N°170 AVRIL 2014

GNU LINUX MAGAZINE / FRANCE

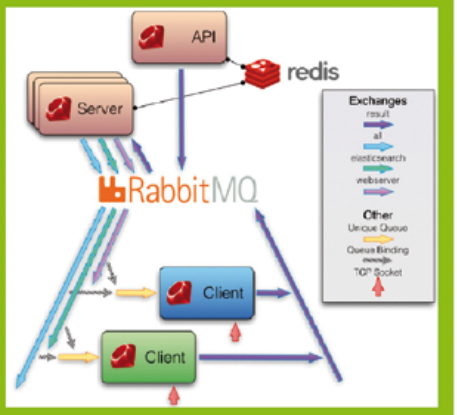
MOBILITÉ / ACTU

62% des smartphones Android sous Jelly Bean : les apports de cette version p.04

ADMINISTRATION ET DÉVELOPPEMENT SUR SYSTÈMES OPEN SOURCE ET EMBARQUÉS

RÉSEAU / SUPERVISION

Découvrez Sensu, un logiciel de surveillance système et réseau p.66



RETOUR / EXPÉRIENCE

Des pistes de réflexion pour optimiser un projet libre p.38

CODE / PYTHON

Moins verbeux que XML, utilisez le format YAML pour stocker des données p.80

L 19275 - 170 - F: 7,90 € - RD

SYSADMIN / ANALYSE

Marre de rebooter vos serveurs sans comprendre pourquoi ils ont planté ? Essayez Collectd et PerfWatcher p.50

VOIP / SIP

Installez votre premier serveur

ASTERISK

ET FILTREZ LES TÉLÉVENDEURS ! p.28

1. Mise en place du réseau exemple
2. Configuration de base
3. Configuration avancée
4. Routage SIP

SGBD / POSTGRESQL

Le planificateur de requêtes : comprenez pourquoi certaines requêtes sont plus lentes que d'autres p.16

HUMEUR / CODE

Pour le Dr KissCool une chose est sûre : la documentation de code est une pure perte de temps ! p.48

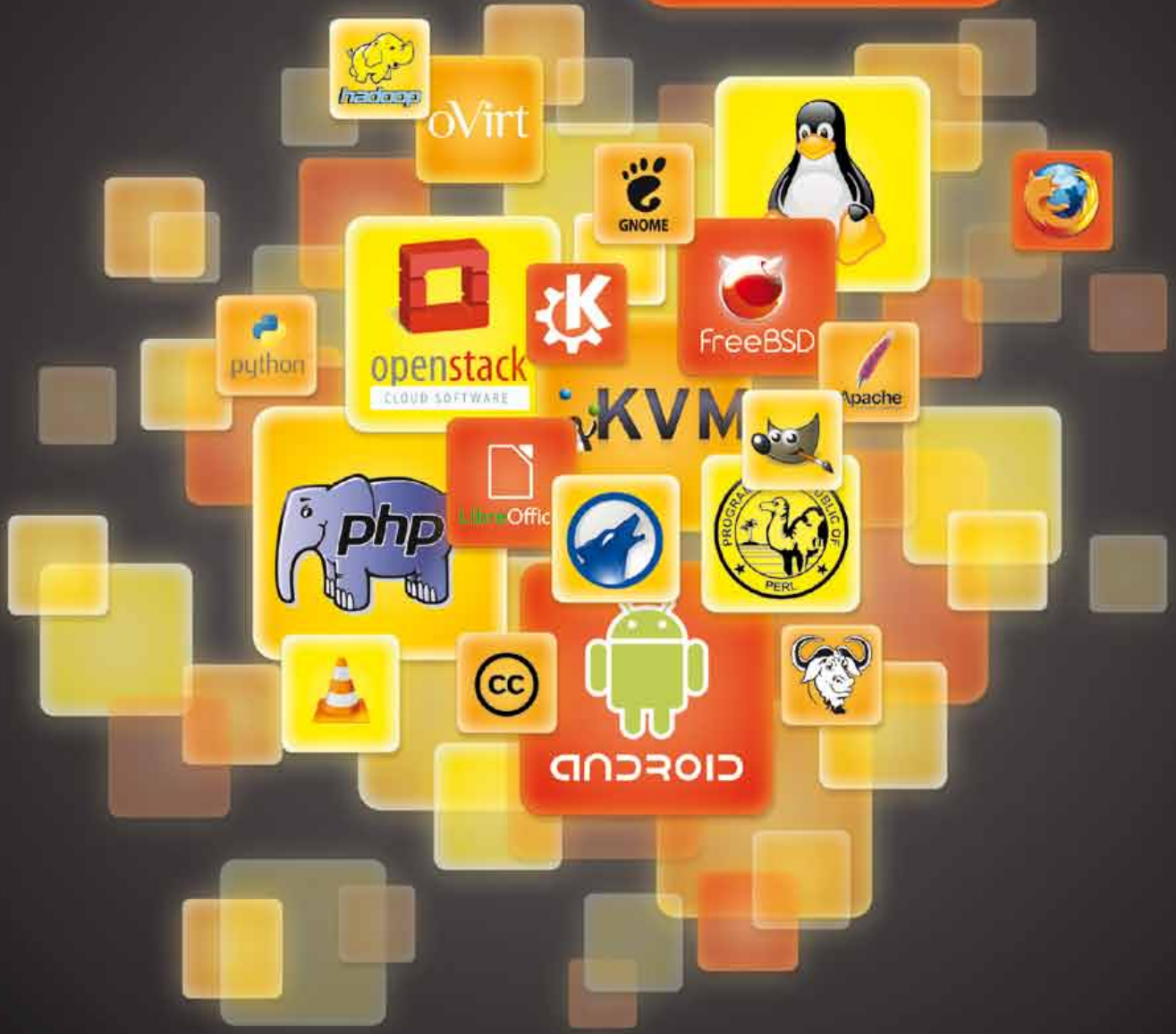
France METRO : 7,90 € – CH : 13 CHF – BEL/PORT.CONT : 8,90 € – DOM TOM : 8,50 € – CAN : 14 $ cad – TUNISIE : 18 TND – MAR : 100 MAD

LINUX Solutions Libres & Open Source

Le salon dédié à linux et aux logiciels libres

20&21 MAI2014

CNIT - Paris La Défense



Toutes les solutions et nouveautés informatiques en Open Source...
Pour encore plus de libre au service de l'entreprise !

Un événement Tarsus GROUPE MÉDIA B TO B

Partenaire officiel monannuaire pro.com

www.solutionslinux.fr

SOMMAIRE N°170

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
Rédacteur en chef adjoint : Tristan Colombo
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

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 €

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

Membre April www.april.org

ÉDITORIAL

170 numéros et des kilomètres d'articles plus tard...

