercules prend en compte trois architectures différentes : S/370 (24bits), ESA/390 (31 bits) et z/Arch ou ESAME (64 bits). Le paramètre OSTAILOR a pour effet de limiter les messages qu'affiche Hercules aux seuls messages qui font du sens pour l'OS considéré. Ceci permet de réduire le *bruit* dans la console Hercules. Enfin, le paramètre MAINSIZE précise la taille mémoire de la machine considérée.

La deuxième partie du fichier de configuration comporte la définition de la configuration IO de la machine émulée, dans laquelle nous avons donc :

- deux unités de disques, ou DASD, aux adresses 0100 et 0101. Le 3ème paramètre informe Hercules de la position dans le filesystem des fichiers à utiliser pour émuler ces disques (on peut rapprocher celà de l'utilisation de fichiers 'raw' pour héberger des machines virtuelles Xen ou KVM).

- Enfin, une connexion réseau de type CTCI aux adresses 0A00 et 0A01. Les paramètres précisent le driver utilisé ainsi que les IPs aux deux bouts de la connexion: 10.1.1.1 est l'IP côté mainframe, et 10.1.1.2 est l'IP côté host.

2.3 Installation

Nous sommes prêts à lancer Hercules, de la facon suivante :

```
$ hercules -f hercules.cnf
```

Et sous nos yeux ébahis, un mainframe - émulé - démarre :

```
Hercules Version 3.07
(c)Copyright 1999-2010 by Roger Bowler, Jan Jaeger, and others
Built on Jun 19 2011 at 04:25:40
Build information:
  Debian
  Modes: S/370 ESA/390 z/Arch
  Max CPU Engines: 8
  Using setresuid() for setting privileges
  Dynamic loading support
  Using shared libraries
  HTTP Server support
  Regular Expressions support
  Automatic Operator support
  National Language Support
  Machine dependent assists: cmpxchg1 cmpxchg4 cmpxchg8
Running on x220.guigui2.lan Linux-3.8.0-23-generic.#34-Ubuntu SMP Wed May
29 20:22:58 UTC 2013 x86_64 MP=4
HHCHD018I Loadable module directory is /usr/lib/hercules
Crypto module loaded (c) Copyright Bernard van der Helm, 2003-2010
  Active: Message Security Assist
          Message Security Assist Extension 1
          Message Security Assist Extension 2
HHCCF065I Hercules: tid=7FC99AF8A740, pid=25419, pgid=25419, priority=0
HHCDA020I /home/guigui/hercules/dasd.linux.0100 cyls=10017 heads=15
tracks=150255 trklen=56832
HHCDA020I /home/guigui/hercules/dasd.linux.0101 cyls=10017 heads=15
tracks=150255 trklen=56832
HHCCT073I 0A00: TUN device tun0 opened
HHCCP002I CPU0000 thread started: tid=7FC99853B700, pid=25419, priority=15
HHCTT001W Timer thread set priority -20 failed: Permission denied
HHCTT002I Timer thread started: tid=7FC99843A700, pid=25419, priority=0
HHCCP003I CPU0000 architecture mode z/Arch
HHCPN001I Control panel thread started: tid=7FC99AF8A740, pid=25419
HHCIF005E hercifc: ioctl error doing SIOCDIFADDR on tun0: 25 Inappropriate
ioctl for device
HHCAO001I Hercules Automatic Operator thread started;
          tid=7FC95362F700, pri=0, pid=25419
[...]
Command ==>
```

L'émulateur a bien identifié notre configuration, comme en témoigne les lignes suivantes :

```
HHCDA020I /home/guigui/hercules/dasd.linux.0100 cyls=10017 heads=15
tracks=150255 trklen=56832
HHCDA020I /home/guigui/hercules/dasd.linux.0101 cyls=10017 heads=15
tracks=150255 trklen=56832
HHCCT073I 0A00: TUN device tun0 opened
```

Le message de *plaign* concernant l'ioctl error peut être ignorée consciencieusement, elle ne prête aucunement à conséquence. En admettant que l'image iso ait été montée sur **/mnt** comme dans l'exemple précédent, on peut lancer l'installation de Debian/s390x par la commande suivante, que l'on tapera sur la ligne de commande (identifiée par le prompt **Command ==>**) :

```
Command ==> ipl /mnt/d390.ins
```

Le contenu de ce fichier est le suivant :

```
$ cat /mnt/d390.ins
* Debian GNU/Linux for S/390 (boot from CD-ROM or FTP-Server)
boot/linux_vm 0x00000000
boot/root.off 0x0001040c
boot/root.siz 0x00010414
boot/parmfile 0x00010480
boot/root.bin 0x00800000
```

Il indique les emplacements mémoire où charger les différents programmes nécessaires à l'éxécution du programme d'installation.

Une fois lancé, le kernel va booter et, quelques secondes plus tard, on arrive à la première question du programme d'installation :

```
Configure the network device

Please choose the type of your primary network interface that you
will need for
installing the Debian system (via NFS or HTTP). Only the listed
devices are
supported.
Network device type:
  1: ctc: Channel to Channel (CTC) or ESCON connection,
  2: qeth: OSA-Express in QDIO mode / HiperSockets,
  3: iucv: Inter-User Communication Vehicle - available for VM guests
only,
  4: virtio: KVM VirtIO,
Prompt: '?' for help>
```

Hercules est configuré pour utiliser une connexion réseau de type CTC, il faut donc choisir l'option 1. Attention à bien faire précéder toutes les réponses saisies dans la console d'Hercules d'un **.**, pour que l'émulateur n'intercepte pas la saisie.

```
The following device numbers might belong to CTC or ESCON
connections.
CTC read device:
  1: 0.0.0a00,      2: 0.0.0a01,
Prompt: '?' for help>
```

Ici, on sélectionne l'option 1, qui correspond à la première addresse de notre connexion CTC.

```
The following device numbers might belong to CTC or ESCON connections.
CTC write device:
  1: 0.0.0a00,      2: 0.0.0a01,
Prompt: '?' for help>
```

Ici, on choisit la seconde adresse de notre connexion CTC, soit 0.0.0a01.

```
Protocol for this connection:
  1: S/390 (0)  *!,  2: Linux (1),      3: OS/390 (3),
Prompt: '?' for help, default=1>
```

On choisit le protocole s390, soit l'option 1.

```
Configure a network using static addressing

The IP address is unique to your computer and may be:

* four numbers separated by periods (IPv4);
* blocks of hexadecimal characters separated by colons (IPv6).

You can also optionally append a CIDR netmask (such as "/24").

If you don't know what to use here, consult your network administrator.
IP address:
Prompt: '?' for help>
```

L'IP sera dans mon cas: 10.1.1.2. On saisit donc .10.1.1.2, pour que l'installeur prenne en compte la réponse.

```
The point-to-point address is used to determine the other endpoint of the point
to point network.  Consult your network administrator if you do not know the
value.  The point-to-point address should be entered as four numbers separated
by periods.
Point-to-point address:
Prompt: '?' for help>
```

Ici, on renseigne l'adresse du côté du host: 10.1.1.1. L'installeur demande ensuite les DNS à utiliser, le nom de la machine (debian dans mon cas), ainsi que son nom de domaine (guigui2.lan). Ensuite, il génère une clé SSH qui sera utilisée pour la suite de l'installation. Enfin, d-i nous demande un mot de passe pour la poursuite de l'installation. Note: Soyez patient, la génération de la clé SSH peut prendre un certain temps, voire des fois un peu plus longtemps.

```
Continue installation remotely using SSH

You need to set a password for remote access to the Debian installer. A
malicious or unqualified user with access to the installer can have disastrous
results, so you should take care to choose a password that is not easy to
guess. It should not be a word found in the dictionary, or a word that could be
easily associated with you, like your middle name.

This password is used only by the Debian installer, and will be discarded once
you finish the installation.
Remote installation password:
[...]
Please enter the same remote installation password again to verify that you
have typed it correctly.
Re-enter password to verify:
```

Classiquement, on saisit le mot de passe deux fois, saisie puis vérification. A ce stade on a terminé la partie spécifique *mainframe* de l'installation. La suite se déroulera plus classiquement dans un installeur en mode texte, identique aux autres plate-formes supportées par Debian :

```
Start SSH

To continue the installation, please use an SSH client to connect to the IP
address 10.1.1.2 and log in as the "installer" user. For example:

   ssh installer@10.1.1.2

The fingerprint of this SSH server's host key is:
d2:a5:5c:8e:88:94:54:6d:4e:60:8e:9e:f9:63:75:32

Please check this carefully against the fingerprint reported by your SSH
client.
Press enter to continue!
```

Contrairement à ce que stipule l'installeur, oubliez de presser la touche *Enter*. Avant de lancer une session SSH, on va faire en sorte que la machine en cours d'installation puisse joindre le réseau à l'aide d'une paire de règles IPtables :

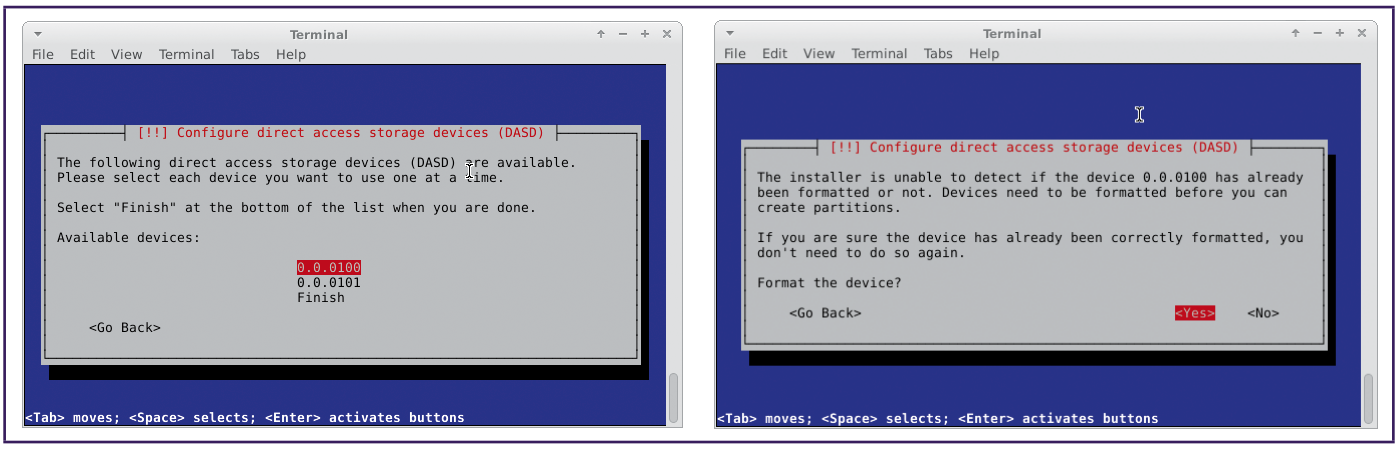


Fig. 1

Fig. 2

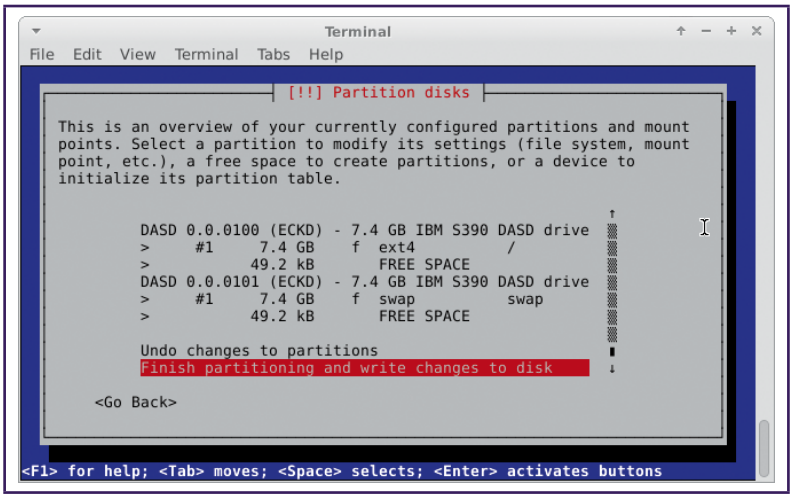


Fig. 3

```
$ sudo iptables -A FORWARD -d 10.1.1.0/24 -j ACCEPT
$ sudo iptables -A FORWARD -s 10.1.1.0/24 -j ACCEPT
$ sudo iptables -t nat -A POSTROUTING -o eth0 -s 10.1.1.0/24 -j
MASQUERADE
$ echo 1 | sudo tee /proc/sys/net/ipv4/ip_forward
```

On pourra alors dégainer son terminal préféré pour se connecter en tant qu'utilisateur installer à l'adresse configurée précédemment :

```
$ ssh installer@10.1.1.2The authenticity of host '10.1.1.2
(10.1.1.2)' can't be established.
RSA key fingerprint is d2:a5:5c:8e:88:94:54:6d:4e:60:8e:9e
:f9:63:75:32.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '10.1.1.2' (RSA) to the list of known
hosts.
installer@10.1.1.2's password:
```

Le mot de passe sera cette fois précisé **sans** le point en début de ligne. Nous voilà maintenant dans un installeur Debian tout ce qu'il y a de plus classique.

On sélectionne la région dans laquelle on se trouve, le pays, le mirroir Debian de son choix. Debian-installer va ensuite télécharger les fichiers nécessaires à la poursuite de l'installation, avant de nous demander de saisir le mot de passe de l'utilisateur root, suivi des informations de création d'un utilisateur non privilégié.

La seule partie spécifique concerne la configuration de l'espace disque. On commence par configurer le volume 0.0.0100, de la manière suivante (voir Figures 1 et 2 page précédente).

Puis recommencer l'opération pour le second disque. Une fois formatés, les disques sont partitionnés classiquement. Dans cet exemple, on utilise le disque 0.0.0100 comme partition root, formaté en ext4 et le disque 101 comme espace de swap (Fig. 3).

Une fois les changements dans la table de partition écrits sur disque, d-i procède au téléchargement et à l'installation du système de base. Après quelques temps, tasksel vous demandera ce que vous souhaitez installer sur votre serveur, puis l'installation se termine. Soyez patients. Il m'a fallu environ 1h00 pour faire l'installation sur un laptop propulsé par un Core i7 @ 3.2GHz. N'oubliez pas qu'Hercules est un émulateur. Tout est fonctionnel, mais généralement bien moins véloce qu'un **vrai** System z.

Après environ 1h15 et 275 millions d'instructions (Hercules affiche le nombre d'instructions exécutées dans le coin inférieur droit) plus tard, notre Debian / s390x est prête.

2.4 Démarrage depuis le disque

Une fois l'installation terminée, il est nécessaire de redémarrer le système depuis le disque. Pour se faire, il suffit de lancer la commande ipl avec l'adresse du disque de boot en paramètre (100 dans le cas décrit dans cet article).

```
Command ==> ipl 100
```

depuis Hercules.

```
ipl 100
HHCCP007I CPU0000 architecture mode set to ESA/390
zIPL v1.16.0-build-20121130 interactive boot menu

0. default (debian)

1. debian
2. old

Please choose (default will boot in 10 seconds):
```

Le menu de démarrage est affiché par le bootloader (zIPL, spécifique à la plate-forme). Patientez une dizaine de secondes et le démarrage s'effectue :

```
HHCCP041I SYSCONS interface active
    0.000000! Initializing cgroup subsys cpuset
    0.000000! Initializing cgroup subsys cpu
    0.000000! Linux version 3.2.0-4-s390x (debian-kernel@lists.debian.org) (gcc
version 4.6.3 (Debian 4.6.3-14) ) #1 SMP Debian 3.2.46-1
    0.000000! setup: Linux is running natively in 64-bit mode
    0.000000! cpu: The CPU configuration topology of the machine is: 0 0 0 0 0
```

On peut ensuite reprendre son client ssh habituel pour se connecter à l'adresse 10.1.1.2 configurée pendant l'installation.

```
guigui@x220 [~] $ ssh 10.1.1.2
guigui@10.1.1.2's password:
Linux debian 3.2.0-4-s390x #1 SMP Debian 3.2.46-1 s390x

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.
```

```
Debian GNU/Linux comes with ABSOLUTELY NO
WARRANTY, to the extent
permitted by applicable law.
Last login: Sun Jun 30 21:42:39 2013 from 10.1.1.1
```

Vous êtes alors aux commandes d'un système Debian tout à fait classique, à ceci prêt que le hardware - émulé - est différent. Toutes les commandes, les outils, sont présents, comme sur une Debian s'éxecutant sur du matériel plus courant. Pour s'enquérir de la configuration spécifique à la plateforme s390x (System z dans la terminologie Linux), on pourra utiliser les commandes fournies par IBM dans le package **s390-tools** :

```
guigui@debian:~$ dpkg -L s390-tools | grep bin/
/usr/bin/cmsfs-fuse
/usr/bin/iucvconn
/usr/sbin/hyptop
/usr/sbin/chreipl
/usr/sbin/vmur
/sbin/lsmem
/sbin/iucvtty
/sbin/tunedasd
/sbin/zgetdump
/sbin/chccwdev
/sbin/lszfcp
/sbin/fdasd
/sbin/dasdinfo
/sbin/chmem
/sbin/tape390_display
/sbin/lschp
/sbin/lsdasd
/sbin/lszcrypt
/sbin/chzcrypt
/sbin/qetharp
/sbin/dasdfmt
/sbin/lstape
/sbin/qethconf
/sbin/lscss
/sbin/zipl
/sbin/vmcp
/sbin/lsqeth
/sbin/dasdview
/usr/sbin/chshut
/usr/sbin/lsshut
/usr/sbin/lsreipl
```

A simple titre d'exemple, on pourra, en tant qu'utilisateur root, lancer la commande **lscss** pour lister les différents *Channel Subsystems* connectés à notre machine. En simplifiant à l'extrême, ces channels subsystems correspondent aux périphériques (disques, cartes réseau, carte crypto) accessibles par notre machine :

```
root@debian:/home/guigui# lscss
Device   Subchan.  DevType CU Type Use  PIM PAM POM  CHPIDs
========================================================================
0.0.0100 0.0.0000  3390/0c 3990/c2 yes  80  80  ff   01000000 00000000
0.0.0101 0.0.0001  3390/0c 3990/c2 yes  80  80  ff   01000000 00000000
0.0.0a00 0.0.0002  3088/01 3088/08 yes  80  80  ff   0a000000 00000000
0.0.0a01 0.0.0003  3088/01 3088/08 yes  80  80  ff   0a000000 00000000
```

La sortie de la commande reflète bien la configuration que nous avons décrite dans le fichier de configuration d'Hercules en début d'article : 2 disques, aux adresses 100 et 101 et une carte réseau correspondant aux devices 0a00 et 0a01. Dernier exemple, la commande **lsdasd** fait la correspondance entre l'adresse du disque et le device node utilisé par **udev** pour identifier ce disque :

```
root@debian:/home/guigui# lsdasd
Bus-ID     Status      Name      Device  Type  BlkSz  Size      Blocks
==============================================================================
0.0.0100   active      dasda     94:0    ECKD  4096   7043MB    1803060
0.0.0101   active      dasdb     94:4    ECKD  4096   7043MB    1803060
```

Ici, notre disque à l'adresse 100 est accessible et accedé sous linux par le device **/dev/dasda**, **dasda1** étant comme attendu la première partition sur ce volume. Pour plus d'informations sur les commandes disponibles dans le package s390-tools, je ne peux que vous conseiller de consulter les pages de man mais également de jeter un oeil (voire deux) sur La documentation Linux on System z, disponible sur le site DeveloperWorks **[6]**. Ce bouquin, le *Devices, Drivers, Features and Commands* est une documentation très précise sur l'ensemble des drivers et outils spécifiques à la plateforme.