En septembre 1998 paraissait le premier numéro de *Linux/Magazine France* (on avait oublié le « GNU » au début, honte à nous). Trimestriel, bimestriel, puis mensuel dès le 3ème numéro, ce que certains appellent LinuxMag et d'autres GLMF a traversé les années et vu la démocratisation de GNU/Linux et du logiciel libre, tout comme je l'ai fait moi-même. GNU/Linux a évolué, le monde de l'informatique personnelle et professionnelle a évolué, le magazine a évolué, vous lecteurs avez évolué et moi aussi j'ai évolué. L'impossible est arrivé parce que des développeurs, des administrateurs et des utilisateurs ne savaient pas que c'était impossible. Et maintenant, GNU/Linux est omniprésent sur Internet, mais aussi en téléphonie, dans votre poche, dans votre routeur, dans votre téléviseur, votre box... Tout autour de nous. On l'espérait, on le désirait, on le voulait il y a plus de 15 ans et c'est bel et bien arrivé ! Même si la présence de GNU/Linux sur le desktop de M. Tout-le-monde reste très modeste, quel impressionnant parcours !

Aujourd'hui, des choses se font jour, des perspectives apparaissent, d'autres chemins se dévoilent. Les motivations, les sentiments et l'obsession qui ont fait GNU, Linux et tout l'écosystème du logiciel libre, toujours présents en son sein, dépassent désormais ses limites. Nous sommes à l'aube d'un changement de paradigme majeur dans nos relations avec la technologie. Le changement est dans l'air, on le ressent, on l'attend...

Le changement est bon, tout autant que le renouveau. Ceci est valable tant à titre personnel que pour un mouvement comme le logiciel libre, mais également pour une publication comme celle que vous tenez entre vos mains. Ainsi, le renouveau prendra forme dans GLMF avec une nouvelle version intégrant les fruits d'une réflexion interne tant au niveau rédactionnel que dans sa concrétisation matérielle, mais aussi et surtout, avec un changement de rédacteur en chef.

Je vais, en effet, prendre du recul vis-à-vis de GLMF en transférant les rênes à quelqu'un que vous connaissez déjà, car contributeur de longue date, j'ai nommé Tristan Colombo. Je vous laisse entre de bonnes mains, car c'est avec une grande confiance que je lui confie ainsi la tâche d'apporter sa vision, sa tonalité et une fraîcheur nouvelle au cœur du magazine et ce, dès le prochain numéro. En ce qui me concerne, je ne serai guère loin, puisque toujours chef des rédactions, rédacteur en chef d'*Open Silicium* et en train de travailler à un nouveau projet non sans rapport avec le changement de paradigme dont je vous parlais plus haut... Nous en dirons davantage sur ce point d'ici peu.

Pour conclure, puisqu'il s'agit donc de mon dernier édito dans ce magazine, je voudrais remercier l'ensemble des contributeurs de ces 170 numéros, ainsi que des 71 hors-séries, mais également vous lecteurs sans qui ce voyage n'aurait pas eu lieu et n'aurait pas eu de sens. Ce fut un plaisir aussi enrichissant que satisfaisant. Je vous souhaite une bonne lecture pour ce numéro et tous ceux qui suivront, en vous disant « à bientôt » pour de nouvelles aventures !

Denis Bodor

INSIDE ANDROID : JELLY BEAN (4.1 À 4.3)

par Benjamin Zores, architecte Android @ Alcatel-Lucent

À la manière du désormais habituel « Kernel Corner » et pour faire suite à la série « À la découverte d'Android », je vous propose de continuer cette plongée dans les entrailles du système de Google par cette nouvelle série « Inside Android », qui présentera les changements introduits par chaque nouvelle version de l'OS.

Les plus fidèles lecteurs s'en souviendront certainement : le dernier volume mettait en avant le système de mise à jour OTA et la voie ouverte vers la migration d'Ice Cream Sandwich (4.0) vers Jelly Bean (4.1). Mon plan à l'époque était donc de vous présenter Android 4.1 (disponible depuis juillet 2012). Mais depuis, Google a eu la bonne idée de sortir la 4.2 (novembre 2012) et 4.3 (juillet 2013). Ces trois versions appartiennent à la même famille : Jelly Bean. Google a d'ailleurs officialisé Android 4.4 (Kit Kat) en novembre 2013, mais ce sera pour le prochain numéro :-)

Voyons donc ce que cette version Jelly Bean a dans le ventre et quel est son impact. Le système étant désormais « ancien » (bien qu'une très grosse majorité du parc de téléphones et autres tablettes n'ait pas encore migré dessus), je vous propose donc un tour d'horizon des trois versions à la fois. Et comme à l'accoutumée, mon but est de vous présenter les changements radicaux au sein des mécanismes internes. Loin de moi l'idée de vous expliquer les changements d'API applicative pour le développeur et l'utilisateur final, vous trouverez très facilement toute la documentation souhaitée sur ces sujets. Enfin, statistique intéressante : si Ice Cream Sandwich totalisait **5,5 Go de sources pour 230 composants**, Jelly Bean 4.3 en compte désormais **8 Go pour 361 composants**. Les choses grossissent à vue de nez...

1 Petits rappels...

Avant d'aller plus loin, procédons à quelques rappels quant à l'architecture d'Android, telle que le présente la figure 1.

Android est un système d'exploitation basé sur le noyau Linux et propose un framework applicatif et un ensemble d'API écrites en Java. Si le code des applications elles-mêmes est bien écrit en langage Java, il est compilé et transformé de telle sorte que du byte-code Dalvik soit en fait actif sur votre téléphone. Entre ces deux mondes bien distincts que sont le noyau et l'applicatif Java, résident un certain nombre de librairies, *daemons* système et autres couches d'abstraction qui permettent de créer le lien.

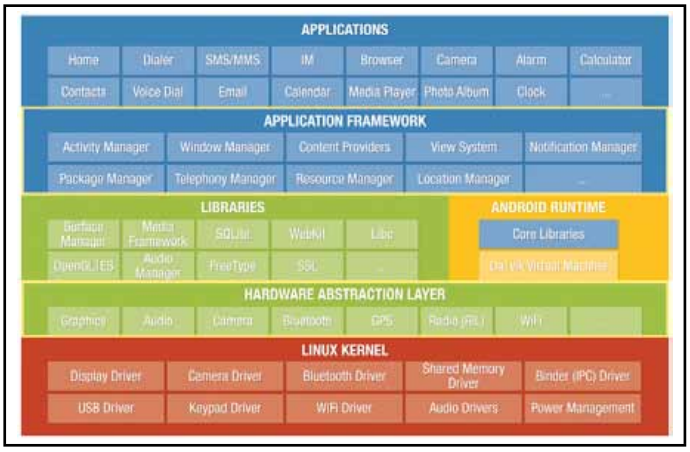


Fig. 1 : Architecture logicielle d'Android

De plus, et afin de permettre à tous les constructeurs (disposant de spécificités propres à leur matériel) d'être compatibles avec le framework applicatif universel d'Android, nous trouverons un middleware appelé HAL (*Hardware Abstraction Layer*), qui permet cette abstraction. En tant que développeur système, concepteur de téléphones/tablettes, ou autre ingénieur de systèmes embarqués, c'est cette HAL qu'il va falloir développer pour rendre votre produit compatible avec Android. Cette HAL est le bien souvent très peu documentée par Google et c'est pourtant au sein de cette dernière que se produisent les changements les plus radicaux lors d'une montée en version. C'est donc principalement autour de cette dernière (et d'une manière générale, sur les couches basses ou « système ») que nous allons nous concentrer.