Conclusion

Cet article vous aura donné un bref apercu des capacités de l'émulateur Hercules. L'installation de Debian sur un mainframe n'est finalement pas très différente de l'installation sur n'importe quelle autre plateforme. Debian n'est qu'un des nombreux OS qui peut s'exécuter sur ce système. Pour les plus aventureux d'entre vous, vous pouvez essayer d'installer un des systèmes qu'on peut trouver à l'une de ces adresses :

- http://www.ibiblio.org/jmaynard/ (OS/360, VM/370 entre autres).
- http://www.cbttape.org.

Bonnes expérimentations ! ■

Références

[1] http://www-03.ibm.com/systems/z/hardware/zenterprise/zec12_specs.html

[2] https://en.wikipedia.org/wiki/ZEC12

[3] https://en.wikipedia.org/wiki/Z/Architecture

[4] https://en.wikipedia.org/wiki/IBM_mainframe

[5] http://publibfp.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/dz9zr002/CCONTENTS

[6] http://www.ibm.com/developerworks/linux/linux390/documentation_dev.html

CRÉATION D'UN SERVEUR DE DÉMARRAGE PXE SOUS NETBSD, POUR INSTALLER... NETBSD !

par Nils Ratusznik [administrateur système et réseau, membre de l'équipe LinuxFr.org (spécialisé dans les dépêches NetBSD et Red Hat) et squatteur de canal IRC chez NetBSDfr]

Créer un serveur de démarrage PXE est un sujet qui a déjà été abordé de nombreuses fois. Ce qui est moins courant, c'est de démarrer l'installeur NetBSD par le réseau. Et ce qui l'est encore moins, c'est de créer le serveur d'installation avec une machine NetBSD !

1 Pour ceux qui ne suivent pas, au fond

PXE, si vous n'en avez jamais entendu parler, signifie « Preboot Execution Environment ». Wikipédia définit ainsi PXE : « *L'amorçage PXE (sigle de Pre-boot eXecution Environment) permet à une station de travail de démarrer depuis le réseau en récupérant une image de système d'exploitation qui se trouve sur un serveur. L'image ainsi récupérée peut être le système d'exploitation brut ou bien le système d'exploitation personnalisé avec des composantes logicielles (suite bureautique, utilitaires, packs de sécurité, scripts, etc...).* »

Et cela tombe bien, c'est ce que nous voulons obtenir, avec quelques éléments en plus, comme la possibilité de choisir différentes versions de notre OS préféré, que ce soit en terme de version ou d'architecture (i386 ou amd64). Grosso modo, le scénario de démarrage PXE est le suivant :

- la machine démarre sur la carte réseau, laquelle cherche à obtenir une adresse IP et de quoi démarrer ;
- le serveur DHCP lui fournit l'adresse IP, l'adresse IP du serveur où trouver de quoi démarrer, ainsi que le nom du fichier et le protocole de téléchargement (généralement TFTP, HTTP est possible dans le cas de gPXE) ;
- la machine télécharge donc le fichier et l'exécute ;
- le fichier exécuté est dans notre cas un chargeur de démarrage, qui va se procurer sa configuration (en TFTP ou autrement) sur le serveur d'où il a été téléchargé ;
- l'utilisateur fait face à un menu de démarrage, avec un choix par défaut et peut sélectionner une option, qui ira télécharger un noyau, un initrd, un code permettant d'exécuter une image disquette ou ISO contenant un système d'exploitation ;
- selon le cas, l'OS chargé aura besoin de certains composants, situés selon les choix sur un serveur TFTP, FTP, HTTP ou NFS.

2 Prérequis

Cet article est basé sur ma configuration personnelle, qui peut s'avérer quelque peu... surdimensionnée, du moins pour un usage personnel. Je dispose d'un PC à processeur Atom, disposant de trois disques durs, officiant comme serveur TFTP, DHCP et DNS secondaire, miroir local HTTP, partages CIFS et hyperviseur Xen. Le rôle de DHCP et de DNS primaire est tenu par un Soekris. La machine Atom est donc considérée comme le « serveur PXE ». Ces machines fonctionnent, bien entendu, sous NetBSD. Côté clients, je dispose de deux vieux ordinateurs portables (générations Pentium 3 et Athlon XP) et d'une machine virtuelle VirtualBox (avec le pack d'extensions d'Oracle), contenant 1 Go de mémoire vive, accédant au réseau par pont et dont la carte réseau émulée est une Intel Pro 1000 Desktop. Ces machines sont bien entendu configurées pour démarrer sur la carte réseau, que ce soit de manière ponctuelle ou permanente.

Il n'est bien entendu pas nécessaire d'avoir autant de machines différentes, ni de monter une configuration DNS et DHCP redondée. Une infrastructure de démarrage PXE nécessite au minimum une machine avec les services DHCP et TFTP, mais nous allons ajouter dans celle-ci un serveur HTTP. Si dans l'absolu un seul client, physique ou virtuel, devrait suffire pour les tests, je ne saurais trop recommander à ceux qui envisagent un déploiement en production de tester avec

des machines identiques à leur(s) cible(s). Mon expérience, qui s'est étendue au-delà de NetBSD, m'a montré que d'une carte réseau à l'autre, d'une génération de machine à l'autre et d'un système à l'autre, les résultats peuvent être différents. Si vous souhaitez reproduire cette architecture avec une solution de virtualisation, je vous recommande donc de faire attention à la carte réseau émulée et d'essayer de la modifier si votre machine virtuelle ne démarre pas sur le réseau (après vous être assuré que le reste de la configuration est correct).

Les logiciels à installer seront signalés au fur et à mesure. Pour éviter que cet article soit démesurément long, je pars du principe que le lecteur sait installer des logiciels sous NetBSD en utilisant **pkg_add**, **pkgin** ou la compilation de packages via **pkgsrc**. L'installation du système n'est pas non plus abordée, car celle réalisée pour le serveur est tout ce qu'il y a de plus classique. De nombreux articles ont déjà traité ces sujets et de fort belle manière !

Enfin, le lecteur attentif remarquera que ce document met en place les briques dans le sens inverse de leur utilisation (détaillée au paragraphe précédent). C'est totalement volontaire et permet de tester facilement les étapes de la mise en place et de « casser » votre configuration DHCP au dernier moment par exemple. Autre avantage : si vous voulez juste faire une installation en réseau sans démarrage réseau, il suffit de s'arrêter à la fin de la partie « Arborescence netinstall et serveur HTTP ».

3 Miroir, miroir, dis-moi qui est le plus beau

Côté arborescence, j'ai choisi de mettre l'ensemble des fichiers publiquement accessibles dans le répertoire **/srv/www** de mon serveur. Ce répertoire sera à la fois la racine du serveur TFTP et la racine du serveur HTTP. Un sous-répertoire **pub** va contenir les nombreuses données et outils utilisés non seulement pour démarrer l'installeur NetBSD, mais aussi pour accueillir les sets ainsi que d'autres systèmes à démarrer et à installer par le réseau. Nous allons aussi prévoir l'arrivée de futurs autres OS, en créant un répertoire dédié à NetBSD (chaque version sera dans un sous-répertoire) :

```
root@arreat:~# mkdir -p /srv/www/pub/NetBSD
```

Reste à peupler ce répertoire. NetBSD est connu pour sa portabilité, il est donc possible de télécharger des sets, images ISO et installeurs pour de nombreuses architectures mais seules deux nous intéressent ici : i386 et amd64, respectivement nos PC classiques, en 32 et 64 bits. Il est possible de télécharger les images iso pour i386 et amd64, de les monter en loopback et de copier leur contenu dans deux répertoire amd64 et i386. Mais ce n'est pas drôle. Nous allons synchroniser sur un serveur **rsync** ces parties et utiliser pour cela **net/rsync** :

```
root@arreat:~# mkdir -p /srv/www/pub/scripts
root@arreat:~# vi /srv/www/pub/scripts/rsync_netbsd.sh
/usr/pkg/bin/rsync \
        -avzPH --stats \
        --delete --delete-after \
        --include="/NetBSD-[5-6].?/" \
        --include="/NetBSD-[5-6].?/amd64/" \
        --include="/NetBSD-[5-6].?/amd64/**" \
        --include="/NetBSD-[5-6].?/i386/" \
        --include="/NetBSD-[5-6].?/i386/**" \
        --include="/NetBSD-[5-6].?/shared/" \
        --include="/NetBSD-[5-6].?/shared/ALL/" \
        --include="/NetBSD-[5-6].?/shared/ALL/**" \
        --include="/NetBSD-[5-6].?/source/" \
        --include="/NetBSD-[5-6].?/source/**" \
        --include="/NetBSD-[5-6].?.?/" \
        --include="/NetBSD-[5-6].?.?/amd64/" \
        --include="/NetBSD-[5-6].?.?/amd64/**" \
        --include="/NetBSD-[5-6].?.?/i386/" \
        --include="/NetBSD-[5-6].?.?/i386/**" \
        --include="/NetBSD-[5-6].?.?/shared/" \
        --include="/NetBSD-[5-6].?.?/shared/ALL/" \
        --include="/NetBSD-[5-6].?.?/shared/ALL/**" \
        --include="/NetBSD-[5-6].?.?/source/" \
        --include="/NetBSD-[5-6].?.?/source/**" \
        --include="/iso/" \
        --include="/iso/[5-6].?/" \
        --include="/iso/[5-6].?/*amd64*" \
        --include="/iso/[5-6].?/*i386*" \
        --include="/iso/[5-6].?.?/" \
        --include="/iso/[5-6].?.?/*amd64*" \
        --include="/iso/[5-6].?.?/*i386*" \
        --exclude=* \
        rsync://rsync.fr.NetBSD.org/NetBSD/ /srv/www/pub/NetBSD/ 2>&1
| tee -a /var/log/sync_netbsd.log
root@arreat:~# /srv/www/pub/scripts/rsync_netbsd.sh
# (s'en suivent de nombreuses minutes d'attente...)
```

Ce script récupère toutes les versions 5 et 6 de NetBSD et permet d'éviter de télécharger les versions plus anciennes. Bien sûr, il peut facilement être adapté lors de la sortie d'une future version 7. Mais nous n'en sommes pas encore là...

A titre d'indication, et après plusieurs versions de NetBSD passées, voici un aperçu de mes répertoires (ne tenez pas compte de **NetBSD-5.1.0_PATCH**, il s'agit d'une version recompilée par mes soins) :

```
root@arreat:~# ls -hl /srv/www/pub/NetBSD/
total 7,0K
drwxr-xr-x   6 99    wheel   512B Sep 18  2012 NetBSD-5.0/
drwxr-xr-x   6 99    wheel   512B Jul 30  2009 NetBSD-5.0.1/
drwxr-xr-x   6 99    wheel   512B Sep 18  2012 NetBSD-5.0.2/
drwxr-xr-x   7 99    wheel   512B Sep 18  2012 NetBSD-5.1/
drwxr-xr-x   4 root  wheel   512B Mar 22  2011 NetBSD-5.1.0_PATCH/
drwxr-xr-x   6 99    wheel   512B Jan  6  2012 NetBSD-5.1.1/
drwxr-xr-x   7 99    wheel   512B Sep 18  2012 NetBSD-5.1.2/
drwxr-xr-x   6 99    wheel   512B Nov 29  2012 NetBSD-5.2/
drwxr-xr-x   6 99    wheel   512B Oct 15  2012 NetBSD-6.0/
drwxr-xr-x   6 99    wheel   512B Dec 29  2012 NetBSD-6.0.1/
drwxr-xr-x   6 99    wheel   512B May 16 08:26 NetBSD-6.0.2/
drwxr-xr-x   6 99    wheel   512B May 16 02:54 NetBSD-6.1/
drwxr-xr-x   6 99    wheel   512B Aug 25 13:29 NetBSD-6.1.1/
drwxr-xr-x  15 root  daemon  512B Aug 25 09:19 iso/
root@arreat:~# ls -hl /srv/www/pub/NetBSD/NetBSD-6.1
```

```
total 2,0K
drwxr-xr-x  4 99  wheel  512B May 16 02:04 amd64/
drwxr-xr-x  4 99  wheel  512B May 16 00:02 i386/
drwxr-xr-x  3 99  wheel  512B May 15 22:24 shared/
drwxr-xr-x  3 99  wheel  512B May 15 22:24 source/
```

Passons à présent à la configuration du serveur web. Comme je suis le seul à me servir de ce système d'installation dans mon réseau local et que je souhaite une configuration simple ne nécessitant pas 421337 packages à installer et à mettre à jour, j'ai choisi d'utiliser celui qui est inclus dans le système de base : **bozohttpd**. La configuration de ce dernier est on ne peut plus simple, en voici pour preuve la liste des paramètres disponibles :

```
root@arreat:~# grep httpd /etc/defaults/rc.conf
httpd=NO                httpd_flags=""
                        httpd_wwwdir="/var/www"
                        httpd_wwwuser="_httpd"
```

Je me suis donc contenté d'ajouter les lignes suivantes dans mon **/etc/rc.conf** :

```
# bozohttpd
httpd=YES
httpd_flags="-X -S 'AHP Intranet'"
httpd_wwwdir="/srv/www"
```

L'option **-X** active le listage des fichiers lorsqu'aucun fichier **index.html** n'est présent et l'option **-S** permet de changer la signature du serveur web. Cette dernière option est totalement inutile dans le cadre de notre objectif mais reste amusante à connaître. Le démarrage, l'arrêt et la relance de **bozohttpd** s'effectuent via un classique **/etc/rc.d/httpd (start|stop|restart)**.

A la fin de cette première étape, nous disposons d'un serveur HTTP capable de servir les fichiers d'une installation de NetBSD. Je vous encourage à tester ce premier point avec une machine virtuelle, en démarrant sur le fichier **boot.iso** que vous trouverez sur le serveur dans **/pub/NetBSD-6.1/i386/installation/cdrom/** (remplacer **i386** par **amd64** pour tester la version 64 bits). Il nous faut encore installer ce qui va nous permettre de démarrer par le réseau.

4 Pxelinux et pxeboot sont dans un bateau...

De mon expérience, 99% des documentations sur le démarrage en réseau utilisent l'outil **pxelinux**, présent dans la suite **syslinux**. Le pourcentage restant utilise GRUB. Mais ça, c'était sans compter NetBSD, qui dispose de son propre chargeur de démarrage : **pxeboot**. Si on souhaite créer un serveur de démarrage PXE uniquement pour NetBSD, ce dernier est amplement suffisant. Si, comme moi, vous souhaitez en plus démarrer des installeurs d'autres systèmes, il faut chaîner **pxelinux** et **pxeboot**. Voyons dans un premier temps la configuration de **pxeboot**, puis nous ajouterons **pxelinux**.

Comme le serveur PXE est sous NetBSD, **pxeboot** est déjà disponible : il s'agit du fichier **/usr/mdec/pxeboot_ia32.bin**. Si ce n'est pas le cas mais que l'arborescence détaillée plus haut est déjà synchronisée, vous le trouverez là : **/srv/www/pub/NetBSD/NetBSD-6.1/i386/installation/misc/pxeboot_ia32.bin** (disponible aussi pour **amd64** et pour les autres versions de NetBSD). Par défaut, ce fichier ne permet pas grand-chose, car il se contente de télécharger sur un partage NFS un fichier nommé **netbsd** désignant un noyau, le plus souvent d'installation. Cependant, il est possible d'étendre ses possibilités via l'outil **installboot**, fourni aussi en standard dans NetBSD. La possibilité qui nous intéresse le plus est celle de faire lire à **pxeboot** un fichier de configuration de type **boot.cfg**. Il sera donc possible de choisir via un menu le noyau d'installation souhaité.