2 Graphisme et « projet Butter »

Un des grands reproches fait à Android à partir d'ICS (4.0) consiste en sa relative « lenteur » ou manque de réactivité par rapport à la compétition (i.e. iPhone et iPad d'Apple). Google a donc lancé le projet Butter (ou « beurre »), afin de fluidifier quelque peu son système mobile, projet qui a été intégré à Jelly Bean (JB). Les résultats se qualifient en trois termes : plus rapide, plus fluide, plus réactif. Ceci a été rendu possible grâce aux ajouts suivants :

- **Mise en tampon triple** (*triple buffering*) en lieu et place d'une anciennement double, améliorant sensiblement la coordination et la synchronisation des animations entre le CPU, le GPU et l'écran d'affichage ;
- **Synchronisation Verticale** (VSYNC), permettant l'amélioration des performances d'affichage, empêchant le « déchirement » (*tearing*) d'une image à l'autre et augmentant la vitesse d'affichage de 30 à 60 images par seconde ;
- **Meilleure réactivité aux gestes du toucher**, permettant de prédire les actions à venir de l'utilisateur sur la dalle tactile, améliorant ainsi le temps de chargement des actions correspondantes.

Pour mieux comprendre les détails d'implémentation (et les changements induits) de chacune de ces fonctionnalités, je vous renvoie à la figure 2, qui a le mérite de présenter l'architecture graphique d'Android.

Pour résumer et simplifier les choses, Android a recours au composant logiciel **SurfaceFlinger** pour gérer la composition des différentes fenêtres et surfaces à l'écran. Ce dernier est logiciel et donc par nature, lent. S'il en a la possibilité, SurfaceFlinger va essayer de décharger ces opérations coûteuses au GPU, via la HAL et le Hardware Composer (si le périphérique le supporte). De nombreuses améliorations ont été apportées à l'unité de rendu 2D accélérée par le matériel, afin d'optimiser le rendu d'opérations de dessins « classiques » (remplissage de lignes, cercles, ovales, rectangles...). Enfin, le processus de rendu devient multi-threadé et peut désormais utiliser de multiples cœurs CPU pour découpler certaines tâches (même si les opérations peuvent ensuite être faites par le GPU).

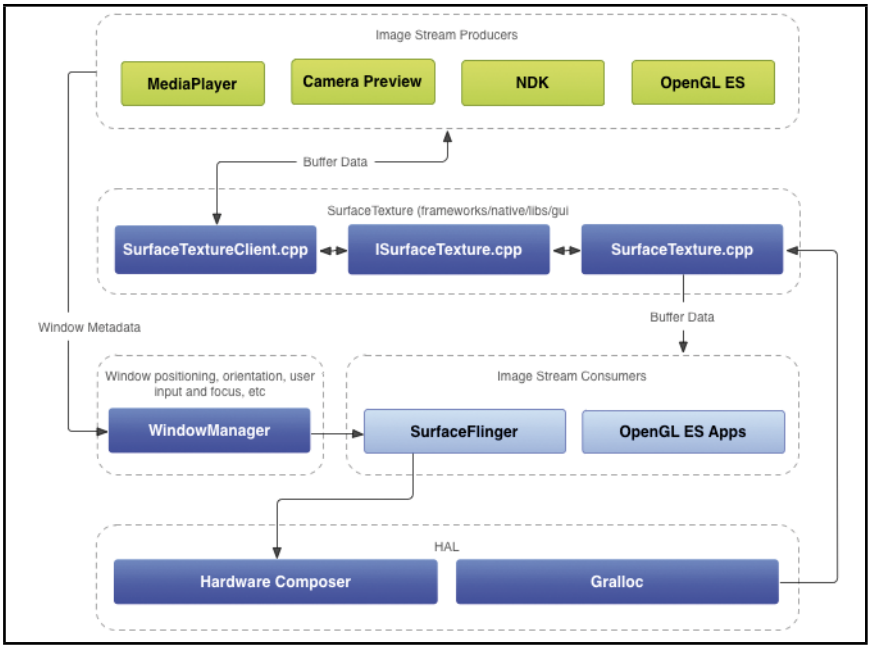


Fig. 2 : Architecture graphique d'Android

Au niveau de notre HAL, l'impact est réel et une nouvelle API a été mise en place. On passe donc de la HAL hwcomposer 1.0 à la 1.1 (stable), voire à la 1.2 (encore en *draft*). Les détails se situent au sein du fichier **`hardware/libhardware/include/hardware/hwcomposer.h`**. Ainsi, deux nouveaux layers font leur apparition : **`HWC_BACKGROUND`** et **`HWC_FRAMEBUFFER_TARGET`**. Le premier est appelé avant l'appel à la fonction **`prepare()`** pour assigner une couleur de fond de surface. Le second indique que ce layer constituera la surface du framebuffer utilisée comme cible de la composition OpenGL ES.

2.1 Synchronisation verticale

La fonctionnalité de VSYNC permet de synchroniser certains événements aux cycles de rafraîchissement de l'écran. Il n'y a rien de plus frustrant en effet que de voir l'écran se déchirer (sentiment perceptible de ralentissement), car le GPU est en train de dessiner à l'écran pendant que ce dernier est encore en train de dessiner une partie de l'image précédente. De ce fait, et afin d'obtenir un rafraîchissement continu de 60 images par seconde, une plage de 16ms est attribuée au rafraîchissement de chaque image. Ces frontières en place, et cadencées par *tick* d'horloge, les applications commencent toujours à dessiner au début d'une plage VSYNC et SurfaceFlinger commence la composition au début de la suivante. Chacun dispose donc d'un délai maximum de 16ms et de par l'utilisation de triples buffers, il n'y a pas de distorsion d'image possible.

À vous donc d'implémenter (ou de mettre à jour) votre HAL hwcomposer pour le support de la fonction suivante :

```
void (*vsync)(const struct hwc_procs* procs,
int disp, int64_t timestamp);
```

qui sera appelée à chaque réception d'un événement VSYNC. Il est cependant impératif que les chronomètres (*timestamps*) de l'appelant et de l'appelé

soient synchrones. Un délai maximum de 1ms (bien que 0.5ms ou moins soit fortement recommandé) est autorisé.

S'il en est ainsi pour la composition et l'affichage de surfaces, il en est de même pour les tampons (*buffers*) utilisés pour la composition. Ces derniers (fournis par la HAL Gralloc) doivent pouvoir être acquis et relâchés de manière coordonnée avec VSYNC et requièrent donc une synchronisation explicite. Cette dernière permet aux producteurs et consommateurs de ressources graphiques de notifier lorsqu'ils en ont fini avec un buffer, permettant au système de mettre ces derniers dans une queue asynchrone, tout en garantissant que plus personne n'en a besoin.