Commençons par copier le fichier à ce qui sera la racine du serveur TFTP :

```
root@arreat:~# cp -vp /usr/mdec/pxeboot_ia32.bin /srv/www/
```

Ensuite, on active les options adéquates :

```
root@arreat:~# installboot -v -e -o bootconf,modules /srv/www/pxeboot_ia32.bin
```

Attention : **installboot** est un petit malin, malgré l'option **-v** (verbose), il n'est pas très bavard. Si une option est déjà active, il va la supprimer. Ne lancez donc pas deux fois de suite la commande et si vous avez un doute, repartez du fichier d'origine. Enfin, on rédige un fichier **boot.cfg** (la liste des noyaux n'est pas exhaustive puisque nous avons synchronisé toutes les versions 5 et 6) :

```
root@arreat:~# vi /srv/www/boot.cfg
banner=Welcome to the NetBSD PXE Installer
banner=========================================================================
banner=
banner=This menu contains only the installation kernels.  Binary sets to complete the
banner=installation must be downloaded separately.  The installer can download them
banner=if this machine has a working internet connection.
banner=
banner=ACPI (Advanced Configuration and Power Interface) should work on all modern
banner=and legacy hardware.  However if you do encounter a problem while booting,
banner=try disabling it and report a bug at http://www.NetBSD.org/.
menu=Drop to boot prompt:prompt
menu=Install NetBSD 5.2 amd64:boot tftp:pub/NetBSD/NetBSD-5.2/amd64/binary/kernel/netbsd-INSTALL.gz
menu=Install NetBSD 5.2 i386:boot tftp:pub/NetBSD/NetBSD-5.2/i386/binary/kernel/netbsd-INSTALL_FLOPPY.gz
menu=Install NetBSD 6.0.2 amd64:boot tftp:pub/NetBSD/NetBSD-6.0.2/amd64/binary/kernel/netbsd-INSTALL.gz
menu=Install NetBSD 6.0.2 i386:boot tftp:pub/NetBSD/NetBSD-6.0.2/i386/binary/kernel/netbsd-INSTALL.gz
menu=Install NetBSD 6.1 amd64:boot tftp:pub/NetBSD/NetBSD-6.1/amd64/binary/kernel/netbsd-INSTALL.gz
menu=Install NetBSD 6.1 i386:boot tftp:pub/NetBSD/NetBSD-6.1/i386/binary/kernel/netbsd-INSTALL.gz
menu=Install NetBSD 6.1.1 amd64:boot tftp:pub/NetBSD/NetBSD-6.1.1/amd64/binary/kernel/netbsd-INSTALL.gz
menu=Install NetBSD 6.1.1 i386:boot tftp:pub/NetBSD/NetBSD-6.1.1/i386/binary/kernel/netbsd-INSTALL.gz
default=1
timeout=10
clear=1
```

Si le système PXE ne doit servir que des installations de NetBSD, on peut s'arrêter là et passer à la partie suivante (configuration du serveur TFTP). Vérifions au moins que les fichiers **pxeboot_ia32.bin** et **boot.cfg** peuvent être téléchargés en HTTP, ils sont disponibles à la racine du serveur web.

Pour pouvoir avoir plus de choix que NetBSD, continuons avec **pxelinux**. Nous allons continuer dans l'arborescence que nous connaissons déjà mais, une fois n'est pas coutume, nous n'allons pas installer **pxelinux** via les paquets de la distribution mais directement depuis le site du noyau Linux avec **net/wget**. On notera que le téléchargement n'est possible qu'en HTTPS, d'où l'utilisation de l'option **–no-check-certificate**

```
root@arreat:~# cd /srv/www/pub
root@arreat:/srv/www/pub# wget --no-check-certificate https://www.
kernel.org/pub/linux/utils/boot/syslinux/syslinux-6.01.tar.gz
root@arreat:/srv/www/pub# tar -xvzpf syslinux-6.01.tar.gz
root@arreat:/srv/www/pub# ln -sv syslinux-6.01 syslinux
```

De la sorte, il sera possible de passer simplement à une version supérieure de syslinux et de revenir en arrière si besoin. Comme vu plus tôt, il est prévu que la racine du serveur TFTP soit **/srv/www** et non **/srv/www/pub**. Pour rendre accessible les vrais fichiers de démarrage, remontons d'un niveau et effectuons nos liens symboliques :

```
root@arreat:~# cd /srv/www/
root@arreat:/srv/www# for i in $(find /srv/www/pub/syslinux/bios/
-type f -iname *.c32); do ln -sv $i; done
root@arreat:/srv/www# for i in $(find /srv/www/pub/syslinux/bios/
-type f -iname *.0); do ln -sv $i; done
```

Les fichiers **.0** servent à démarrer sur le réseau, tandis que les fichiers en **.c32** sont des modules appelés par la configuration (**chain.c32** permet de chaîner un démarrage, **menu.c32** permet de créer un menu d'affichage perfectionné). Lorsque nous configurerons le serveur DHCP, nous indiquerons au client d'aller télécharger le fichier **pxelinux.0**. Les autres fichiers seront nécessaires selon la configuration que nous appliquerons au niveau de notre menu de démarrage. Le fichier **gpxelinux.0** fournit les mêmes fonctionnalités que **pxelinux.0** mais permet entre autres de démarrer sur HTTP au lieu de TFTP, pour télécharger des fichiers de taille assez importante (au hasard l'initrd obèse des Fedora depuis la 14 ou la 15). Le prix à payer s'avère être une compatibilité moins étendue avec les cartes réseau. Comme nous n'en avons pas un besoin immédiat, nous allons rester sur **pxelinux.0**.

Une fois **pxelinux** installé, il nous reste à le configurer. C'est là que réside une grande partie de l'intelligence du système, puisque c'est dans cette configuration que nous créons notre « menu de démarrage réseau », où il sera possible de choisir de démarrer le disque dur local, de démarrer un système, ou de chaîner vers **pxeboot** pour installer NetBSD. Rendons à César ce qui est à Jules, ce fichier est fortement inspiré du fichier de configuration ISOLINUX de la distribution Linux « System Rescue CD », que je vous recommande au passage.

```
root@arreat:~# mkdir /srv/www/pxelinux.cfg
root@arreat:~# vi /srv/www/pxelinux.cfg/default
UI menu.c32

PROMPT 0
TIMEOUT 90
MENU AUTOBOOT Starting default choice in # seconds
ONTIMEOUT local1
MENU TABMSG Press <TAB> to edit options
MENU TITLE Another Home Page PXE Menu
MENU ROWS 16
MENU TIMEOUTROW 22
MENU TABMSGROW 24
MENU CMDLINEROW 24
MENU HELPMSGROW 26
MENU WIDTH 78
MENU MARGIN 6

MENU color title    1;31;40    #FFFF0000 #00000000 std
MENU color sel      7;37;40    #FF000000 #FFC0C0C0 all
MENU color unsel    37;44      #FF000000 #00000000 none
MENU color hotsel   1;7;37;40  #FF000000 #FFC0C0C0 all
MENU color tabmsg   1;31;40    #FFFFFF00 #00000000 std
MENU color help     1;31;40    #FFFFFFFF #00000000 none

LABEL local1
        MENU LABEL *) Boot from first hard disk
        kernel chain.c32
        append hd0
        TEXT HELP
        Boot local OS installed on first hard disk
        ENDTEXT

LABEL local2
        MENU LABEL *) Boot from second hard disk
        kernel chain.c32
        append hd1
        TEXT HELP
        Boot local OS installed on second hard disk
        ENDTEXT

MENU SEPARATOR
MENU BEGIN
MENU TITLE A) INSTALL AN OPERATING SYSTEM

# inserer ici d'autres OS

LABEL nbpxe
        MENU LABEL NetBSD PXE installer
        KERNEL pxeboot_ia32.bin

MENU SEPARATOR
LABEL return
  MENU LABEL Return to main menu
```

```
  MENU EXIT
MENU END
MENU BEGIN
MENU TITLE B) BOOT SPECIAL TOOLS
# inserer ici certains outils speciaux
MENU SEPARATOR
LABEL return
  MENU LABEL Return to main menu
  MENU EXIT
MENU END
MENU BEGIN
MENU TITLE C) BOOT SOME LIVE SYSTEMS
# inserer ici des OS complets live
MENU SEPARATOR
LABEL return
  MENU LABEL Return to main menu
  MENU EXIT
MENU END
```

Les arborescences de **pxelinux** et de **pxeboot** sont publiques, vous pouvez donc facilement vérifier que les chargeurs de démarrage **pxelinux.0** et **pxeboot_ia32.bin** peuvent être téléchargés, ainsi que les fichiers de configuration (situés sur le serveur HTTP dans **/pxelinux.cfg/default** et dans **/boot.cfg**). Mais il nous faut aussi les télécharger en TFTP...

5 TFTP

NetBSD, en plus de comprendre un serveur HTTP dans son système de base, contient aussi un serveur TFTP. Pour être actif, celui-ci a besoin du super-serveur **inetd**. Configurons donc **tftpd** en éditant le fichier **/etc/inetd.conf** (on notera que **tftpd** n'écoute que sur ipv4) :

```
tftp            dgram   udp     wait.600        root    /usr/libexec/
tftpd      tftpd -d -l -s /srv/www
```

Il nous faut aussi démarrer **inetd** par défaut, mais nous avons de la chance, ceci est normalement le cas :

```
root@arreat:~# grep inetd /etc/defaults/rc.conf
# inetd is used to start the IP-based services enabled in /etc/inetd.
conf
inetd=YES               inetd_flags="-l"        # -l logs libwrap
```

Si ce n'est pas le cas, il faudra ajouter au fichier **/etc/rc.conf** la ligne suivante :

```
inetd=YES               inetd_flags="-l"
```

Et dans tous les cas, nous redémarrons Inetd pour la prise en compte de l'ajout de **tftpd** :

```
root@arreat:~# /etc/rc.d/inetd restart
Stopping inetd.
Starting inetd.
```

Continuons notre bonne habitude et vérifions que notre serveur TFTP fonctionne. Pour cela, il nous faut un client. On peut utiliser une machine distante, ou notre serveur lui-même, qui en possède justement un :

```
root@arreat:~# which tftp
/usr/bin/tftp
```

Déplaçons-nous là où ça ne gênera pas :

```
root@arreat:~# mkdir /tmp/tftptest && cd /tmp/tftptest
root@arreat:/tmp/tftptest#
```

Et testons :

```
nils@arreat:~$ mkdir /tmp/tftptest
nils@arreat:~$ cd /tmp/tftptest/
nils@arreat:/tmp/tftptest$ tftp
tftp> connect <ip du serveur>
tftp> binary
tftp> get pxelinux.0
Received 42534 bytes in 0.1 seconds
tftp> get gpxelinux.0
Received 106660 bytes in 0.2 seconds
tftp> get menu.c32
Received 25708 bytes in 0.1 seconds
tftp> get pxeboot_ia32.bin
Received 53444 bytes in 0.1 seconds
tftp> get pub/NetBSD/NetBSD-6.1.1/amd64/INSTALL.txt
Received 97413 bytes in 0.2 seconds
tftp> quit
nils@arreat:/tmp/tftptest$ ll
total 318K
-rw-r--r--  1 nils  wheel   95K Sep 24 16:37 INSTALL.txt
-rw-r--r--  1 nils  wheel  104K Sep 24 16:36 gpxelinux.0
-rw-r--r--  1 nils  wheel   25K Sep 24 16:36 menu.c32
-rw-r--r--  1 nils  wheel   52K Sep 24 16:36 pxeboot_ia32.bin
-rw-r--r--  1 nils  wheel   42K Sep 24 16:36 pxelinux.0
```

Le test est donc concluant, tous nos fichiers sont arrivés :)

6 Configuration du serveur DHCP

Attaquons-nous maintenant à la dernière brique de notre infrastructure, qui paradoxalement est appelée en premier lors de son utilisation : le serveur DHCP. Tout comme les serveurs HTTP et TFTP, NetBSD dispose d'un serveur DHCP dans son système de base, il s'agit du célèbre ISC DHCPD. Le principe est le suivant : lorsqu'une machine demande une adresse IP (v4) sur le réseau, le serveur DHCP va lui répondre en lui donnant de nombreuses informations, dont :

- son adresse IP ;
- le masque de sous-réseau ;
- l'adresse de la passerelle par défaut ;
- l'adresse d'un ou de plusieurs serveurs DNS ;
- l'adresse du **next-server**, ce sera ici notre serveur PXE ;
- un fichier à télécharger, ici le fichier **pxelinux.0**.

A noter que le réseau en exemple ici a les caractéristiques suivantes :

- adressage 192.168.6.0/24 ;
- paserelle par défaut 192.168.6.254 ;
- serveurs DNS 192.168.6.5 et 192.168.6.60, domaine local anotherhomepage.loc ;
- les adresses IP DHCP allouées sont entre 192.168.6.200 et 192.168.6.249 ;
- serveur PXE 192.168.6.60.

Nous allons de plus ajouter un petit hack qui va nous éviter de mettre en place un serveur NFS : en effet, **pxeboot** a pour comportement habituel d'aller chercher son fichier **boot.cfg** sur un partage NFS. Pour éviter de mettre en place un tel partage, paramétrons le serveur DHCP pour qu'il indique à **pxeboot** que le fichier est disponible en TFTP.

On remarquera dans le fichier de configuration ci-après deux exemples de machines disposant d'une adresse IP fixe allouée par DHCPD ainsi que la présence en commentaire du fichier **gpxelinux.0**, prêt à remplacer **pxelinux.0**.

```
root@arreat:~# vi /etc/dhcpd.conf
ddns-domainname "anotherhomepage.loc";
ddns-update-style none;
ddns-updates off;
ignore client-updates;
allow booting;
allow bootp;
authoritative;
allow unknown-clients;
# Duree de vie du bail en secondes
max-lease-time 3600;
default-lease-time 1800;
# Sous-reseau interne
subnet 192.168.6.0 netmask 255.255.255.0 {
        pool {
        deny dynamic bootp clients;
        range 192.168.6.200 192.168.6.249;
        option domain-name-servers 192.168.6.5, 192.168.6.60;
        option domain-name "anotherhomepage.loc";
        option routers 192.168.6.254;
        option broadcast-address 192.168.6.255;
        next-server 192.168.6.60;
        option root-path "/srv/www";
        filename "pxelinux.0";
        # Pour expérimenter gPXE
        #filename "gpxelinux.0";
        # Pour ne démarrer que l'installeur NetBSD
        #filename "pxeboot_ia32.bin" ;
        # Si pxeboot nous demande quelque chose, c'est boot.cfg
        # et c'est en TFTP qu'il faut le trouver
        if substring (option vendor-class-identifier, 0, 17) =
"NetBSD:i386:libsa" {
                if filename = "boot.cfg" {
                        filename "tftp:boot.cfg";
                }
        }
        }
group {
use-host-decl-names true ;
host pxelaptop {
        hardware ethernet 00:01:02:03:04:f7;
        fixed-address 192.168.6.198;
        option host-name "pxelaptop";
}
# Virtual Machine
host pxemachine {
        hardware ethernet 08:00:27:d3:8f:2d;
        fixed-address 192.168.6.199;
        option host-name "pxemachine";
}
}
```

ISC DHCPD ne démarre pas automatiquement avec le démarrage de notre serveur :

```
root@arreat:~# grep dhcpd /etc/defaults/rc.conf
dhcpd=NO                dhcpd_flags="-q"
```

Autorisons-le en ajoutant à **/etc/rc.conf** :

```
dhcpd=YES
```

Et enfin, démarrons notre serveur :

```
root@arreat:~# /etc/rc.d/dhcpd start
Starting dhcpd.
```

Par défaut, DHCPD écrit dans **/var/log/messages** :

```
root@arreat:~# tail -f /var/log/messages
Jul 10 16:43:06 arreat dhcpd: Wrote 0 deleted host decls to leases
file.
Jul 10 16:43:06 arreat dhcpd: Wrote 0 new dynamic host decls to
leases file.
Jul 10 16:43:06 arreat dhcpd: Wrote 48 leases to leases file.
Jul 10 16:43:06 arreat dhcpd: pool 7f7ff7ba5150 192.168.6/24 total 50
free 26  backup 24  lts 1
```

7 Test de bout en bout

Il est maintenant temps de tester le bon fonctionnement de notre infrastructure, du début à la fin. Dans le cas d'une machine physique, il faut se rendre dans le BIOS et activer le démarrage par le réseau. Celui-ci peut prendre plusieurs formes, comme par exemple : activer le démarrage de la carte réseau,

Fig. 1 : Exemple de sélection au démarrage via la carte mère, sans passer par tous les écrans du BIOS

puis modifier l'ordre des périphériques de démarrage pour amener la carte réseau avant le lecteur optique ou le disque dur. D'autres machines disposent d'un moyen de démarrer sur le réseau ou de changer l'ordre de démarrage via une touche du clavier, généralement l'une des touches de fonction (F1 à F12).

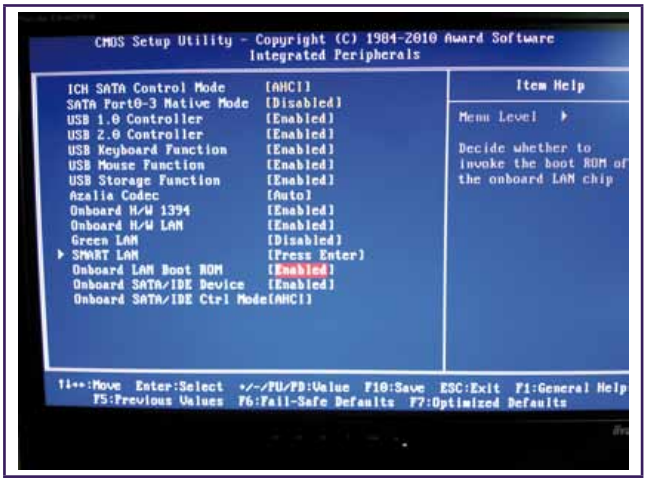


Fig. 2 : Exemple d'activation dans un BIOS de la capacité de démarrage de carte réseau : cela lui permet d'être sélectionnée comme périphérique amorçable

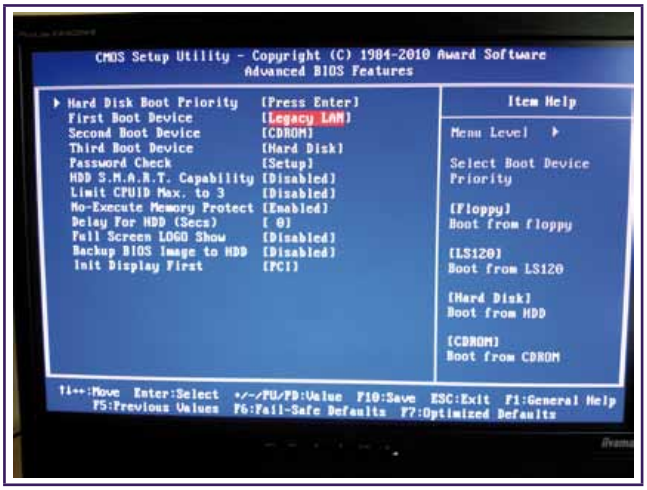


Fig. 3 : Exemple de paramétrage permanent d'ordre de démarrage au niveau du BIOS

Dans le cas d'une machine virtuelle, cela dépend de la solution de virtualisation utilisée. Dans le cas précis d'Oracle VirtualBox, il est impératif d'installer les extensions (attention à leur licence !), qui activent justement le démarrage PXE. Ensuite, le principe est presque le même qu'une machine physique, à la différence que cela se fait dans la configuration de la machine virtuelle : activer le démarrage de la carte réseau, puis, au choix, le déplacer avant la disquette, le CD et le disque dur, ou appuyer sur F12 au démarrage (Fig. 4).

Le menu PXELinux devrait alors apparaître. On sélectionne alors avec les touches du clavier l'entrée du menu NetBSD et

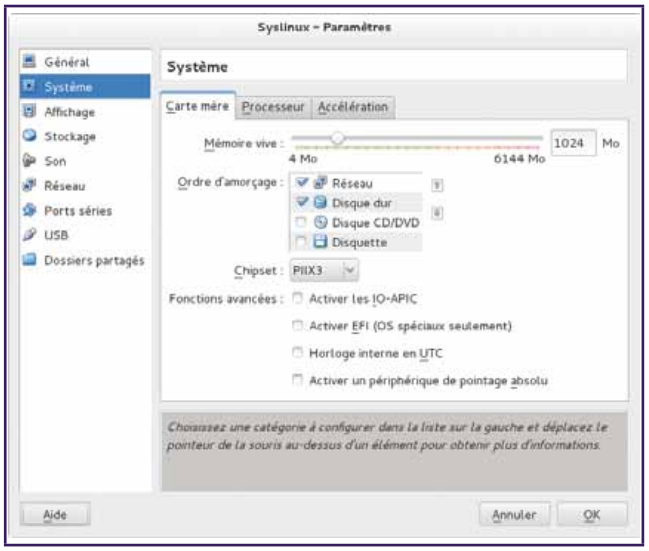


Fig. 4 : Ordre d'amorçage dans Oracle VirtualBox

le menu d'installation de NetBSD s'affiche alors. Attention, celui-ci n'est pas aussi intuitif que PXELinux et le clavier est paramétré en QWERTY.

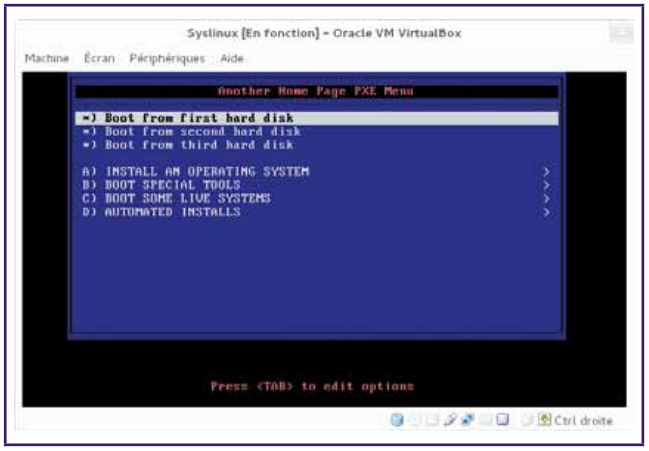


Fig. 5 :Premier menu pxelinux

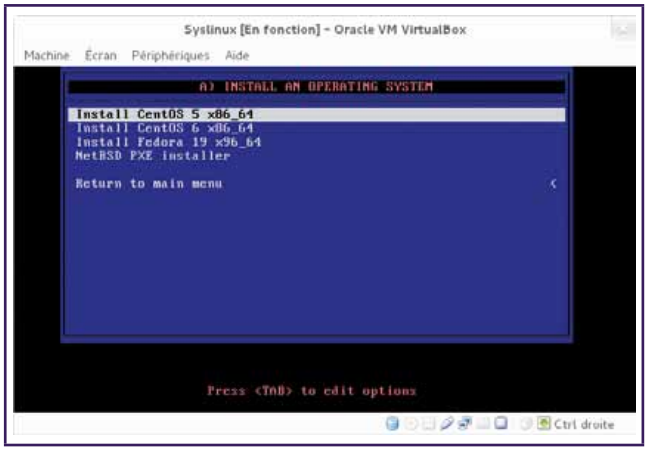


Fig. 6 : Deuxième menu pxelinux

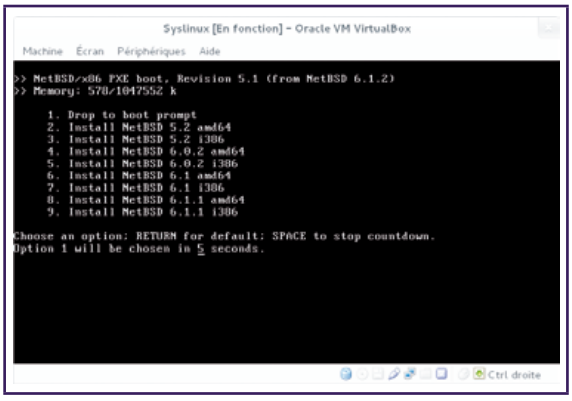


Fig. 7 : Menu pxeboot NetBSD

8 Pimp my PXE... ou pas

Notre menu est fonctionnel et à peu près pratique, mais ce n'est pas le plus beau de la terre. Il est possible de modifier les couleurs et de paramétrer un fond d'écran... ou pas, en fait. Syslinux est parfaitement capable de gérer une image de fond, c'est même grâce à cela que les CD de démarrage de nos distributions Linux favorites fonctionnent. Mais l'utilisation du menu avancé vesamenu.c32 empêchait pxeboot de fonctionner. Il m'aura fallu plusieurs heures et un appel à l'aide sur la liste de diffusion *netbsd-users* avant d'arriver à ce constat. Qui est en cause ? Syslinux ? Pxeboot ? Pour l'instant, je ne sais pas.

9 Quand ça veut pas...

C'est le grand moment. Vous démarrez votre machine, et... rien. Pas de démarrage sur le réseau, pas d'installeur NetBSD en vue, éventuellement la machine démarre sur son disque local et amorce l'OS que vous auriez installé. Sans être un guide exhaustif des erreurs possibles, cette partie aborde celles qui risquent d'arriver. Il se peut aussi que votre problème soit insoluble avec la combinaison BIOS/carte réseau dont vous disposez, la qualité de l'implémentation PXE est variable d'un constructeur à l'autre (voir tableau ci-dessous)...

10 Le mot de la fin

La configuration telle que je vous la livre ne s'est pas inventée en quelques heures. Elle est le résultat de près de 3 ans de vie du système et de paramétrages différents selon la version de NetBSD : pour donner une idée, il m'a été possible avec la branche 5 de NetBSD de démarrer les images ISO directement via PXELinux grâce à l'outil `memdisk` (mais cela n'est plus possible avec les versions 6, sinon cela aurait été trop facile). Pendant un temps, j'ai même modifié avec un éditeur hexadécimal `pxeboot_ia32.bin` pour qu'il s'identifie de manière différente selon la version et l'architecture auprès du serveur DHCP (et donc avec plusieurs clauses `if`...). Cette configuration a le mérite d'être fonctionnelle et testée, et de permettre d'installer d'autres systèmes à côté. Ainsi, mon serveur PXE me permet d'installer non seulement NetBSD, mais aussi CentOS, Fedora (j'installe mon PC de bureau principal de cette manière) et certaines versions de Microsoft Windows.

J'en profite pour remercier la communauté NetBSDfr pour son ambiance chaleureuse qui donne envie de s'intéresser à ce système ! ■

Liens utiles

Syslinux : http://www.syslinux.org/wiki/index.php/The_Syslinux_Project

NetBSD : http://www.netbsd.org/

Documentation « MENU » Syslinux : http://www.syslinux.org/wiki/index.php/Comboot/menu.c32

Pas de joli menu graphique dans PXELinux avec pxeboot : http://mail-index.netbsd.org/netbsd-users/2013/08/25/msg013203.html

Type de problème	Résolutions possibles
La machine ne peut pas démarrer sur le réseau.	Paramétrer le BIOS ou le mettre à jour si besoin. Si la machine ne peut démarrer sur la carte réseau, utiliser une image CD ou disquette du projet Etherboot (http://rom-o-matic.net/). Pour VirtualBox, pensez au pack d'extension et à paramétrer la carte réseau en Intel PRO/1000.
La carte réseau n'obtient pas d'adresse IP	Le serveur DHCP est en cause : est-il démarré et correctement paramétré ?
La carte réseau n'arrive pas à télécharger un fichier.	Probablement une erreur TFTP : celui-ci est paramétré via inetd, il faut donc vérifier à ce niveau s'il accepte les connexions. Il se peut aussi que le fichier ne soit pas au bon endroit ou que les droits soient insuffisants.
PXELinux ne trouve pas pxeboot_ia32.bin	Assurez-vous que le fichier pxeboot_ia32.bin est bien téléchargeable via TFTP dans la même arborescence que PXELinux et que c'est bien ce chemin qui est utilisé dans le fichier de configuration. On peut mettre hors de cause le lancement du serveur TFTP puisqu'il a été utilisé pour télécharger le chargeur PXELinux et sa configuration.
L'installeur NetBSD ne trouve pas les sets.	Dans notre exemple, les sets sont téléchargés via HTTP mais il n'est pas possible pour le moment de paramétrer en amont l'installeur NetBSD pour lui donner l'URL vers un miroir personnel. Pensez donc à paramétrer l'URL des sets vers lors de l'installation et assurez-vous qu'ils sont accessibles.

INTRODUCTION À FLASK, LE MICRO SYSTÈME MAOUSSE COSTAUD

par Emile <iMil> Heitor [pour GCU canal historique]

Je sais pas pour vous, mais moi, le clicka-web, ça m'ennuie. J'éprouve un profond respect pour ces génies de l'ergnonomie qui arrivent à produire des interfaces comme GitHub ou Twitter, truffées d'Ajax/jQuery[1], super inuitives, rapides et branchées sur des systèmes d'information ultra-complexes.

Récemment, j'ai eu à produire une interface de gestion de déploiement de services au sein d'un parc de machines virtuelles. Cette interface devait se brancher sur un Web-service et s'interconnecter aux mécanismes d'orchestration Fabric **[2]** et Salt **[3]**, tous deux en Python. Afin de garder une certaine cohérence dans le choix des langages, j'ai finalement jeté mon dévolu sur Flask **[4]**, le « micro framework » qui m'a fait aimer développer du clicka-web.

1 C'est mou ?

Flask est décrit comme un « microframework » en *Python*, il est basé sur *Werkzeug* pour la partie serveur / WSGI **[6]** et sur *Jinja 2* **[7]** pour le *templating*. Pour ne rien gâcher, *Flask*, ainsi que ses dépendances précédemment citées, sont sous licence BSD. Puisque *Flask* s'appuie sur *Werkzeug*, il est en mesure de fonctionner de façon autonome, sans l'aide d'un serveur *WSGI* tiers. C'est cette méthode que nous allons utiliser au début de cet article afin de rapidement mettre le pied à l'étrier, nous verrons plus tard comment configurer le serveur *uWSGI* **[8]** de façon à disposer d'une installation prête pour la production.

Flask est disponible dans la plupart des systèmes de paquets de vos UNIX-like favoris, mais nous choisirons ici d'utiliser la version disponible via *pip*, ainsi nous manipulerons la dernière version en date, la 0.10 :

```
$ pip install Flask
```

Munis du *framework*, nous allons pouvoir écrire notre première application *Flask* :

```
$ cat glmf.py
from flask import Flask
app = Flask(__name__)
@app.route('/')
def glmf_rules():
return 'GLMF rules!\n'
if __name__ == '__main__':
app.run()
```

Dans ce minuscule bout de code, après avoir importé puis instancié la classe *Flask*, nous utilisons le décorateur `@app.route` de façon à définir l'URL qui sera gérée par la fonction à suivre, ici, nous prenons en charge la racine du site. Comme on peut s'en douter, l'application basique que nous avons écrite répondra « GLMF rules! » lorsqu'elle sera interrogée sur l'URL déclarée. Finalement, lorsque le présent script est appelé par une application tierce, nous invoquons la fonction `run()` qui démarrera le serveur Web qu'embarque *Flask*.

On commence par démarrer le service :

```
$ python glmf.py
* Running on http://127.0.0.1:5000/
127.0.0.1 - - [20/Jul/2013 10:51:47] "GET /
HTTP/1.1" 200 -
```

Et on teste simplement l'URL définie :

```
$ curl -o- -s http://localhost:5000/
GLMF rules!
```

Nous constatons que, sans plus de paramètres passés à la fonction `run()`, *Flask* écoute sur l'interface `loopback`, pour modifier ce comportement, il suffit de passer un paramètre `host` :

```
if __name__ == '__main__':
app.run(host='0.0.0.0')
```

Ici, l'application écoutera sur l'ensemble des interfaces réseau du système. Afin de déboguer efficacement notre application, il est également possible de passer un paramètre `debug` à la fonction `run()` :

```
if __name__ == '__main__':
app.run(host='0.0.0.0', debug=True)
```

Ce qui aura pour effet de proposer une fenêtre de débogage interactive utilisable depuis un navigateur. Il est possible, à travers la fonction `route()`, de passer des arguments possiblement typés à une fonction associée à une URL :

```
@app.route('/<mag>')
def glmf_rules(mag):
return '{0} rules!\n'.format(mag)
```

Et de tester :

```
$ curl -o- -s http://localhost:5000/GLMF%20
Le%20Meilleur%20Mag%20De%20La%20Galaxie
GLMF Le Meilleur Mag De La Galaxie rules!
```

2 Des serpents dans mon Web 2.0

Muni de ces basiques concepts, nous allons maintenant découvrir les fonctionnalités qui font la force de *Flask*, en particulier comment produire une véritable page Web dynamique en un temps record.

La fonction **route()** utilisée dans les décorateurs de fonctions accepte le paramètre **methods**, c'est à travers ce dernier que l'on pourra choisir, au sein de la fonction, l'action à mener lorsqu'on atterrit sur l'URL déclarée. On passe au paramètre **methods** un tableau contenant la liste des méthodes supportées par la fonction :

```
@app.route('/', methods=['GET', 'POST'])
```

On choisit ensuite l'action à mener grace à la variable **request.method**. L'objet global **request** devra être importé, comme l'objet *Flask* :

```
from flask import Flask, request

@app.route('/')
def entree():
if request.method == 'GET':
return fais_des_trucs()
else:
return fais_d_autres_trucs()
```

L'objet **request**, comme on peut s'en douter, regroupe les informations relatives à la requête effectuée, par exemple on accède aux paramètres passés via la méthode **GET** à l'aide de **request.args.get** :

```
def test():
return '{0}\n'.format(request.args.get('foo'))
...
$ curl -o- -s http://localhost:5000/?foo=bar
bar
```

Ou encore aux valeurs d'un formulaire en appelant **request.form['champs']**, nous reviendrons sur cette fonctionnalité plus tard.

3 Structure d'un projet Flask

Un projet Flask typique consiste en l'arborescence suivante :

```
.
|-- gmlf.py
|-- static
|   |-- style.css
`-- templates
    |-- layout.html
    `-- site.html
```

À la racine du projet on trouve le ou les fichiers *Python* moteur de l'application, un repertoire **static** qui, comme son nom l'indique, regroupe les fichiers statiques utilisés, et un répertoire **template** contenant les *templates* au format *Jinja 2* qui seront utilisés pour générer des pages *HTML* de façon simple et intuitive.

C'est cette capacité que nous allons explorer immédiatement, afin de rapidement s'apercevoir du potentiel de ce *microframework*.

3.1 Templates

Il serait évidemment extrêmement fastidieux d'écrire l'intégralité du code *HTML* au sein du moteur lui même, aussi, la pierre angulaire du rendu de pages passe par l'écriture de simples templates qui savent récupérer des variables, voire des fonctions, issues du moteur de l'application. Créons pour l'occasion un *template* simpliste qui affichera dans une page *HTML* correctement formatée l'argument passé au paramètre *foo* via la méthode *GET* :

```
$ cat templates/main.html
<!doctype html>
<title>En voila une belle page</title>
Valeur de foo: {{ foo }}
$ cat glmf.py
from flask import Flask, request, render_
template
app = Flask(__name__)
@app.route('/')
def test():
return render_template('main.html',
foo=request.args.get('foo'))
if __name__ == '__main__':
app.run(debug=True)
```

On remarquera l'import d'un nouvel objet **render_template**, qui très naturellement est l'objet utilisé pour réaliser le rendu du *template Jinja*. L'appel à la fonction associée est limpide, on passe en premier paramètre le nom du template, puis ensuite la liste des paramètres que l'on souhaite manipuler au sein du template, ici uniquement la variable **foo** à laquelle on affecte le contenu du paramètre passé en **GET**. On vérifie simplement le bon fonctionnement du mécanisme :

```
$ curl -o- -s http://localhost:5000/?foo=bar
<!doctype html>
<title>En voila une belle page</title>
Valeur de foo: bar
```

Ceux d'entre vous ayant fait l'acquisition du GNU/Linux Magazine hors série sur *Python* le savent probablement déjà, *Jinja 2* propose une panoplie d'outils bien plus puissants que le simple affichage de variables passées au *template*, en l'occurrence, nous bénéficions ici d'un mini-langage de programmation muni entre autres de boucles, conditions et opérations sur chaînes basiques. Modifions notre exemple en ce sens :

```
$ cat templates/main.html
<!doctype html>
<title>En voila une belle page</title>
{% if foo %}
Valeur de foo: {{ foo }}
{% else %}
J'ai rien dans foo, par contre, j'ai :
{% for arg in request.args %}
- {{ arg }} qui contient {{ request.args.get(arg) }}
{% endfor %}
{% endif%}
...
$ curl -o- -s "http://localhost:5000/?fox=bar&foy=baz"
<!doctype html>
<title>En voila une belle page</title>

J'ai rien dans foo, par contre, j'ai :

- fox qui contient bar

- foy qui contient baz
```

Ah oui là tout de suite ça cause un peu plus. Comme on peut le voir, nous avons un accès total à l'objet **request** depuis le *template Jinja* en plus de la variable explicitement passée. Dans ce *template* nous testons l'existence d'une variable et bouclons sur les différents paramètres « GET » disponibles à travers l'objet **request**, une belle liste de possibilités en perspective...

3.2 Fichiers statiques

Nous l'avons vu, un projet *Flask* peut contenir dans son arborescence un repertoire **static**, ce dernier contiendra par exemple des fichiers **JavaScript**, **CSS** ou encore des fichiers média. Il sera

de bon ton de ne pas simplement déclarer un fichier statique via son chemin relatif ou absolu dans le *template* mais d'utiliser plutôt la fonction **url_for()**, qu'il conviendra d'importer également dans l'application :

```
$ head -1 glmf.py
from flask import Flask, request, render_template, url_for
...
$ head -3 templates/main.html
<!doctype html>
<title>En voila une belle page</title>
<link rel=stylesheet type=text/css href="{{ url_for('static',
filename='style.css') }}">
```

Le mot clé **static**, passé à la fonction **url_for** est défini par *Flask*, cependant, **url_for()** est utilisable à souhaits tant dans le code de l'application que dans les *templates*, cette fonction très pratique renvoie simplement l'URL à laquelle correspond une fonction, soit donc la route associée.