Cette synchronisation explicite est permise par :

- l'utilisation d'un pilote noyau permettant la synchronisation temporelle pour un composant matériel donné, communiquant avec le compositeur matériel ;
- l'implémentation au niveau de la HAL v1.1 des mécanismes de synchronisation adéquats au sein des fonctions **`set()`** et **`prepare()`** ;
- la communication entre kernel-space et user-space au moyen de la nouvellement introduite **`libsync`** (cf. **`system/core/include/sync/sync.h`** et **`system/core/libsync`** pour davantage de détails).

La réactivité aux latences du toucher est quant à elle réduite de par la synchronisation des gestes avec les périodes de VSYNC. Mais elle l'est également par anticipation de l'endroit où votre doigt sera vraisemblablement au moment du prochain rafraîchissement d'écran (i.e. dans 16ms, au prochain VSYNC). Enfin, après une période de veille, le système procède également à une augmentation instantanée de la fréquence CPU, de manière à réduire la latence de prise en compte du prochain toucher.

2.2 Écrans multiples

Jelly Bean apporte également le support de multiples écrans. Si ICS permettait d'utiliser un écran LCD et un écran HDMI, l'OS imposait un mode miroir (ou clone). Le même contenu était donc affiché et la même résolution était appliquée entre les écrans. Les développeurs ont désormais la possibilité d'afficher du contenu différent sur chaque écran.

La HAL hwcomposer voit donc apparaître une nouvelle fonction **`hotplug()`** lorsqu'un nouvel écran est connecté ou déconnecté. À noter que l'écran principal est toujours connecté. La callback **`hotplug()`** ne sera donc pas appelée pour ce dernier.

```
void (*hotplug)(const struct hwc_procs* procs, int disp, int connected);
```

En outre, la HAL propose désormais la fonction **`getDisplayAttributes()`**, permettant d'interroger chaque écran pour en connaître ses propriétés. Cette dernière a été rendue indispensable de par la capacité même à pouvoir interchanger les écrans.

```
int (*getDisplayAttributes)(struct hwc_composer_device_1* dev, int
disp, uint32_t config, const uint32_t* attributes, int32_t* values);
```

Les différents attributs adressables sont les suivants :

- **`HWC_DISPLAY_VSYNC_PERIOD`** : la période de VSYNC (en nanosecondes) ;
- **`HWC_DISPLAY_WIDTH`** et **`HWC_DISPLAY_HEIGHT`** : la résolution de l'écran (en pixels) ;
- **`HWC_DISPLAY_DPI_X`** et **`HWC_DISPLAY_DPI_Y`** : le nombre de pixels par milliers de pouces (ceci permettant de stocker la densité de pixels sur un entier sans trop perdre en précision).

Enfin, une nouvelle fonction **`blank()`** voit le jour, permettant tout simplement d'allumer/éteindre l'écran :

```
int (*blank)(struct hwc_composer_device_1* dev, int disp, int blank);
```

De la même façon, la fonction **`prepare()`** se voit désormais modifiée pour pouvoir adresser différemment chacun des écrans :

```
int (*prepare)(struct hwc_composer_device_1 *dev, size_t numDisplays,
hwc_display_contents_1_t** displays);
```

Enfin, si la possibilité est désormais offerte aux applications d'interagir avec différents écrans, ces dernières ne le font pas directement via la HAL. Un nouveau service *Display Manager* a donc vu le jour, permettant de faire cette abstraction. Ce dernier permet ainsi de lister les écrans, récupérer leurs capacités, y instaurer un canal d'affichage vidéo sécurisé (comprendre chiffré), etc.

2.3 OpenGL ES 3.0

Enfin, Jelly Bean apporte également le support d'OpenGL ES 3.0, aussi bien pour les développeurs Java via le framework applicatif, que pour les développeurs utilisant le NDK. Ce dernier permet (entre autres, et uniquement pour les périphériques dont le GPU est compatible) d'accélérer le rendu 2D, mais également d'optimiser la gestion des textures ETC2/EAC et d'améliorer le rendu et la fidélité des gradients. L'API de ce dernier se retrouve au sein du répertoire **`frameworks/native/opengl/include/GLES3`**.

3 Périphériques de saisie

Jelly Bean apporte peu de changements en ce sens, ou du moins, rien d'intrusif (comme cela avait pu être le cas entre Gingerbread et ICS). La compatibilité des API est donc conservée, bien que de nouvelles fonctionnalités apparaissent. On

sent clairement qu'Android devient une plateforme pour le jeu vidéo. Les applications peuvent ainsi être notifiées de l'ajout de nouveaux contrôleurs (USB, Bluetooth...) ou de leur suppression. L'ajout d'un nouveau clavier ou d'un nouveau gamepad lors d'un jeu permettra ainsi à ce dernier de proposer une partie multijoueur. Il est enfin également possible de récupérer les informations propres à chaque périphérique, pour en connaître ses capacités et ainsi les contrôler (c'est le cas par exemple des manettes avec fonctions vibrantes).

4 Wi-Fi

Jelly Bean apporte ici une petite révolution fonctionnelle avec la prise en charge de la norme Miracast, également appelée *Wireless Display*, qui permet d'afficher le contenu de votre smartphone/tablette vers un écran compatible (nativement ou via un dongle USB compatible Miracast et branché en HDMI), le tout sans fil avec une qualité d'affichage « correcte ». S'il existe de nombreux récepteurs compatibles Miracast, il y a très peu d'émetteurs et Android fait figure de référence en ce sens. La connexion se fait au travers d'une liaison P2P et vous permet réellement de diffuser tout type de contenu.

Mais pour ce faire, en plus de l'implémentation du protocole Miracast en lui-même, plusieurs critères sont requis :

- Le chip radio (matériel) doit être compatible avec la norme Wi-Fi P2P (cela semble être une évidence) ;
- Le chip radio (matériel) doit être capable de supporter des connexions multiples (une traditionnelle, vers son point d'accès, l'autre en direct, vers l'écran Wi-Fi) ;
- La politique audio du fichier **audio_policy.conf** d'AudioFlinger doit contenir une règle **r_submix** pour le module de mixage audio distant. Ce module permet d'enregistrer et de diffuser le flux audio du système vers l'encodeur audio de l'écran Wi-Fi par le biais du MediaServer d'Android ;
- Le module de mixage audio **audio.r_submix.default** doit être présent sur le téléphone ;
- Optionnellement, le périphérique doit disposer de clés HDCP si l'on souhaite diffuser du contenu protégé par DRM.

Si tous ces critères sont réunis, il est alors possible de mettre à jour le fichier **frameworks/base/core/res/res/values/config.xml** pour y faire figurer la ligne suivante et ainsi autoriser le framework applicatif à utiliser la technologie Miracast :

```
<bool name="config_enableWifiDisplay">true</bool>
```

Ceci étant, et si vous avez bien suivi les deux premiers prérequis, Android doit désormais savoir se connecter à deux réseaux Wi-Fi simultanément. Le système qui ne supportait qu'une seule interface de connexion réseau, doit donc désormais en supporter au moins deux (deux interfaces et deux adresses IP différentes).