4 Et maintenant, on mélange

Pour se familiariser avec les notions que nous venons de découvrir, nous allons réaliser un minuscule formulaire qui pourrait servir de page d'authentification simple d'un système d'information. Le moteur *Jinja 2* permet l'utilisation de *blocks*, ce qui permet d'écrire des templates minimaux qui viendront s'imbriquer les uns aux autres. Voici par exemple une structure générique qui accueillera l'ensemble des *blocks* spécifiques :

```
$ cat templates/layout.html
<!doctype html>
<title>Bienvenue, visiteur du futur !</title>
<link rel=stylesheet type=text/css href="{{ url_for('static',
filename='style.css') }}">
<div class="page">
{% block body %}{% endblock %}
</div>
```

Ici, les directives **{% block body %}{% endblock %}** indiquent où pourra se raccrocher un *block* fils qui portera le nom **body**. Dans notre cas, **login.html** aura la forme suivante :

```
$ cat templates/login.html
{% extends "layout.html" %}
{% block body %}
<h2>Identification</h2>
<hr>
<form method="POST" action="/">
<label>Utilisateur</label><br />
<input type="text" name="user" id="user"><br />
<label>Mot de passe</label><br />
<input type="password" name="passwd" id="passwd"><br />
<input type="submit" name="action" value="login">
</form>
{% endblock %}
```

Le formalisme de ce *template* est assez explicite, on étend le *template* **layout.html** avec un *block* nommé **body**, que *Jinja* identifiera comme étant à insérer dans le *block* du même nom. Maintenant que nous disposons de nos *templates* de base, passons au code *Python* de l'application :

```
from flask import Flask, request, render_template, url_for, \
session, redirect, abort
app = Flask(__name__)
app.secret_key = 'la cle en toc'
lps = { 'imil': 'passpourri', 'jeanclaude': 'tergal', 'pascal':
'brutal' }
@app.route('/welcome')
def welcome():
if 'user' in session:
return "Identification reussie, {0} !\n".format(session['user'])
else:
return "Dehors, manant\n"
@app.route('/', methods=['GET', 'POST'])
def login():
if request.method == 'POST':
for u in lps.keys():
if u == request.form['user'] and lps[u] == request.form['passwd']:
session['user'] = request.form['user']
return redirect(url_for('welcome'))
return abort(401)
else:
return render_template('login.html')
if __name__ == '__main__':
app.run(debug=True)
```

Quelques explications s'imposent. Nous importons en premier lieu trois nouveaux objets :

- **session** contiendra des éléments de session propres à l'utilisateur, nous nous contenterons d'y stocker le nom de l'utilisateur ayant réussi son identification. Comme il est évidemment nécessaire de sécuriser cette session et les informations qu'elle transporte, il est impératif de déclarer une clé secrète au sein de l'application.
- **redirect** permet de rediriger la navigation à travers un code HTTP 302.
- **abort** stoppe toute activité de navigation en envoyant un code de retour au navigateur.

Comme cela est nécessaire dans un contexte de session, nous plaçons une clé secrète dans la variable **app.secret_key**. Suite à quoi, pour les besoins de ce test basique, nous codons en dur quelques utilisateurs dans un dictionnaire *Python*. Ces prérequis honorés, nous déclarons une fonction qui n'aura pour seul but que de signaler à l'utilisateur qu'il a réussi son identification en vérifiant si le nom **login** est bien présent dans la session en cours.

Fig. 1

Le vrai travail s'effectue dans la fonction d'entrée, **login**, déclarée comme la racine du « site ». S'il s'agit d'une requête de type **POST**, et donc qu'il s'agit très certainement du résultat de notre formulaire dûment rempli, nous parcourons

l'ensemble des couples utilisateur / mots de passe du dictionnaire *lps*. Si l'un des couples correspond aux valeurs passées dans le formulaire et récupérées via `request.form`, nous déclarons l'utilisateur dans la session courante, puis le redirigeons sur la *route* qui tient la fonction `welcome` qui lui affichera un message de bienvenue. Si l'utilisateur n'est pas reconnu, nous utiliserons la fonction `abort` qui renverra le code HTTP de retour passé en paramètre, ici 401, code classiquement utilisé pour signifier un échec d'authentification.

Évidemment, il est non seulement possible mais largement conseillé d'utiliser des méthodes d'authentification autrement plus sophistiquées, et il existe pour cela un module *Flask* tout trouvé répondant au nom de `Flask-login` **[9]**. J'ai moi-même utilisé ce module pour les besoins de notre système de déploiement et couplé à ce dernier le module `simpleldap` **[10]**, un exemple d'utilisation prêt à l'emploi de ce *combo* est visible sur mon blog **[11]**.

5 Allez, on met en prod

Si l'utilisation du moteur HTTP *Werkzeug* embarqué dans *Flask* rend très pratique la phase de débogage, il est évidemment hors de question de reposer sur un système démarré « à la main » en production. Nous nous appuierons sur deux briques connues et robustes pour diffuser notre application sur le réseau: *nginx* et *uWSGI*.

Le fâmeux serveur HTTP *nginx* dispose en effet d'une fonctionnalité lui permettant de s'adresser à un serveur *WSGI*, la seule configuration requise est la suivante :

```
location / {
    include uwsgi_params;
    uwsgi_pass unix:///var/run/uwsgi/app/
gmlf/socket;
}
```

Ces quatre lignes sont à placer dans le *virtual host* de votre choix au sein de la configuration du serveur *nginx*. Le chemin déclaré pour la directive `uwsgi_pass` est variable selon l'installation de *uWSGI* réalisée par votre distribution et/ou UNIX-like favori, dans le cas présent il s'agit d'une machine Debian GNU/Linux. J'ai choisi ici d'utiliser une *socket UNIX* mais il est bien sûr également possible de communiquer avec le serveur *WSGI* sur une *socket TCP*.

La configuration du serveur *WSGI uWSGI*, sur une machine Debian, requiert la création d'un fichier `.ini` dans le répertoire `/etc/uwsgi/apps-available` puis un lien logique du fichier créé dans `/etc/uwsgi/apps-enabled/` :

```
$ cat /etc/uwsgi/apps-available/glmf.ini
[uwsgi]
workers = 2
log-date = true
plugins = python
chdir = /var/www/glmf
module = gmlf
callable = app
```

Où :

- `workers` représente le nombre de processus *uWSGI* démarrés,
- `log-date` demande de préfixer les lignes de *logs* par la date,
- `plugins` demande le chargement du plugin *Python*,
- `chdir` change le repertoire de travail à la valeur assignée,
- `module` est le nom du module à charger,
- `callable` donne le nom de l'application W*SGI* à appeler par défaut.

Il n'est pas rare de lire les fichiers `.ini` *uWSGI* concaténant *module* et *callable* de cette façon :

```
module = gmlf:app
```

Une fois les deux serveurs configurés, il ne nous reste plus qu'à les (re-)démarrer, ce qui nous donne chez Debian :

```
$ sudo /etc/init.d/uwsgi restart
$ sudo /etc/init.d/nginx restart
```

Cette fois, notre application écoute, à travers *nginx*, sur le port 80 (si ce dernier à été configuré ainsi) et dispose d'un système de gestion de service digne de ce nom. À nous la prod !

6 Il est pourri ton Web 2.0 mec...

Oui, bon, ok, ça fait pas encore des animations au chargement et ça autocomplète pas les champs, mais ne jugez pas trop vite, car *Flask* coopère nativement parfaitement avec des bibliothèques comme *jQuery*, en effet, à l'aide de la fonction *jsonify()*, il est aisé de déclarer des *routes* spécialement dédiées à répondre à des appels *AJAX* et ainsi remplir « automagiquement » un *drop-down menu* ou encore pré-remplir des listes afin de proposer des champs disposant de fonctions d'*autocomplete*. Ajoutons à cela plus de 250 *plugins* permettant d'interfacer à peu près n'importe quoi, après avoir passé quelques heures à manipuler ce soit-disant microframework et quelques-unes de ses extensions, on s'aperçoit que ses possibilités sont à la mesure de son accessibilité. ■

Références

[1] http://jquery.com/

[2] http://docs.fabfile.org/en/1.6/

[3] http://saltstack.com/community.html

[4] http://flask.pocoo.org/

[5] http://werkzeug.pocoo.org/

[6] http://fr.wikipedia.org/wiki/Web_Server_Gateway_Interface

[7] http://jinja.pocoo.org/

[8] http://uwsgi-docs.readthedocs.org/en/latest/

[9] https://flask-login.readthedocs.org/en/latest/

[10] https://pypi.python.org/pypi/simpleldap

[11] http://imil.net/wp/2013/07/06/ldap-flask-login-snippet/

SDL 2 ET OPENGL ES 2 SUR SYSTÈMES ANDROID 2+

par Yves Bailly

La bibliothèque SDL est apparue en version 2.0.0 en août dernier, après plusieurs années de développement. Parmi les nombreuses nouvelles fonctionnalités et améliorations, il est désormais possible de l'utiliser pour développer des applications pour systèmes Android en langages C/C++, une bonne nouvelle pour ceux qui restent allergiques au langage Java.

Il est probablement inutile de présenter la bibliothèque SDL **[1]**, l'une des plus utilisées pour la création de jeux, aussi bien dans le monde libre (de FreeCiv à 0 A.D.) que dans nombres de titres propriétaires. Diffusée selon les termes de la très permissive licence zlib, sa simplicité et sa grande portabilité en font un choix naturel pour de nombreux projets.

Nous allons nous intéresser ici à l'utilisation de SDL2 dans l'objectif de créer une application pour systèmes Android, les plus répandus sur les appareils mobiles (téléphones et tablettes). Et pour faire un peu mieux que simplement afficher « bonjour » à l'écran, nous allons créer une application 3D exploitant les fonctionnalités de OpenGL ES **[2]**, le sous-ensemble de OpenGL dédié aux systèmes embarqués. Pour rester compatibles avec la plupart des matériels existants, nous ne nous appuierons pas sur la très récente version 3 de OpenGL ES mais nous nous limiterons à la version 2.

Dans ce qui suit, nous supposerons que le répertoire contenant le SDK Android **[3]** est disponible par une variable d'environnement `$ANDROID_SDK`, tandis que le NDK **[4]** est disponible par une variable `$ANDROID_NDK`.

1 L'application

Nous allons rester simples dans nos ambitions, malgré l'utilisation de la 3D. Notre application va se contenter d'afficher un triangle coloré tournant autour d'un axe, l'un des exercices les plus élémentaires en OpenGL. Le résultat ressemblera à la Figure 1, où l'application (et le triangle) tourne sur un téléphone Android assez basique, un modèle à moins de 100€ avec écran de 3,5 pouces (320x480 pixels, 262000 couleurs), processeur dual-core Cortex A9 et Android 4.1 (Jelly Bean). S'il s'agit d'un matériel récent, l'application a également été testée sur un appareil plus ancien sous Android 2.3.

Ce programme étant très simple, on pourrait tout placer dans un unique fichier, voire pratiquement dans une unique fonction `main()`. Mais pour être plus proche d'un cas « réel », nous allons séparer l'initialisation, la boucle d'événements et le rendu dans autant de fichiers sources. De plus, nous allons programmer en C++ plutôt qu'en C, exploitant les services de GLM **[5]** pour les aspects mathématiques (notamment les matrices de transformation).

Figure 1 : Un triangle rotatif sous Android

Le premier problème à résoudre est l'accès aux fonctions OpenGL ES 2 et la création du contexte graphique. Fort heureusement, SDL nous simplifie grandement la vie. Ces quelques lignes suffisent :

```
#include <SDL_opengles2.h>
/* ... */
  SDL_Init(SDL_INIT_VIDEO bitor SDL_INIT_TIMER);
  SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK, SDL_GL_CONTEXT_
PROFILE_ES);
  SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 2);
  SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1);
  SDL_GL_SetAttribute(SDL_GL_ACCELERATED_VISUAL, 1);
  SDL_Window* const window =
    SDL_CreateWindow("Triangle",
                     SDL_WINDOWPOS_CENTERED,
                     SDL_WINDOWPOS_CENTERED,
```

```
                    320, 480,
                    SDL_WINDOW_OPENGL bitor SDL_WINDOW_RESIZABLE);
  SDL_GLContext context = SDL_GL_CreateContext(window);
  SDL_GL_MakeCurrent(window, context);
/* ... */
```

L'inclusion en première ligne nous donne accès aux types, contantes et prototypes des fonctions OpenGL ES 2, sans qu'il soit nécessaire de les déclarer nous-mêmes. Après l'initialisation du moteur SDL (ligne 3), les appels à **SDL_GL_SetAttribute()** permettent de qualifier le contexte OpenGL que nous voudrons obtenir : un profile « ES » (ligne 4) par opposition au profile « desktop » (bureau) par défaut, en version 2 (ligne 5), exploitant le double tampon (ligne 6) pour améliorer les performances et requérant l'accélération matérielle (ligne 7). Ensuite nous pouvons créer la fenêtre (lignes 9 à 13). En pratique, la taille donnée n'a pas beaucoup d'importance : il est fréquent que les applications soient systématiquement affichées maximisées ou en plein écran par Android. Enfin nous pouvons obtenir le contexte OpenGL, dans lequel sera effectué nos tracés.

Naturellement, il faudrait systématiquement vérifier le bon déroulement de chaque opération, notamment la création du contexte. Tout ces contrôles ont été omis ici.

Le reste de l'application ne présente pour l'instant pas beaucoup d'intérêt. La boucle d'événements est tout ce qu'il y a de plus classique, une boucle infinie reposant sur la fonction **SDL_PollEvent()** pour recevoir les événements – actions de l'utilisateur, changement de taille de la fenêtre, etc. La rotation est mise en œuvre en utilisant une minuterie (un timer, fonction **SDL_AddTimer()**) qui va demander un rafraîchissement toutes les 20 millisecondes, si possible.

Le dessin de la scène, justement, est également très simple, malgré la lourdeur de la « mise en route » d'OpenGL pour un cas aussi élémentaire. Juste une paire de shaders (dans des variables chaînes), un tampon pour les positions des trois points et un autre pour la couleur de chacun, une matrice de projection orthographique ajustée à la taille de la fenêtre et une matrice de transformation pour la rotation en fonction du temps. Tout cela en moins de 150 lignes, en laissant de la place.

Juste une petite particularité : pour quitter l'application, mettre fin à son fonctionnement, il est nécessaire de passer par la fonction **exit()**. Il ne suffit pas de terminer la fonction **main()**. Cela tient au fait que le code natif (C/C++) est en fait « embarqué » dans un thread distinct du thread principal de la « vraie » application, qui est nécessairement en Java.

Pour notre exemple, tous les codes sources sont placés dans un dossier nommé **triangle**, y compris les en-têtes de GLM, dans un sous-dossier nommé **glm**.

2 Compilation pour Android

C'est maintenant que les choses sérieuses commencent. Pour commencer, décompactez quelque part l'archive contenant le code source de SDL2 que vous venez de télécharger (n'est-ce pas ?), dans un répertoire temporaire ou un endroit plus persistant, peut importe. À l'heure où ces lignes sont écrites, cela nous donne un dossier **SDL2-2.0.0**. Surtout, ne configurez ni ne compilez rien pour l'instant ! Si vous avez voulu prendre les devants et déjà configuré SLD2 avec le traditionnel **./configure**, repartez d'une archive fraîchement extraite.

Copiez le dossier **android-project** contenu dans l'archive à l'endroit où vous stockez vos projets, puis renommez-le du nom de votre programme. Pour nous, cela sera **triangle_android**. Ensuite, histoire de remettre la poule dans l'œuf, copiez ou liez le dossier d'origine **SDL2-2.0.0** (oui, celui issu de l'archive des sources SDL2) dans le répertoire **triangle_android/jni** et renommez-le en **SDL**. Enfin, placez le dossier contenant vos propres sources dans le répertoire **triangle_android/jni/src**.

Finalement l'arborescence devrait ressembler à la Figure 2, qui ne montre que l'essentiel.

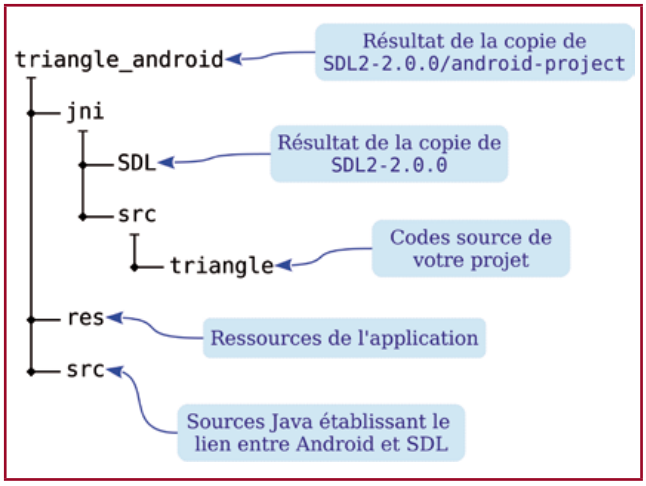


Figure 2 : L'arborescence d'un projet SDL2/Android

Maintenant, un peu de paramétrage. Ou plutôt beaucoup de paramétrage... Commençons par le fichier **jni/src/Android.mk**, dédié à la compilation de nos propres sources.

Ajoutez la ligne suivante pour spécifier la version du NDK que vous souhaitez utiliser, par exemple après la ligne **LOCALE_MODULE** – ici, on indique la dernière version disponible :

```
TARGET_PLATFORM := android-9
```

Ensuite, ajoutez vos codes sources à la variable **LOCAL_SRC_FILES**. Dans notre cas, cela donne :

```
LOCAL_SRC_FILES := $(SDL_PATH)/src/main/android/SDL_android_main.c \
  triangle/event_loop.cpp \
  triangle/scene.cpp \
  triangle/main.cpp
```

Nous allons utiliser quelques aspects du langage C++11, il est nécessaire d'en informer le compilateur GCC avec la ligne :

```
LOCAL_CPPFLAGS += -std=c++0x
```

Et comme nous allons utiliser OpenGL ES version 2, il faut nous lier à la bonne bibliothèque, en ajustant la ligne suivante :

```
LOCAL_LDLIBS := -lGLESv2 -llog
```

La liaison avec la bibliothèque **log** nous donnera accès à quelques facilités pour afficher des « traces », que nous verrons un peu plus loin.