Ceci implique donc quelques changements sommaires au sein de la HAL Wi-Fi :

```
/* PRIMARY refers to the connection on the primary interface
 * SECONDARY refers to an optional connection on a p2p interface
 *
 * For concurrency, we only support one active p2p connection and
 * one active STA connection at a time
 */
#define PRIMARY     0
#define SECONDARY   1
#define MAX_CONNS   2
```

Et du prototype de quelques fonctions de base :

```
-int wifi_start_supplicant();
-int wifi_start_p2p_supplicant();
+int wifi_start_supplicant(int p2pSupported);
-int wifi_connect_to_supplicant();
+int wifi_connect_to_supplicant(const char *ifname);

-int wifi_command(const char *command, char *reply, size_t *reply_
len);
+int wifi_command(const char *iface, const char *command, char
*reply, size_t *reply_len);
```

Enfin, de nouvelles fonctionnalités ont également été introduites de par la mise à jour du composant WPA Supplicant de la version 0.8.x à la version 2.1-devel. Si vous êtes intéressé par les détails, je vous laisse les découvrir par la lecture attentive de la couche d'abstraction JNI au sein du fichier **frameworks/base/wifi/java/android/net/wifi/WifiNative.java**, qui redirige l'ensemble des requêtes du service Wi-Fi d'Android vers des commandes au daemon **wpa_supplicant**.

Mais si l'on souhaite en rester au pitch marketing, les évolutions majeures sont les suivantes :

- **Auto-découverte de services Wi-Fi P2P**. Introduite en 4.0 avec ICS, la fonctionnalité de P2P (utilisée par Miracast) permet une connexion point-à-point idéale pour partager simplement du contenu entre deux périphériques proches. Jelly Bean ajoute à cela des API de pré-association, permettant d'interroger le périphérique distant de ses capacités applicatives (en termes de services) et éventuellement de le filtrer, avant de le proposer à la connexion. La fonctionnalité se base sur les capacités NSD (*Name Server Daemon*) des différents périphériques, qui permettent ainsi l'affichage humainement compréhensible des services qu'ils exposent.
- **Support des réseaux WPA2-Entreprise**. Les applications sont désormais en mesure de configurer d'elles-mêmes les accréditations dont elles ont besoin pour se connecter à

ce type de réseaux. De nouvelles API voient le jour, permettant la configuration d'accréditations compatibles EAP (*Extensible Authentication Protocol*) et *Encapsulated* EAP Phase 2.

5 Bluetooth

La palme d'or du sous-système qui a été radicalement bousculé revient au Bluetooth. Et pour cause, ce dernier, comme tout système GNU/Linux qui se respecte, utilisait la stack BlueZ jusqu'à Android 4.1. Jelly Bean 4.2 a changé la donne en basculant vers une toute nouvelle stack introduite par Broadcom et prénommée **BlueDroid** (cf. `external/bluetooth/bluedroid`). Vous trouverez un schéma d'architecture de cette nouvelle solution au sein de la figure 3.

Compatibilité applicative oblige, le framework applicatif reste inchangé. Côté plateforme par contre, tout est à revoir. Ce changement a cependant été rendu nécessaire de par la nécessité de supporter davantage de profils Bluetooth et pour supporter le norme BTLE (*Bluetooth Low-Energy*), également appelée *Bluetooth Smart Ready*. Et si jusqu'à présent, le service Bluetooth d'Android faisait des appels JNI directs à BlueZ, il se comporte désormais comme les autres sous-systèmes et utilise une HAL (standardisée) qui elle-même repose sur une version (personnalisable) de la stack BlueDroid et d'éventuelles extensions propres à chaque constructeur de chip.

Un appel Bluetooth se fait donc au sein des cinq éléments présentés sur cette figure :

1. Une application utilise l'API `android.bluetooth` exposée par le framework applicatif, qui interagit avec le service Bluetooth au travers de requêtes IPC à Binder (le bus de messages d'Android).
2. Le service système Bluetooth (cf. `packages/apps/Bluetooth`) implémente le service et les différents profils au niveau applicatif et dialogue avec la couche d'abstraction HAL par JNI. Chaque profil supporté est implémenté par JNI au sein du fichier `packages/apps/Bluetooth/jni/com_android_bluetooth_{profile}.cpp`. En version 4.3, Android supporte les profils suivants : A2DP, AVRCP, GATT, HDP, HFP, HID et PAN.
3. La HAL mappe les appels Bluetooth applicatifs en requêtes spécifiques à chaque composant matériel. Le fichier d'entête `hardware/libhardware/include/hardware/bluetooth.h` décrit l'ensemble des fonctions en relation directe avec le matériel (mise sous/hors tension, appairage...). Les fichiers d'entête `hardware/libhardware/include/hardware/bt_{profile}.h` décrivent chacun un profil Bluetooth propre.
4. Les fonctions de la HAL sont ensuite implémentées par la stack BlueDroid elle-même. Google fournit donc une implémentation de référence, proposée par Broadcom, qui semble convenir à tout un chacun. Libre cependant à chaque constructeur de la modifier et de l'étendre.
5. Enfin, la possibilité est laissée à chaque constructeur d'ajouter ces extensions propriétaires pour le niveau de communication HCI par la création d'un composant `libbt-vendor` spécifique.

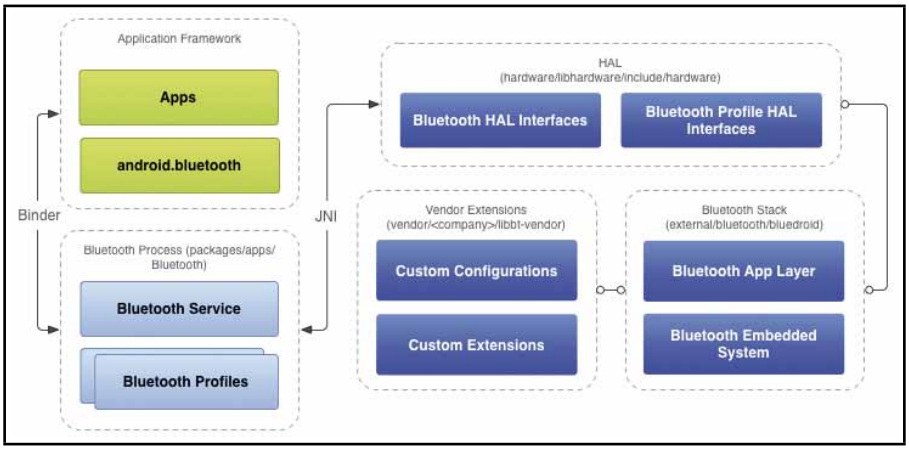


Fig. 3 : Architecture Bluetooth d'Android

Pour en revenir à BlueDroid, la stack est composée de deux couches principales :

- *Bluetooth Embedded System* (BTE), qui implémente les couches système ;
- *Bluetooth Application Layer* (BTA), qui communique avec le framework applicatif.

À noter que la stack propose sa propre implémentation du codec audio SBC utilisé par le profil A2DP pour diffuser du son.