Nous en avons terminé avec **jni/src/Android.mk**. Un autre fichier doit être modifié pour le paramétrage de la compilation en elle-même, le fichier **jni/Application.mk** :

- si vous utilisez des fonctionnalités de la STL (Standard Template Library) du C++, décommentez ou ajoutez la ligne **APP_STL := stlport_static** ;
- ajoutez **NDK_TOOLCHAIN_VERSION=4.8** pour compiler avec la version la plus récente de GCC ;
- enfin, pour cibler les processeurs supportant le jeu d'instruction ARMv7, ajoutez la ligne **APP_ABI := armeabi-v7a**.

Les processeurs ARMv7 présentent l'avantage de disposer d'une unité de calcul en virgule flottante (FPU), ce qui améliore considérablement les performances pour une application exploitant OpenGL. Toutefois les matériels plus anciens ou plus bas de gamme ne disposent souvent pas d'un tel processeur. Pour une compatibilité maximale, spécifiez simplement **armeabi** dans **APP_ABI**. Vous pouvez également inscrire **x86** si vous ciblez un matériel à base de processeur type Intel.

Nous sommes enfin prêts à compiler notre code C/C++. Le répertoire courant étant la racine du projet, **triangle_android** dans notre exemple, exécutez :

```
$ $ANDROID_NDK/ndk-build
```

Le script **ndk-build**, fourni avec le NDK Android, est un emballage de la commande **make** qui positionne automatiquement un certain nombre de variables. Vous pouvez donc lui passer les options usuelles de **make**, comme **-B** pour forcer une recompilation totale ou **-jN** pour compiler plusieurs fichiers en parallèle.

Si tout se passe bien, le résultat de cette compilation devrait prendre la forme de deux bibliothèques, **libSDL2.so** et **libmain.so**, placées dans le répertoire **libs/armeabi-v7a/**.

3 Construction du paquetage

Nous devons maintenant procéder à la création du paquetage Android, le fichier ***.apk** qui pourra effectivement être installé sur votre matériel ou votre émulateur préféré (bien qu'un « vrai » matériel soit de loin préférable, ne serait-ce que pour des raisons de performances). Pour rester « manuel », nous allons effectuer l'opération en ligne de commande, en ajustant encore quelques fichiers. Si vous préférez un bel environnement de développement intégré et graphique, voyez plus loin la section consacrée à Eclipse **[6]**.

L'essentiel se passe dans le fichier **AndroidManifest.xml**, à la racine du projet.

Pour commencer, modifiez l'attribut **package** de l'élément **<manifest>** en remplaçant **"org.libsdl.app"** par l'identifiant de votre programme, par exemple dans notre cas **"org.ybailly.triangle"**.

Dans l'élément **<activity>** (dans **<application>**), ajoutez l'attribut **android:configChanges="orientation"** pour recevoir les événements de changement d'orientation, ce qui inclus également les changements de taille de la fenêtre. Changez également l'attribut **android:name** pour y placer le nom de l'activité principale de votre programme à la place du nom **SDLActivity**, pour nous ce sera **Triangle**. Par soucis de cohérence, faite de même dans l'attribut **name** de l'élément **<project>** du fichier **build.xml**.

Finalement notre « manifeste » modifié ressemble à ceci :

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="org.ybailly.triangle"
      android:versionCode="1"
      android:versionName="1.0"
      android:installLocation="auto">
    <application android:label="@string/app_name"
                 android:icon="@drawable/ic_launcher"
                 android:allowBackup="true"
                 android:theme="@android:style/Theme.NoTitleBar.Fullscreen">
        <activity android:name="Triangle"
                  android:label="@string/app_name"
                  android:configChanges="orientation">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
```

```
        </activity>
    </application>
    <uses-sdk android:minSdkVersion="10" android:targetSdkVersion="10" />
    <uses-feature android:glEsVersion="0x00020000" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
</manifest>
```

Ajuster l'API cible dans l'élément **<uses-sdk>** et dans le fichier **project.properties** ou laisser la valeur par défaut **android-10** (Android 2.3 et supérieurs). Cela dépend évidemment des versions du SDK que vous avez installées – utilisez **$ANDROID_SDK/tools/android update sdk** pour en ajouter au besoin.

Nous ne décrirons pas davantage ce fichier, reportez-vous à la documentation officielle Android pour plus d'informations.

Nous devons maintenant créer la classe qui va représenter l'activité principale (et unique en fait) de notre application. D'après l'attribut **package** dans **<manifest>** et l'attribut **android:name** dans **<activity>** du fichier **AndroidManifest.xml** précédent, créez un fichier **src/org/ybailly/triangle/Triangle.java** avec ce contenu :

```
package org.ybailly.triangle;
import org.libsdl.app.SDLActivity;
public class Triangle extends SDLActivity { }
```

Ensuite, dans le fichier **res/values/strings.xml**, changez le contenue **"SDL App"** de l'élement **<string>** pour le nom « public » de votre application, par exemple **"Mon Triangle"**.

Enfin, dernière opération, créez un fichier **local.properties** à la racine du projet et inscrivez une ligne comme celle-ci, pour indiquer où se trouve le SDK Android :

```
sdk.dir=/home/yves/opt/android-sdk-linux
```

Nous pouvons maintenant créer le paquetage Android, simplement en exécutant depuis la racine du projet :

```
$ ant debug
```

Pour tester une application utilisant OpenGL ES, il est vivement recommandé d'utiliser un « vrai » matériel : l'émulateur ne fonctionne pas toujours très bien avec OpenGL, sans parler de ses performances… disons, très moyennes. Dans tous les cas, soit avec un matériel connecté, soit avec un émulateur en fonctionnement, exécutez :

```
$ ant debug install
```

Cela va installer votre application. Pour la démarrer :

```
$ $ANDROID_SDK/platform-tools/adb shell am start -n org.ybailly.
triangle/org.ybailly.triangle.Triangle
```

Remarquez que l'on retrouve les noms du paquetage et de l'activité dans cette commande.

4 Déboggage

Un moyen rudimentaire mais simple de débogger une application est de laisser des « traces », c'est-à-dire d'afficher des informations sur ce qui se déroule lors de son exécution. Pour une application Android, le problème est de récupérer ces traces.

Si vous ne tenez pas particulièrement à maintenir un code multi-plateforme, le plus simple est sans doute d'utiliser la fonction **__android_log_print()** déclarée dans **<android/log.h>**, par exemple :

```
__android_log_print(ANDROID_LOG_INFO, "Triangle", "Received event
%d", event.type);
```

Le premier paramètre est le « niveau » d'importance du message, le second est un marqueur identifiant sa source. Le troisième et les suivants sont le message lui-même, sur le même modèle que ce bon vieux **printf()**. Consultez la documentation officielle pour plus de détails.

Une fois l'application lancée, par exemple avec la méthode évoquée plus haut, vous pouvez obtenir les messages émis par **__android_log_print()** à l'aide de la commande **logcat** de l'utilitaire **adb** :

```
$ $ANDROID_SDK/platform-tools/adb logcat Triangle:I *:S
```

À nouveau nous n'allons pas entrer dans les détails, la documentation officielle devrait vous donner toutes les informations nécessaires. Disons simplement que les paramètres après **logcat** sont des filtres de messages, qui ici indiquent que nous voulons les messages identifiés par « **Triangle** » (le marqueur utilisé plus haut) et au moins du niveau **I**nformation. Le paramètre ***:S** a pour effet de supprimer tous les autres messages (essayez de l'enlever, juste une fois).

Si vous voulez maintenir un code relativement multi-plateforme et utiliser les plus communs **printf()** ou **std::cout**, qui normalement aboutissent dans un trou noir sous Android, il est possible de faire en sorte qu'ils soient redirigés vers le **logcat** Android. Toutefois cette méthode s'avère assez peu fiable, jusqu'à provoquer un plantage du matériel pour certains. La meilleur solution consiste donc à « emballer » les appels usuels dans une ou plusieurs fonctions, dont le contenu sera adapté selon la plate-forme, par exemple quelque chose comme ça (mais en plus sophistiqué sans doute) :

```
void Print(char const* str)
{
#ifdef __ANDROID__
  __android_log_print(ANDROID_LOG_INFO, "Mon_Marqueur", str);
#else
  std::cout << str << "\n";
#endif
}
```

5 Intégration dans Eclipse

Si vous ne connaissez pas Eclipse, il s'agit d'un environnement de développement intégré (IDE) proposant une interface graphique pour la mise en œuvre de nombres d'outils utilisés en développement. C'est un peu le Visual Studio du monde libre, mais en plus beau, plus souple, plus lourd et écrit en Java. D'ailleurs son objectif premier était d'offrir un IDE pour le développement Java, mais rapidement de nombreux greffons (plugins) sont apparus pour permettre l'utilisation d'autres langages – dont le C++.

Si vous ne l'avez pas déjà, il vous faut donc installer Eclipse. Pour les distributions majeures, il suffit normalement de faire appel à votre gestionnaire de paquets préféré, par exemple pour Debian et ses dérivées :

```
# apt-get install eclipse eclipse-anyedit
eclipse-cdt
```

En plus du paquetage Eclipse, on installe également le greffon AnyEdit qui fournit pas mal de fonctions pratiques lors de l'édition de code sources ou fichiers textes (affichage des espaces, conversion entre tabulations et espace, etc.), ainsi que le greffon CDT, dédié au développement en C/C++. Ceci va occuper environ 260Mo sur votre disque dur.

Ensuite nous allons installer le greffon spécifique à la programmation pour Android. Passez par le menu ***Help > Install new software***. Dans la fenêtre qui apparaît, cliquez sur le bouton ***Add***, en haut à droite. Indiquez « ADT Plugin » dans le champ ***Name***, et https://dl-ssl.google.com/android/eclipse/ dans le champ ***Location***, puis cliquez sur ***OK***. Cochez ***Developer Tools*** et ***NDK Plugins*** dans la liste, puis cliquez ***Next***, puis ***Next***, puis acceptez les différentes licences, puis ***Finish***. Le téléchargement peut prendre un peu de temps. Lorsque tout est terminé, redémarrez Eclipse s'il ne vous le propose pas.

Il est maintenant nécessaire d'indiquer où se trouvent le SDK et le NDK Android. Passez par le menu ***Windows > Preferences*** puis allez dans la section **`Android`**. Indiquez le répertoire où se trouve votre SDK. Puis dans la sous-section **`NDK`**, indiquez le répertoire où se trouve le NDK.

Revenons à notre projet. Préparez-le comme indiqué dans la Section 2. Cela fait, depuis Eclipse, créez un nouveau projet par le menu ***File > New > Other*** ou utilisez le raccourci ***Ctrl+N***. Dans la liste des modèles de projets qui apparaît, choisissez ***Android > Android project from existing code***, puis ***Next***. Indiquez le répertoire racine qui contient le projet, **`triangle_android`** dans notre exemple. Il est alors probable que deux projets apparaissent dans la liste des projets identifiés : le nôtre repéré par son chemin complet, ainsi que **`jni/SDL/android-project`** – issu des sources même de SDL et que nous avions préalablement copié comme modèle pour notre projet. Décochez-le pour éviter des ennuis. Éventuellement changez le nom du projet qui nous intéresse, puis cliquez sur ***Finish***.

Une fois le projet disponible dans Eclipse, cliquez-droit sur son nom et choisissez ***Android tools > Add native support*** : ainsi les fichiers C/C++ seront pris en compte lors de la construction du paquetage.

Nous pouvons maintenant paramétrer le paquetage Android au travers de l'interface graphique, en reproduisant toutes les opérations décrites dans la Section 3 plus haut. Si vous ouvrez le fichier **`Androidmanifest.xml`**, vous en aurez une représentation sous forme de champs de saisie, de listes, etc. plutôt qu'une version XML « brute » en mode texte. La Figure 3 montre à quoi ressemble la

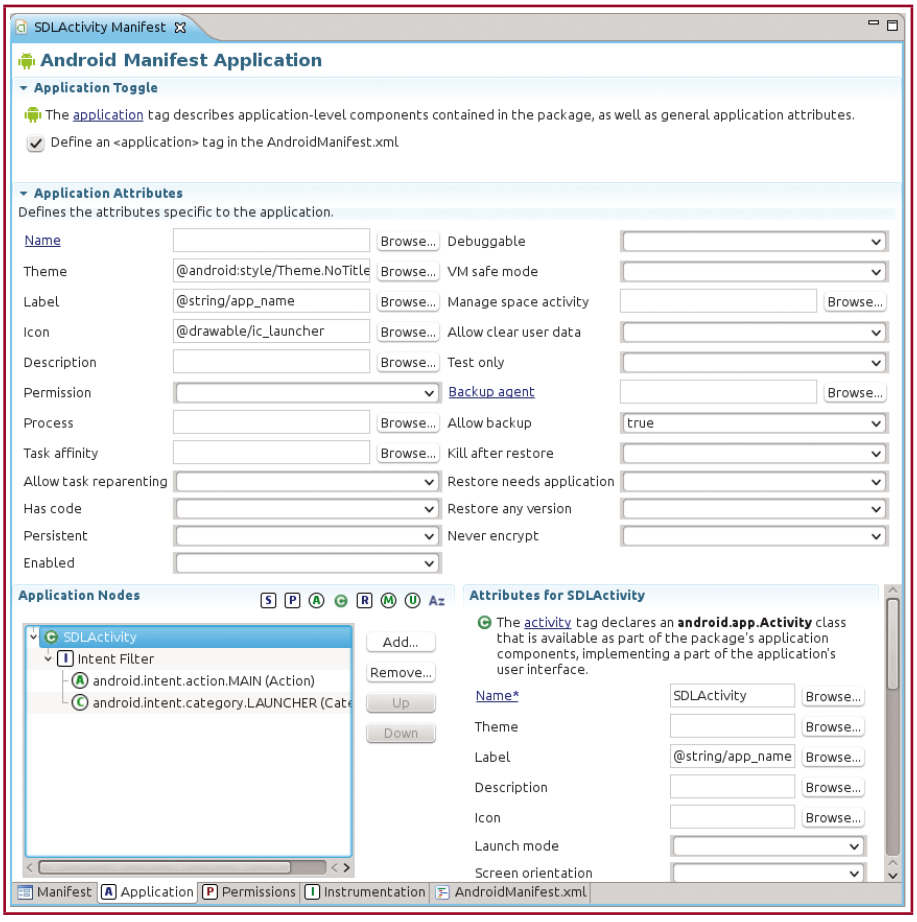


Figure 3 : La vue de l'élément <activity> du manifeste dans Eclipse

vue de l'élément **<application>**, obtenue en sélectionnant l'onglet éponyme en bas de la vue du manifeste. L'onglet le plus à droite donne accès à la version « brute », si nécessaire.

Nous ne détaillerons pas davantage l'utilisation de Eclipse pour le développement Android (natif ou non), qui nécessiterait un manuel complet. Au besoin, il existe de nombreuses ressources sur l'Internet pour vous permettre d'avancer dans cet exercice.

6 Les événements spécifiques

SDL2 nous donne accès à quelques événements qui sont spécifiques à un système Android, ou plus généralement à une surface tactile. Par exemple, nous pouvons détecter lorsqu'un doigt est posé sur la surface et s'en détache. Dans la boucle d'événements, on pourrait utiliser cette information pour changer l'axe de rotation de notre triangle :

```
SDL_Event event;
while ( SDL_PollEvent(&event) )
{
  switch ( event.type )
  {
    /* ... */
    case SDL_FINGERDOWN:
      finger_down_x = event.tfinger.x;
      finger_down_y = event.tfinger.y;
      break;
    case SDL_FINGERUP:
      {
        float const dx = event.tfinger.x - finger_down_x;
        float const dy = event.tfinger.y - finger_down_y;
        if ( std::abs(dx) > std::abs(dy) )
          Set_Rotation_Axis(glm::vec3(0.0f, 1.0f, 0.0f));
        else
          Set_Rotation_Axis(glm::vec3(1.0f, 0.0f, 0.0f));
      }
      break;
```

Les informations sur la pression reçue sont dans une sous-structure **SDL_TouchFingerEvent** de la structure **SDL_Event**, accessible par le membre **tfinger**. Les champs **x** et **y** donnent la position du doigt au moment de l'événement, normalisée entre 0 et 1 – donc indépendamment de la résolution courante, ce qui est fort pratique.

Concernant les événements de type clavier, il est possible d'intercepter la touche qui affiche normalement un menu et la touche « arrière » ou « retour ». Le code suivant utilise la première pour à nouveau changer l'axe de rotation, tandis que la seconde provoque la terminaison de l'application :

```
case SDL_KEYDOWN:
  if ( event.key.keysym.sym == SDLK_MENU )
  {
    Set_Rotation_Axis(glm::vec3(0.0f, 0.0f, 1.0f));
  }
  else if ( event.key.keysym.sym == SDLK_AC_BACK )
  {
    SDL_GL_DeleteContext(context);
    SDL_DestroyWindow(window);
    SDL_Quit();
    exit(0);
  }
  break;
```

Dans les sources de SDL, consultez l'exemple **test/testgesture.c** ainsi que la documentation pour avoir une idée des possibilités qui sont offertes.

Si vous avez besoin d'une « véritable » entrée de texte, la fonction **SDL_StartTextInput()** fera apparaître le clavier virtuel, et **SDL_StopTextInput()** le fera disparaître. La saisie en elle-même pourra être obtenue en interprétant les événements **SDL_TEXTINPUT** et **SDL_TEXTEDITING**. Consultez également le fichier d'en-tête **SDL_system.h**, vous y trouverez des fonctions dédiées aux plate-formes mobiles, comme **SDL_AndroidGetJNIEnv()** pour obtenir l'environnement JNI (Java Native Interface, qui permet l'appel de fonctions entre le code Java et le code C) sous Android.

Conclusion

À quelques détails prèt, SDL2 nous permet de développer des applications pour Android tout en conservant une certaine portabilité : l'immense intérêt est d'avoir une majorité de code commun à une version « mobile » et à une version « bureau ». Et cela jusqu'à la création d'un contexte OpenGL et son utilisation, ouvrant les portes d'un affichage 3D sophistiqué – mais l'affichage 2D garde évidemment tout son intérêt, nombres de jeux n'ont pas besoin de la troisième dimension.

Dans un prochain article, nous reviendrons sur l'utilisation de la bibliothèque Qt pour le développement sur Android, qui est annoncé comme pleinement supporté dans la version 5.2 à venir (avant la fin de l'année 2013), suite à l'intégration dans Qt même du projet Necessitas que nous avions évoqué il y a quelques mois (voir GNU/Linux Magazine 156 de janvier 2013). ■

Références

[1] SDL : http://www.libsdl.org/

[2] OpenGL ES : http://www.khronos.org/opengles/

[3] SDK Android : http://developer.android.com/sdk/index.html

[4] NDK Android : http://developer.android.com/tools/sdk/ndk/index.html

[5] GLM : http://glm.g-truc.net/

[6] Eclipse : http://www.eclipse.org/

RECONNAISSANCE FACIALE FACILE AVEC OPENCV ET PYTHON !

par Frédéric Le Roy

En ces temps sombres où le grand public commence à prendre conscience de la faiblesse (voir de l'absence) de la notion de vie privée sur internet, je vous invite à passer du côté obscur et à devenir votre propre Big Brother...

1 La reconnaissance faciale, kesako ?

La reconnaissance faciale est l'art d'associer une identité à un visage. Pour l'être humain c'est quelque chose de simple et naturel, dès le plus jeune âge. Pour un ordinateur, c'est moins simple... La première question étant « où est situé le visage dans une image ? », suivie de « à qui appartient ce visage et quel est mon degré de confiance envers mon diagnostic ? ».

Trouver un visage dans une image est une fonctionnalité qu'aujourd'hui on trouve partout, que ce soit sur les logiciels de retouche photo, sur les écrans des appareils photos mais aussi sur les téléphones. L'association du visage à une identité est plus rare mais présente sur des sites grand public (comme facebook par exemple pour citer un seigneur sith très puissant), ce qui fait que finalement c'est une fonctionnalité assez banal pour monsieur Toutlemonde.

2 Pourquoi vouloir faire de la reconnaissance faciale ?

Les motivations peuvent être multiples, que ce soit dans le domaine privé ou professionnel. Dans le cadre d'une entreprise, cela peut servir dans des politiques de sécurité (identifier et référencer les personnes rentrant dans un local à haute sécurité), dans du référencement d'images (exemple de facebook), ...

Notez que l'utilisation de la reconnaissance faciale n'est pas considérée comme une sécurité forte. En effet, il suffit de porter un masque avec la photo d'une personne pour tromper la caméra ! Elle sera donc utilisée en complément d'autres moyens et pourra, par exemple, lever une alerte si un visage détecté ne correspond pas au badge.

Dans le cadre de la vie privée, les besoins sont moins évidents... A titre personnel, je pense installer une caméra dans mon entrée, pointée vers la porte pour identifier les visiteurs (enfin ceux référencés). Une fois une personne identifiée, un événement sera diffusé sur le réseau et intercepté par ma solution domotique qui lorsqu'elle me verra rentrer pourra donc m'annoncer ce que je lui aurais programmé !

Ayant, malgré mon penchant pour le côté obscur, un certain respect de la vie privée, cette caméra ne sera accessible d'aucune IHM (que ce soit via un navigateur ou autre moyen) et les données seront traitées de manière anonyme. Par sécurité (en cas de vol), les captures de visages seront quand même stockées pendant N jours et ceci dans un fichier chiffré : ainsi en cas de piratage, les photos de ma femme se baladant en sous vêtement ne se retrouveront pas sur internet :). Mais tout ceci sort du cadre de cet article : nous nous cantonnerons à la reconnaissance faciale.

3 Ce qui ne sera pas abordé dans cet article

En plus de la lingerie de ma femme, je ne détaillerais pas dans cet article les principes mathématiques qui permettent de détecter un visage et de l'identifier : non seulement, ce sont des choses qui me dépassent, mais en plus je risque de mal vous retransmettre ce que j'aurais lu. Je vous invite à lire les documentations en ligne d'OpenCV pour ceci **[OpenCV]**. Elles regorgent d'explications mathématiques en tout genres...

Nous aborderons donc humblement comment créer une librairie qui utilise OpenCV afin de créer des collections de photos, de trouver et d'identifier des personnes sur les images venant d'une caméra.

4 Premier contact avec OpenCV

Le titre indiquant que nous allons travailler en python, installons le package **python-opencv** via la commande :

```
# apt-get install python-opencv
```

Attention, pour cet article j'ai utilisé la version 2.4.3 :

```
$ python
Python 2.7.5+ (default, Jun 13 2013, 13:26:49)
[GCC 4.8.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import cv2
>>> print cv2.__version__
2.4.3
```

Nous aurons aussi besoin de la librairie **numpy**. Installez la dernière version disponible sur votre système :

```
# apt-get install python-numpy
```

Pour ma part, j'ai travaillé avec la version 1.7.1.

Créons maintenant notre premier programme. Il aura pour effet d'ouvrir le flux de la caméra de mon ordinateur portable et de l'afficher dans une fenêtre :

```
#!/usr/bin/env python2
import cv2

class PremierContact():

    def __init__(self):
        self.camera = None

    def prepare_camera(self, capture_device):
        self.camera = cv2.VideoCapture(capture_device)

    def is_camera_available(self):
        if self.camera.isOpened():
            rval, frame = self.camera.read()
            return rval
        else:
            rval = False
            return False

    def capture(self):
        cv2.namedWindow("preview")
        rval = True
        while rval:
            rval, frame = self.camera.read()
            cv2.putText(frame, "Appuyer sur [ESC] pour quitter.", (5, 25),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0,0,0))
            cv2.imshow("preview", frame)

            key = cv2.waitKey(20)
            if key in [27, ord('Q'), ord('q')]: # exit on ESC
                break

if __name__ == "__main__":
    PC = PremierContact()
    PC.prepare_camera(0)
    if PC.is_camera_available():
        PC.capture()
```

Tout d'abord, nous avons importé la librairie **cv2** qui est une des deux librairies officielles de OpenCV pour python. La première et plus ancienne, **cv** est plus lente et a une manière bien spécifique de traiter les images. **Cv2** pour sa part est plus rapide et convertit tout en tableaux « numpy ». **Numpy** est une librairie qui permet des traitements rapides sur les tableaux. Chaque image (**frame** dans notre code) sera un tableau de type **ndarray** et **frame[4,6]** donnera la valeur du pixel à la position (4,6). Il est donc possible de réaliser simplement ses propres traitements avec **cv2**.

Comme j'aime bien travailler avec des classes et que le but final est d'obtenir une librairie, on créée une classe **PremierContact**. Le constructeur (**__init__**) se contentera d'initialiser la seule variable de la classe.

La fonction **prepare_camera** va définir le périphérique utilisé pour la capture, à savoir ici la première caméra trouvée sur la machine (**capture_device=0**).

La fonction **is_camera_available** vérifie que la caméra est bien ouverte et surtout que nous sommes en mesure de capturer une frame. La fonction **read** retourne :

- un statut : True ou False si une frame a pu être capturée ou non,
- la frame capturée.

Finissons avec la fonction **capture** qui va ouvrir une fenêtre de visualisation de la capture. Ensuite, nous attaquons une boucle tant qu'il y des frames à capturer. On récupère la frame ainsi :

```
rval, frame = self.camera.read()
```

On y ajoute un texte :

```
cv2.putText(frame, "Appuyer sur [ESC] pour quitter.", (5, 25),
        cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0,0,0))
```

Le premier paramètre est l'image sur laquelle on ajoute le texte. Ensuite vient le texte, les coordonnées du texte, la police utilisée, la taille de la police et sa couleur sur 3 composantes (ici en noir).

On finit par afficher l'image avec **cv2.imshow(...)** et on teste si une touche a été pressée pour quitter le programme, auquel cas on sort de la boucle while.

On finit le programme par l'instanciation de la classe que nous venons de créer et par l'appel des différentes fonctions dans l'ordre qui va bien.

Voici un exemple de ce que l'on obtient :

Fig. 1

5 Détection des visages

Maintenant passons à la détection des visages. Pour ceci, nous aurons besoin du fichier **lbpcascade_frontalface.xml**. Il vous suffit d'installer le package **libopencv-dev** pour l'obtenir dans **/usr/share/opencv/lbpcascades/lbpcascade_frontalface.xml** :

```
# apt-get install libopencv-dev
```

Et voici le code où nous commençons par les classiques shebang et **import** :

```
#!/usr/bin/env python2

import cv2
import numpy
import time
```

Maintenant on définit l'emplacement du fichier **lbpcascade_frontalface.xml** et le ratio de diminution que nous appliquerons sur l'image avant la détection des visages : **DOWNSCALE**.

```
TRAINSET = "/usr/share/opencv/lbpcascades/lbpcascade_frontalface.xml"
DOWNSCALE = 4
```

Créons maintenant notre classe **Detection** avec son constructeur. Le paramètre **interval** définira le temps d'attente entre la capture de 2 frames (afin de limiter les ressources processeur sur de faibles machines).

```
class Detection():

    def __init__(self, interval = 0.1):
        self.camera = None
        self.classifier = None
        self.interval = interval
```

Nous retrouvons les 2 fonctions du chapitre précédent : **prepare_camera** et **is_camera_available**. Nous allons juste instancier un **CascadeClassifier** dans la fonction **prepare_camera**. Cette fonction, permet de recharger un élément dans une image. Vous trouverez une explication complète sur le fonctionnement à cette url : **[Cascade classification]**. En fonction de ce que l'on cherche sur une image, on pourra changer le **TRAINSET**. Dans le cas de la recherche de visages, j'ai utilisé ici **lbpcascade_frontalface.xml**. Notez que j'aurais aussi pu utiliser le fichier **/usr/share/opencv/haarcascades/haarcascade_frontalface_default.xml**. La différence entre les 2 est liée à la précision et à la rapidité du traitement :

- **haarcascade** est plus précis mais prend également bien plus de ressources et de temps. Si la machine est limitée en puissance ou s'il y a beaucoup de données à traiter, il vaut mieux ne pas faire ce choix.
- **lbpcascade** est moins précis mais aussi beaucoup moins gourmand. A privilégier quand on souhaite économiser des ressources, d'autant que je n'ai pas constaté de gain réel lors des tests avec le fichier **haarcascade** par rapport à **lbpcascade**.

```
    def prepare_camera(self, capture_device):
        self.camera = cv2.VideoCapture(capture_device)
        self.classifier = cv2.CascadeClassifier(TRAINSET)

    def is_camera_available(self):
        if self.camera.isOpened(): # try to get the first frame
            rval, frame = self.camera.read()
            return rval
        else:
            rval = False
            return False
```

Créons maintenant la fonction qui va détecter les visages :

```
    def get_faces(self, frame):
```

Tout d'abord, on diminue l'image originale avec le ratio précédemment défini. Ceci est fait pour simplifier le traitement et optimiser les ressources.

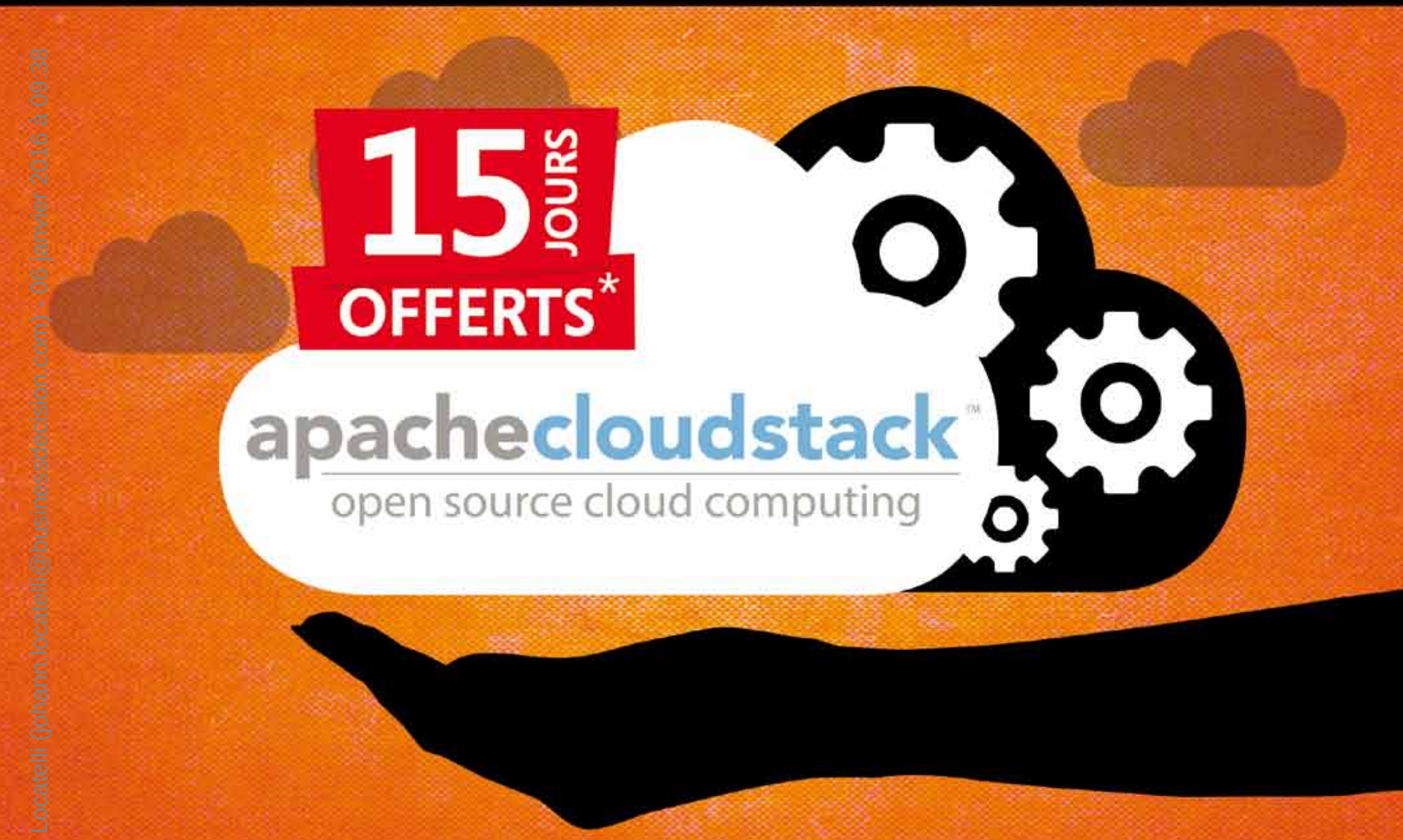
```
        minisize = (frame.shape[1]/DOWNSCALE,frame.shape[0]/DOWNSCALE)
        miniframe = cv2.resize(frame, minisize)
```

Ensuite, on applique simplement le **CascadeClassifier** créé à l'initialisation de la caméra et c'est au final la seule ligne qui permet la détection des visages ! Pour finir on renvoie la liste des visages détectés : il s'agit en fait d'une liste de tuples contenant la position et la taille du visage.

```
        faces = self.classifier.detectMultiScale(miniframe)
        return faces
```

Passons à la capture des images et à un affichage visuel des visages. Nous avons basiquement repris la précédente

DÉPLOYEZ VOS PROJETS OPEN SOURCE
avec CloudStack™ Instances operated by Ikoula



PLATEFORME PACKAGÉE
Ressources utilisables à l'heure, sans limitation : routeur, load balancer, firewall, IP.

CLOUDSTACK INSTANCES
Déployez votre Cloud Public avec des instances et des ressources à la demande.

APIs OUVERTES
Bénéficiez d'une interopérabilité avec des infrastructures existantes sans surcoût.

INTERFACE UNIQUE
Gérez votre réseau de Cloud Public en self-service.

EN SAVOIR PLUS SUR : http://express.ikoula.com/cloudstack

*Test valable, sans engagement de durée, une fois par compte client pour les 15 premiers jours à partir de la date de création du compte CloudStack Instances. Passée cette période toute consommation vous sera facturée au tarif en vigueur.

ikoula Hosting Services

01 84 01 02 50
sales@ikoula.com

NOM DE DOMAINE | MESSAGERIE | HÉBERGEMENT | CERTIFICAT SSL | CLOUD | SERVEUR DÉDIÉ

fonction **capture** dans laquelle nous avons ajouté une temporisation entre les captures et l'appel à la fonction qui détecte les visages, suivi du dessin d'un rectangle autour de chaque visage :

```
    def capture(self):
        cv2.namedWindow("preview")
        i = 0
        resized = None
        rval = True
        while rval:
            time.sleep(self.interval)
            rval, frame = self.camera.read()
```

Ici, appelons la fonction qui va chercher les visages dans la frame capturée :

```
            faces = self.get_faces(frame)
```

Pour chaque visage trouvé, nous allons réappliquer le ratio **DOWNSCALE** pour récupérer les coordonnées et dimensions des visages dans la frame originale (rappelez vous que la détection a été faite sur une image réduite) :

```
            for f in faces:
                x, y, w, h = [ v*DOWNSCALE for v in f ]
```

Et nous dessinons un rectangle autour de chaque visage.