Enfin, et c'est probablement une des raisons principales du changement de BlueZ vers BlueDroid, chaque constructeur a désormais la possibilité de personnaliser BlueDroid. BlueZ est disponible sous licence GPL et Android avait recours à D-BUS dans le seul et unique but de communiquer avec BlueZ. Aucun vendeur ne se risquait alors à implémenter le support de nouveaux profils, de peur de voir son code propriétaire « contaminé » par la GPL. C'est donc pour permettre aux constructeurs de conserver leur code propriétaire que BlueDroid est apparu, sous licence Apache. Les fonctionnalités et profils par défaut peuvent donc très facilement être étendus désormais.

Pour ce faire, et selon les cas, rien de plus simple :

- **Profils additionnels**. Pour cela, il vous est nécessaire d'ajouter les nouvelles API au sein de la stack BTE/BTA, de la HAL, mais également les

bindings JNI nécessaires au sein de l'application Bluetooth. Enfin, vous devrez proposer aux développeurs une extension au SDK leur permettant d'utiliser vos nouvelles API.

- **Extensions Vendeurs**. Si votre chip Bluetooth requiert des commandes AT supplémentaires pour sa programmation, ou encore une configuration particulière, il est possible de créer une librairie `libbt-vendor` qui sera exécutée en lieu et place de l'originale. Vous trouverez un exemple au sein du répertoire `hardware/broadcom/libbt`.
- **Extensions HCI**. Enfin, si vous souhaitez simplement étendre les fonctions HCI de communication avec votre chip, il vous suffit de créer un module `libbt-hci` qui sera exécuté à la place de l'original. Référez-vous au répertoire `external/bluetooth/bluedroid/hci` pour davantage de détails.

6 Near Field Communication (NFC)

La couche NFC de Jelly Bean a quant à elle subi une mise à jour mineure, bien que très pratique. Sur un aspect purement fonctionnel, Android 4.1 a introduit le support de Beam, une technologie permettant le partage instantané d'informations entre deux appareils NFC (passifs ou actifs). NFC est cependant une norme reposant sur des tags de très faible taille. Aussi, dans le cadre de transfert ou partage de fichiers (images, vidéos...), la technologie n'est utilisée que pour l'auto-découverte et l'association entre pairs. Une fois cette étape remplie, la couche NFC relaie le transfert des données à la couche Bluetooth.

Sous le capot, les changements sont un peu plus conséquents. Si vous vous souvenez bien, la couche NFC d'Android, bien qu'utilisant la HAL (couche d'abstraction matérielle), était très centrée sur les puces PN544 de chez NXP. Comme dans le cas du Bluetooth, le mérite en revient donc à Broadcom (en collaboration avec Google) d'introduire une nouvelle stack NFC (pour l'instant en complément de l'actuelle).

Android 4.0 proposait l'implémentation du NFC au travers de la librairie `external/libnfc-nxp`. Cette dernière propose le support des composants PN544 et PN65N au travers d'une couche HCI très spécifique au constructeur NXP.

Android 4.2 ajoute à cette dernière le support de la norme NFC-NCI au travers du projet `external/libnfc-nci` de Broadcom. La norme NCI (*NFC Controller Interface*) permet d'interfacer plus facilement et de manière standardisée les contrôleurs NFC et le SoC. Broadcom propose donc une nouvelle stack « générique », permettant aux constructeurs d'ajouter plus simplement le support de leurs chips sous Android. Le chip BCM2079x de Broadcom reste néanmoins l'implémentation par défaut (aussi bien dans les sources, que matériellement).

En pratique, cela se traduit par une double HAL NFC au sein du fichier `hardware/libhardware/include/hardware/nfc.h`. Chaque constructeur pourra donc décider d'implémenter l'une ou l'autre, même s'il est recommandé d'utiliser NFC-NCI pour tout nouveau développement.