```
                cv2.rectangle(frame, (x,y), (x+w,y+h), (255,0,0))

            cv2.putText(frame, "Appuyer sur [ESC] pour quitter.", (5, 25),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0,0,0))
            cv2.imshow("preview", frame)


            key = cv2.waitKey(20)
            if key in [27, ord('Q'), ord('q')]: # exit on ESC
                break
```

Pour finir, nous allons instancier notre classe et appeler les différentes fonctions dans l'ordre qui va bien :

```
if __name__ == "__main__":
    D = Detection(interval = 0.1)
    D.prepare_camera(0)
    if D.is_camera_available():
        D.capture()
```

Voici un exemple de résultat obtenu : voir Figure 2.

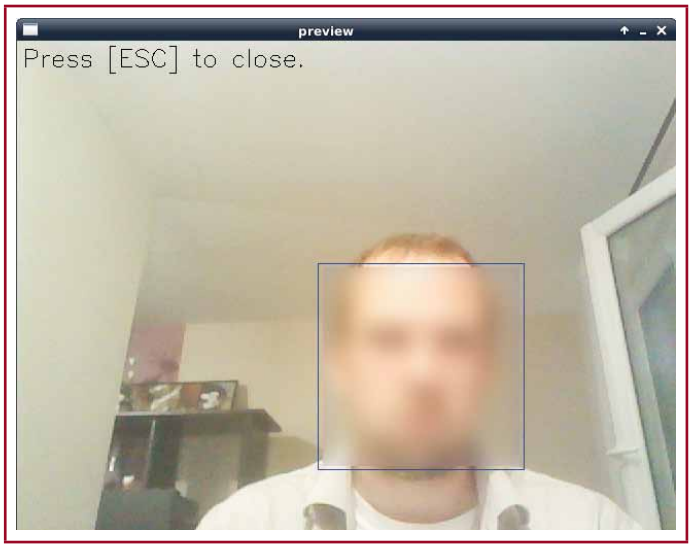


Fig. 2

6 Création d'une collection de visages

Pour identifier un visage, OpenCV aura besoin de ressources. Ces ressources s'appellent des collections et pour faire simple, il s'agit de multiples photos de chaque personne, la photo correspondant à l'intérieur du carré qui repère les visages. Ces photos doivent être de même taille et en niveaux de gris. Elles doivent également être classée dans une certaine arborescence.

Afin d'avoir une « bonne » collection, il faut respecter plusieurs règles :

- aligner les yeux sur une même ligne virtuelle pour toutes les photos,
- prendre des photos sous différentes luminosités,
- avoir des mimiques différentes : bouche ouverte ou non, yeux fermés ou non,
- ...

L'arobrescence devra avoir ce format :

- un dossier « racine »,
- un dossier par personne à identifier dans le dossier racine,
- dans chaque dossier de personnes, N photos nommées ainsi : **1.jpg**, **2.jpg**, **3.jpg**, ...

Ce qui donnerait avec les personnes **fred** et **fany** :

```
images/
images/fred/
images/fred/1.jpg
images/fred/2.jpg
images/fred/3.jpg
images/fred/...
images/fany/
images/fany/1.jpg
images/fany/2.jpg
images/fany/3.jpg
images/fany/...
```

Afin de créer cette collection, nous allons simplement reprendre la classe précédente, ajouter l'appel d'une fonction lors d'un appui sur la touche « c » et, si l'appui est détecté, enregistrer le visage dans la bonne résolution dans le dossier qui va bien avec le numéro qui va bien.

Voyons ce que donne cette nouvelle classe avec en rouge les ajouts/modifications :

```
#!/usr/bin/env python2

import cv2
import numpy
import time
import os

TRAINSET = "/usr/share/opencv/lbpcascades/lbpcascade_frontalface.xml"
DOWNSCALE = 4
```

Définissons la taille des images que nous allons capturer (la hauteur et la largeur auront la même valeur) et le nombre d'image à capturer.

```
IMAGE_SIZE = 170
NUMBER_OF_CAPTURE = 10

class Collection():
```

Notre classe a le droit a un paramètre supplémentaire qui sera le chemin où seront stockées les collections.

```
    def __init__(self, images_path, interval = 0.1):
        self.images_path = images_path
        self.camera = None
        self.classifier = None
        self.interval = interval

    def prepare_camera(self, capture_device):
        self.camera = cv2.VideoCapture(capture_device)
        self.classifier = cv2.CascadeClassifier(TRAINSET)

    def is_camera_available(self):
        if self.camera.isOpened(): # try to get the first frame
            rval, frame = self.camera.read()
            return rval
        else:
            rval = False
            return False


    def get_faces(self, frame):
        minisize = (frame.shape[1]/DOWNSCALE,frame.shape[0]/
DOWNSCALE)
        miniframe = cv2.resize(frame, minisize)
        faces = self.classifier.detectMultiScale(miniframe)
        return faces


    def capture(self):
        cv2.namedWindow("preview")
        i = 0
```

Nous initialisons le numéra de la capture en cours et le tableau qui contiendra les données des images capturées.

```
        capture_num = 1
        captures = []
        resized = None
        rval = True
        while rval:
            time.sleep(self.interval)
            rval, frame = self.camera.read()

            faces = self.get_faces(frame)
            for f in faces:
                x, y, w, h = [ v*DOWNSCALE for v in f ]
                cv2.rectangle(frame, (x,y), (x+w,y+h), (255,0,0))
```

Nous récupérons les données du visage détecté et mis à la bonne taille.

```
                resized = self.extract_and_resize(frame, x, y, w, h)

            cv2.putText(frame, "Press [ESC] to close.", (5, 25),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0,0,0))
```

Affichons un texte pour indiquer à l'utilisateur qu'il doit appuyer sur [c] pour prendre une capture.

```
            cv2.putText(frame, "Press [C] to collect the photo.", (5, 50),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0,0,0))
            cv2.imshow("preview", frame)


            key = cv2.waitKey(20)
            if key in [27, ord('Q'), ord('q')]: # exit on ESC
                break
```

Et si l'utilisateur appuie sur [c], la dernière image de visage (déjà redimensionnée) sera affichée dans une seconde fenêtre et ajoutée à la liste des images capturées. Si le nombre de captures est atteind, on sort de la fonction de capture vidéo en donnant en valeur de retour le tableau contenant tous les visages capturés.

```
            if key in [ord('c'), ord('C')]:
                if resized != None:
                    cv2.imshow("capture", resized)
                    captures.append(resized)
                    if capture_num >= NUMBER_OF_CAPTURE:
                        return captures
                    capture_num += 1
```

La fonction **`extract`** prend en entrée une image capturée par la caméra et les coordonnées d'un visage. Elle va extraire le visage, convertir l'extrait en niveaux de gris et redimensionner l'image du visage aux bonnes dimensions.

```
    def extract_and_resize(self, frame, x, y, w, h):
        cropped = cv2.getRectSubPix(frame, (w, h), (x + w / 2, y + h / 2))
        grayscale = cv2.cvtColor(cropped, cv2.COLOR_BGR2GRAY)
        resized = cv2.resize(grayscale, (IMAGE_SIZE, IMAGE_SIZE))
        return resized
```

La fonction **`add_to_collection`** sera appelée tout à la fin, une fois que toutes les captures seront faites. Elle va créer le dossier concernant l'identité capturée et y stocker les images des visages capturées.

```
    def add_to_collection(self, identity, images):
        os.makedirs("{0}/{1}".format(self.images_path, identity))
        idx = 1
        for img in images:
            cv2.imwrite("{0}/{1}/{2}.jpg".format(self.images_path,
identity, idx), img)
            idx += 1
```

Pour finir, on instancie la classe et on appelle ses fonctions dans l'ordre approprié.

```
if __name__ == "__main__":
    C = Collection(images_path = "./images/", interval = 0.1)
    C.prepare_camera(0)
    if C.is_camera_available():
        captures = C.capture()
        C.add_to_collection("toto", captures)
```

7 Reconnaissance faciale

Il nous reste maintenant à gérer la reconnaissance des visages, ce qui au final va s'avérer très simple ! Tout d'abord, il va falloir créer un modèle puis l'entraîner avec une collection de visages. Pour finir, pour chaque visage détecté il faudra demander une prédiction au modèle. Cette prédiction retournera 2 éléments : l'identité reconnue et un indice de confiance dans la prédiction.

7.1. Choisir le bon modèle

Il existe 3 modèles pour la reconnaissance de visages avec OpenCV :

- *eigenfaces* **[EigenFaces]**
- *fisherfaces* **[FisherFaces]**
- *local binary patterns histograms* **[LBPH]**

Chacun de ces modèles s'utilise de la même manière et il est donc très simple de remplacer le modèle à utiliser par un autre ! Je dois malheureusement avouer que les concepts mathématiques derrière ces modèles me dépassent... La documentation d'OpenCV est plus qu'abondante sur le sujet et je vous invite à la lire afin de vous aider à faire votre choix. Pour ma part, j'ai choisis le modèle « *local binary patterns histograms* » qui ne tient pas compte de la luminosité de l'environnement et qui marchait le mieux lors de mes tests.

Voyons à quoi va ressembler notre classe au final :

```
#!/usr/bin/env python2
import cv2
```

On va importer la librairie **sys** en plus : elle nous sera utile pour parcourir la collection de visages.

```
import sys
import os
import numpy
import time

TRAINSET = "/usr/share/opencv/lbpcascades/lbpcascade_frontalface.xml"
DOWNSCALE = 4

IMAGE_SIZE = 170
NUMBER_OF_CAPTURE = 10

class Recognize():

    def __init__(self, images_path, interval = 0.1):
        self.images_path = images_path
```

On initialise la liste des identités, des images :

```
        self.identities = []
        self.images = []    # X
        self.images_index = []    # Y
        self.camera = None
        self.classifier = None
        self.interval = interval
```

La fonction **read_images** va lire la collection des visages. Pour chaque dossier détecté, une entrée avec le nom du dossier (qui correspond au nom de la personne : fred, fany, ...) est ajoutée dans **self.identities**. Ensuite, pour chaque fichier du dossier, les données de l'image sont lues, converties en niveaux de gris puis redimensionnées à la taille nécessaire s'il le faut (dans la pratique, cette fonctionnalité n'est ici pas utilisée mais je la garde pour un éventuel besoin futur). En cas d'erreur, on affichera une erreur sur la sortie standard (à remplacer bien sûr par des appels à des fonctions de logs lors de l'inclusion dans un projet).

```
    def read_images(self, sz=None):
```

Tout d'abord on réinitialise les valeurs de l'objet en cas de multiples appels à la fonction :

```
        c = 0
        self.images = []
        self.images_index = []
```

On parcourt la liste des dossiers de la collection :

```
        for dirname, dirnames, filenames in os.walk(self.images_path):
            for subdirname in dirnames:
```

Pour chaque identité, on va....

```
                self.identities.append(subdirname)
                subject_path = os.path.join(dirname, subdirname)
```

... lire chaque fichier et....

```
                for filename in os.listdir(subject_path):
                    try:
```

... convertir les données en niveaux de gris et, si demandé, les redimensionner...

```
                        im = cv2.imread(os.path.join(subject_path,
filename), cv2.IMREAD_GRAYSCALE)
                        if (sz is not None):
                            im = cv2.resize(im, sz)
```

... puis stocker ces données transformées dans le tableau des images...

```
                        self.images.append(numpy.asarray(im, dtype=numpy.uint8))
```

... et, dans le tableau qui donne les index des images, on stocke l'index de l'identité...

```
                        self.images_index.append(c)
                    except IOError, (errno, strerror):
                        print "I/O error({0}): {1}".format(errno, strerror)
                    except:
                        print "Unexpected error:", sys.exc_info()[0]
                        raise
                c = c+1

    def prepare_camera(self, capture_device):
        ....
```

```
def is_camera_available(self):
    ...

def get_faces(self, frame):
    ...
```

La fonction capturé a le droit à de nouveaux paramètres :

- **mode** : ce paramètre peut prendre 2 valeurs : **acquire** pour le mode acquisition de visages et **identify** pour la reconnaissance faciale.
- **callback** : il s'agit de la référence vers une fonction de rappel. En mode **identify** la fonction de rappel devra accepter 2 paramètres : une chaîne de caractères qui sera le label de l'identité trouvée et un nombre flottant qui sera l'indice de confiance. La fonction de rappel n'est pas utilisée en mode **acquire**.
- **view** : ce paramètre permet d'afficher (valeur **True**) ou non la fenêtre de visualisation de la capture (qui est inutile en mode **identify** lorsqu'on ne fait plus de debug).

```
def capture(self, mode, callback = None, view = True):
    if view:
        cv2.namedWindow("preview")
    i = 0
    capture_num = 1
    captures = []
    resized = None
    rval = True
    while rval:
        time.sleep(self.interval)
        rval, frame = self.camera.read()
        faces = self.get_faces(frame)
        for f in faces:
            x, y, w, h = [ v*DOWNSCALE for v in f ]
            cv2.rectangle(frame, (x,y), (x+w,y+h), (255,0,0))
            resized = self.extract_and_resize(frame, x, y, w, h)
```

En mode **identify**, pour chaque visage détecter on appelle la fonction **identify** qui nous retourne le libellé de l'identité et un indice de confiance. On affiche ce libellé au dessus du rectangle qui entoure le visage à l'écran et on finit par appeler la fonction de rappel.

```
            if mode == 'identify':
                identity, confidence = self.identify(resized)
                cv2.putText(frame, "%s (%s)" % (identity, confidence), (x, y),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255,0,0))
                callback(identity, confidence)

        cv2.putText(frame, "Press [ESC] to close.", (5, 25),
                cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0,0,0))
        if mode == 'acquire':
            cv2.putText(frame, "Press [C] to collect the photo.", (5, 50),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0,0,0))
        if view:
            cv2.imshow("preview", frame)


        key = cv2.waitKey(20)
        if key in [27, ord('Q'), ord('q')]: # exit on ESC
            break
```

L'ajout de visages à la collection n'est possible qu'en mode **acquire**.

```
            if mode == 'acquire':
                if key in [ord('c'), ord('C')]:
                    if resized != None:
                        cv2.imshow("capture", resized)
                        #cv2.imwrite("test_%s.jpg" % (capture_num), resized)
                        captures.append(resized)
                        if capture_num >= NUMBER_OF_CAPTURE:
                            return captures
                        capture_num += 1

    def extract_and_resize(self, frame, x, y, w, h):
        ...
```

La fonction **train** va créer un modèle (notez ici pour l'exemple 2 exemples de création de modèle dont un en commentaire et entraînez-le avec notre collection de visages.

```
    def train(self):
        #self.model = cv2.createFisherFaceRecognizer()
        self.model = cv2.createLBPHFaceRecognizer()
        self.model.train(numpy.asarray(self.images), numpy.
asarray(self.images_index))
```

La fonction **identify** prend en paramètre les données d'un visage détecté et va demander une prédiction au modèle. Une fois l'index du visage récupéré, on récupère le libellé de l'identité et on retourne le libellé de l'identité ainsi que l'indice de confiance.

```
    def identify(self, image):
            [p_index, p_confidence] = self.model.predict(image)
        found_identity = self.identities[p_index]
        return found_identity, p_confidence

    def add_to_collection(self, identity, images):
        ...
```

La fonction **recognize** permet de lancer la reconnaissance des visages. Tout d'abord on lit la collection d'images, puis on entraîne le modèle, on prépare la caméra, on vérifie si elle est bien accessible et on lance la capture en mode **identify** en indiquant la méthode de callback et l'affichage (ou non) de l'écran de debug.

```
    def recognize(self, camera = 0, callback = None, view = True):
        self.read_images()    # rename function as collection ?
        self.train()

        self.prepare_camera(camera)
        if not self.is_camera_available():
            return
        self.capture(mode='identify', callback=callback, view = view)
```

La fonction **acquire**, quant à elle va lancer la capture afin de permettre l'ajout d'une personne (de libellé **identity**) à la collection. Après avoir préparé et vérifier l'accès à la caméra, on lance la capture en mode **acquire**.

Une fois la capture finie, on récupère un tableau d'images (**captures**) que l'on va écrire dans la collection d'images.

```
    def acquire(self, identity, camera = 0):
        self.prepare_camera(0)
        if not self.is_camera_available():
            return
        captures = self.capture(mode='acquire')
        self.add_to_collection(identity, captures)
```

Maintenant créons, en dehors de notre classe, une fonction de callback de test qui affichera l'identité et l'indice de confiance.

```
def display_recognize(identity, confidence):
    print("identity = {0} (confidence = {1})".format(identity, confidence))
```

Nous pouvons maintenant instancier notre classe et l'utiliser en mode ajout de visages à une collection :

```
R = Recognize(images_path = "./images/", interval = 0.1)
R.acquire(camera = 0, identity = "fred")
R.acquire(camera = 0, identity = "fany")
```

Puis tester la reconnaissance des visages :

```
R.recognize(camera = 0, callback = display_recognize, view = True)
```

Et voilà, nous avons une classe basique (mais suffisante) afin de gérer la reconnaissance des visages !

Conclusion

Et voilà, si on excepte les concepts mathématiques assez complexes, il est relativement simple au final de faire de la reconnaissance de visages... Attention toutefois à créer une collection « propre », à savoir avec les yeux sur la même ligne pour les visages, des images sous différentes luminosités avec les yeux fermés, ouverts, différentes positions de la bouche (fermée, ouverte), etc. De la qualité de votre collection dépendra la qualité de l'identification ! ■

Références

[OpenCV] http://docs.opencv.org/modules/contrib/doc/facerec/facerec_tutorial.html

[Cascade classification] http://docs.opencv.org/modules/objdetect/doc/cascade_classification.html

[EigenFaces] http://docs.opencv.org/modules/contrib/doc/facerec/facerec_api.html#createeigenfacerecognizer

[FisherFaces] http://docs.opencv.org/modules/contrib/doc/facerec/facerec_api.html#createfisherfacerecognizer

[LBPH] http://docs.opencv.org/modules/contrib/doc/facerec/facerec_api.html#createlbphfacerecognizer

ET SI VOUS PASSIEZ À LA PUISSANCE CLOUD ?

VAR

Distinguez-vous de vos concurrents en valorisant votre offre.

SSII

Profitez d'infrastructures IaaS déployées en un clin d'œil.

ÉDITEURS

Passez au SaaS en vous appuyant sur notre savoir-faire.

PROGRAMME PARTENAIRE ARUBA CLOUD

Contactez-nous

- 2 niveaux de marque blanche disponibles.
- Gestion simple, souple et performante de l'infrastructure (publique, privée ou hybride).
- Modélisation et activation immédiate de votre datacenter virtuel dans le pays de votre choix.
- Interface client totalement personnalisable.
- Facturation en " Pay as you go ".
- Finesse dans la gestion des droits utilisateurs.

Aruba, le bon partenaire pour bénéficier de la puissance d'un acteur majeur qui considère que chaque client, dans chaque pays, est unique. **MY COUNTRY. MY CLOUD.**

aruba CLOUD

arubacloud.fr | TÉL : 0810 710 300
(COÛT D'UN APPEL LOCAL)

LINAGORA EST FIÈRE D'ÊTRE, GRÂCE À SES CLIENTS, L'UN DES PRINCIPAUX ÉDITEURS DES LOGICIELS LIBRES FRANÇAIS !

MERCI !

LINAGORA

www.linagora.com

www.obm.org