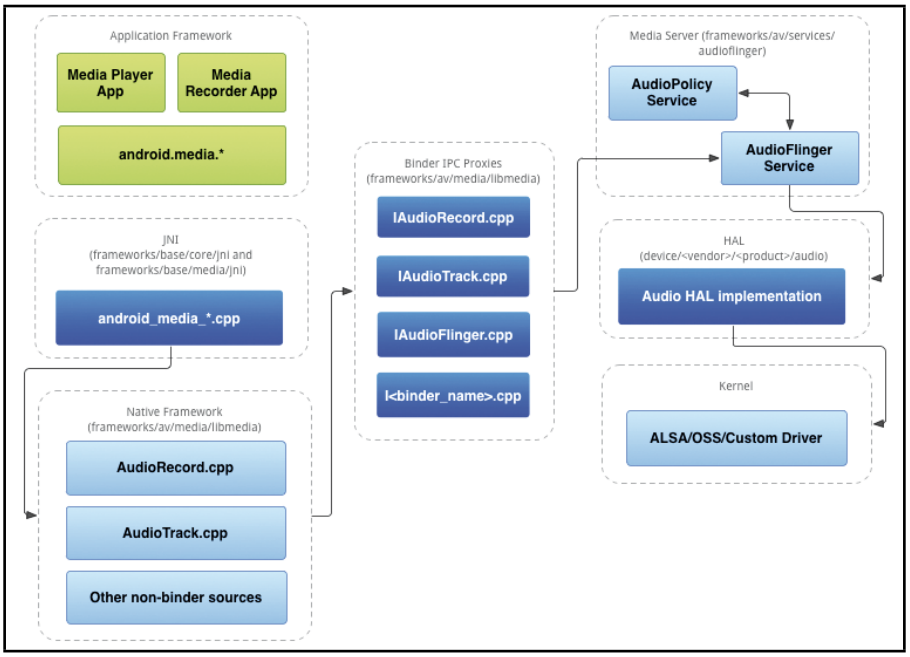


Fig. 4 : Architecture audio d'Android

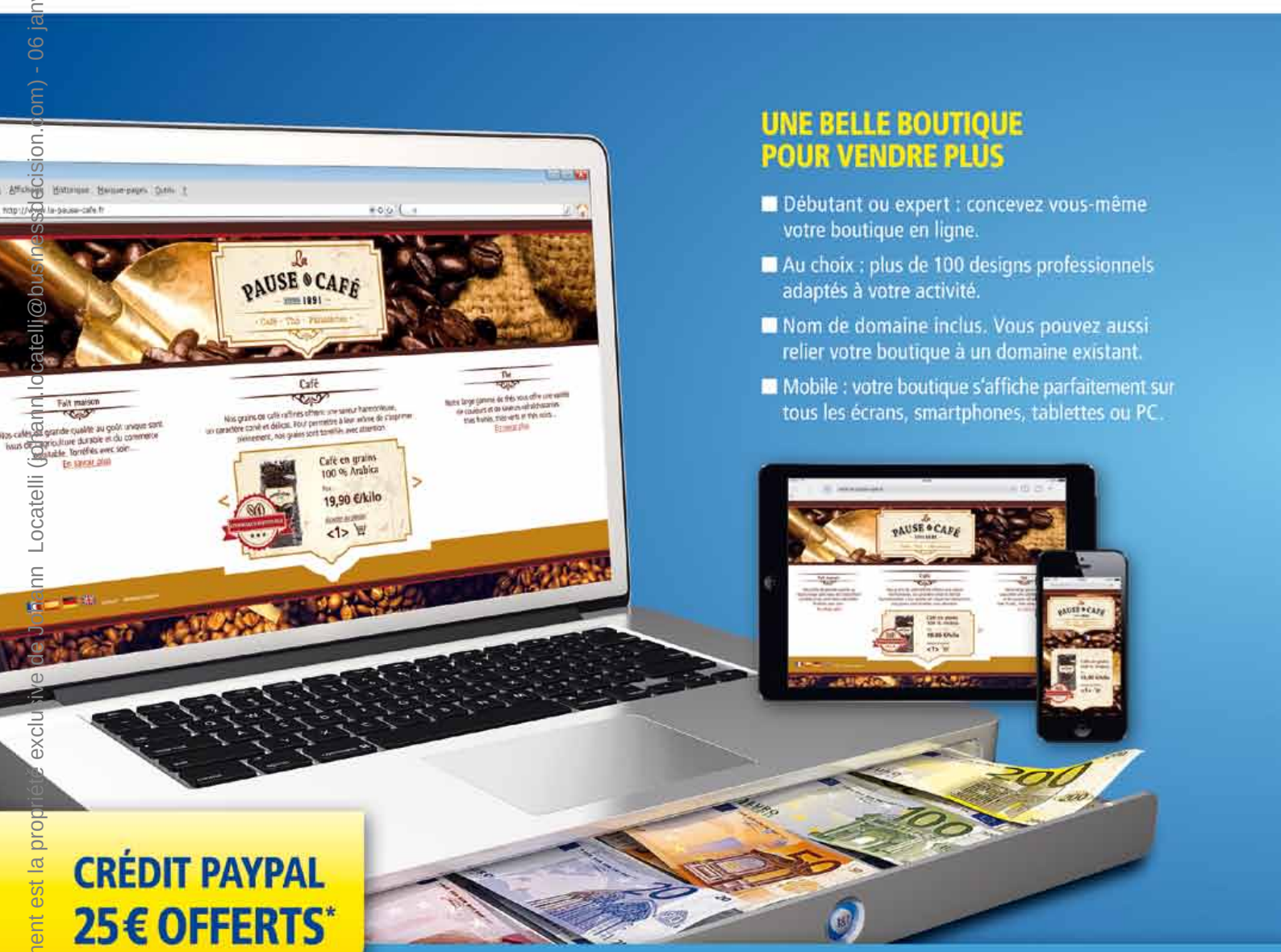
7 Audio

La couche audio se voit elle aussi quelque peu bousculée, principalement de par le support (enfin) de la restitution audio multicanal (e.g. 5.1 en lieu et place de la stéréo), des périphériques de sortie USB, du pré-processing d'effets audio, ou encore du support de périphériques audio distants.

Vous l'aurez naturellement compris, toutes ces belles fonctionnalités ont un impact direct sur les couches de bas niveau. Revenons donc quelques instants sur l'architecture audio d'Android pour restituer le contexte, tel que présenté par la figure 4.

Pour rappeler sommairement les choses, comme pour les autres services d'Android, les applications Java transitent par le framework applicatif Java qui communique avec la couche audio native `libmedia` (écrite en C++). Cette dernière interagit avec le Media Server et principalement les composants

1&1 BOUTIQUES



UNE BELLE BOUTIQUE POUR VENDRE PLUS

- Débutant ou expert : concevez vous-même votre boutique en ligne.
- Au choix : plus de 100 designs professionnels adaptés à votre activité.
- Nom de domaine inclus. Vous pouvez aussi relier votre boutique à un domaine existant.
- Mobile : votre boutique s'affiche parfaitement sur tous les écrans, smartphones, tablettes ou PC.

CRÉDIT PAYPAL 25 € OFFERTS*

MEMBRE DE united internet

DOMAINES | MAIL | HÉBERGEMENT | E-COMMERCE | SERVEURS

* Crédit de 25 € pour chaque compte marchand PayPal ouvert en tant que nouveau vendeur avant le 31/05/2014, depuis la page Web PayPal dédiée mise à disposition par 1&1, à condition d'activer le compte chez 1&1 et de réaliser au moins 10 transactions PayPal effectives dans les 3 mois suivant son ouverture.

EN LIGNE

NOUVEAU !
OFFRE DE LANCEMENT

1 AN À 0,99 € HT/mois**
~~29,99~~

Vous économisez 348 €

LANCEZ-VOUS DANS LA VENTE EN LIGNE !

FONCTIONS AVANCÉES. VENTES RENFORCÉES.

- Totale flexibilité : votre boutique évolue avec votre entreprise.
- Vendez de manière ciblée : produits personnalisables, offres promotionnelles et vente croisée.
- Évaluations et recommandations : donnez la parole à vos clients.
- Vendez dans le monde entier : grand choix de devises, langues et modes de paiement sécurisés comme PayPal.

GAGNEZ DES CLIENTS. GARDEZ-LES LONGTEMPS.

- Obtenez une bonne place sur les résultats des moteurs de recherche grâce au référencement (SEO).
- Vendez aussi sur Amazon, eBay et autres marketplaces.
- Créez facilement votre propre boutique Facebook.
- Fidélisez vos clients avec des newsletters et des bons de réduction.

TOTALEMENT SÛR. TELLEMENT PRO.

- Sécurité certifiée avec Trusted Shops. Exclusivité 1&1 : modèles de textes juridiques à personnaliser.
- Choix du prestataire de livraison : Chronopost, So Colissimo, etc.
- Disponibilité maximale : hébergement dans deux datacenters 1&1 distincts.
- Support 24h/24, 7j/7 : nos experts répondent à toutes vos questions.

1 MOIS POUR ESSAYER

1 CHOIX VOTRE DURÉE D'ENGAGEMENT

1 APPEL UN EXPERT VOUS RÉPOND

1&1

0970 808 911
(appel non surtaxé)

1and1.fr

** 30 jours « satisfait ou 100 % remboursé ». 1&1 Boutique en ligne Basic est à 0,99 € HT/mois (1,19 € TTC) pendant 12 mois. À l'issue des 12 premiers mois, son prix habituel de 29,99 € HT/mois (35,99 € TTC) s'applique. Offre à durée limitée, soumise à un engagement de 12 mois. Offre sans durée minimale d'engagement également disponible. Conditions détaillées sur 1and1.fr.

AudioFlinger et maintenant aussi AudioPolicy Manager, qui s'abstraient du matériel (ou du moins, du type de pilote utilisé par le noyau) via la désormais célèbre HAL. Libre à chaque constructeur d'implémenter son pilote en utilisant ALSA, OSS ou un pilote propriétaire. Seul importe qu'il soit compatible avec la HAL telle que définie dans le fichier **hardware/libhardware/include/hardware/audio.h**. À noter que si vous optez pour ALSA, Google recommande l'usage de **tinyalsa** (cf. **external/tinyalsa**) en lieu et place de la traditionnelle **libasound**, encore une fois pour une raison de licence (Apache vs. GPLv2).

Commençons par poser les bases de la nouvelle HAL audio (v2.0). Cette dernière supporte désormais 4 types de périphériques d'entrées/sorties (ou « modules ») :

- **primary** : il s'agit de la carte son principale, celle généralement disponible au travers de votre SoC.
- **a2dp** : une optionnelle interface Bluetooth (e.g. oreillette ou haut-parleurs).
- **usb** : une optionnelle interface USB externe (e.g. un DAC pour sortie numérique ou encore un dock audio). À noter que cette fonctionnalité est exportée au sein du *Android Open Accessory Development Kit* (ADK), afin de permettre aux développeurs de construire leur équipement compatible.
- **r_submix** : une optionnelle interface distante (e.g. un écran HDMI, comme dans le cas d'une liaison WifiDisplay ou Miracast, comme vu précédemment).

La HAL audio se compose de trois parties distinctes, qu'il vous sera nécessaire d'implémenter :

- **hardware/libhardware/include/hardware/audio.h**. Ceci regroupe les fonctions principales d'interfaçage avec le matériel.
- **hardware/libhardware/include/hardware/audio_policy.h**. Ceci représente l'interface avec l'Audio Policy Manager et débarque avec Jelly Bean. Ce dernier s'occupe du routage audio d'un flux d'une interface à l'autre, ainsi que de la politique de contrôle de volume de chaque interface. Chaque téléphone doit désormais disposer d'un fichier **/system/etc/audio_policy.conf** qui décrit ces politiques.
- **hardware/libhardware/include/hardware/audio_effect.h**. Jelly Bean apporte également la possibilité d'effectuer du pré ou post-traitement audio sur les flux d'entrées/sorties, tel que du ré-échantillonnage, de la conversion multicanal vers stéréo (*downmixing*), ou encore de la suppression de bruit ou d'écho. Ceci peut se faire aussi bien en logiciel qu'en matériel, si le constructeur décide d'implémenter le module requis et qu'il dispose de l'équipement. Dans tous les cas, cette politique d'application d'effets audio doit être également décrite au sein d'un fichier **/system/etc/audio_effects.conf**.

Aussi anecdotique que cela puisse être, notez enfin que Jelly Bean propose enfin le contrôle du volume principal (*master volume*) via les API suivantes de la HAL :

```
int (*get_master_volume)(struct audio_hw_device *dev, float *volume);
int (*set_master_mute)(struct audio_hw_device *dev, bool mute);
int (*get_master_mute)(struct audio_hw_device *dev, bool *mute);
```

7.1 Audio policy

Voyons donc plus en détails le contenu et la configuration de notre fichier **audio_policy.conf**. La balise principale permet de décrire votre matériel : les périphériques d'entrées/sorties fixes, ainsi que le périphérique de sortie audio par défaut. Dans l'exemple ci-dessous, vous trouverez un téléphone avec un haut-parleur, un écouteur et deux microphones :

```
global_configuration {
  attached_output_devices AUDIO_DEVICE_OUT_EARPIECE|AUDIO_DEVICE_OUT_SPEAKER
  default_output_device AUDIO_DEVICE_OUT_SPEAKER
  attached_input_devices AUDIO_DEVICE_IN_BUILTIN_MIC|AUDIO_DEVICE_IN_BACK_MIC
}
```

Enfin, pour chacun des modules (**primary**, **a2dp**, **usb**, **r_submix**) supportés, la configuration des entrées et sorties disponibles doit être décrite selon la syntaxe suivante :

```
audio_hw_modules {
  primary {
    outputs {
        ...
     primary {
        sampling_rates 44100|48000
        channel_masks AUDIO_CHANNEL_OUT_STEREO
        formats AUDIO_FORMAT_PCM_16_BIT
        devices AUDIO_DEVICE_OUT_EARPIECE|AUDIO_DEVICE_OUT_SPEAKER|AUDIO_DEVICE_
        OUT_WIRED_HEADSET|AUDIO_DEVICE_OUT_WIRED_HEADPHONE
        flags AUDIO_OUTPUT_FLAG_PRIMARY
      }
     ...
    }
      inputs {
      primary {
        sampling_rates 8000|11025|12000|16000|22050|24000|32000|44100|48000
        channel_masks AUDIO_CHANNEL_IN_MONO|AUDIO_CHANNEL_IN_STEREO
        formats AUDIO_FORMAT_PCM_16_BIT
        devices AUDIO_DEVICE_IN_BUILTIN_MIC|AUDIO_DEVICE_IN_WIRED_HEADSET|AUDIO_
        DEVICE_IN_BLUETOOTH_SCO_HEADSET|AUDIO_DEVICE_IN_VOICE_CALL|AUDIO_DEVICE_
        IN_BACK_MIC
      }
    }
}
```

Ceci permet ainsi à l'Audio Policy Manager de basculer facilement (et correctement) un flux vers ou depuis une interface.

Comme nous l'avons vu, Android apporte désormais le support de la restitution audio multicanal au travers du port HDMI. D'une manière générale, quel que soit le nombre de canaux audio en entrée, Android le réduit généralement au

nombre supporté par votre matériel (mono ou stéréo). Si votre matériel le supporte, le flux original peut alors être directement envoyé en brut via HDMI, passant outre AudioFlinger. Cette capacité doit alors être déclarée au sein du fichier **audio_policy.conf**, comme le montre l'exemple ci-dessous :

```
hdmi {
  sampling_rates 44100|48000
  channel_masks dynamic
  formats AUDIO_FORMAT_PCM_16_BIT
  devices AUDIO_DEVICE_OUT_AUX_DIGITAL
  flags AUDIO_OUTPUT_FLAG_DIRECT
}
```

Le champ **dynamic** indique à Android de questionner le périphérique sur ses capacités (e.g. 5.1) plutôt que de forcer le format manuellement.

Vous l'aurez compris, il est également très facile d'ajouter le support d'un dock audio USB stéréo, via l'exemple de configuration suivante :

```
usb {
  outputs {
    usb_accessory {
      sampling_rates 44100
      channel_masks AUDIO_CHANNEL_OUT_STEREO
      formats AUDIO_FORMAT_PCM_16_BIT
      devices AUDIO_DEVICE_OUT_USB_ACCESSORY
    }
    usb_device {
      sampling_rates 44100
      channel_masks AUDIO_CHANNEL_OUT_STEREO
      formats AUDIO_FORMAT_PCM_16_BIT
      devices AUDIO_DEVICE_OUT_USB_DEVICE
    }
  }
}
```

Et de même pour le mixeur HDMI distant d'une connexion Miracast, qu'il s'agisse d'une entrée ou d'une sortie :

```
r_submix {
    outputs {
      submix {
        sampling_rates 44100|48000
        channel_masks AUDIO_CHANNEL_OUT_STEREO
        formats AUDIO_FORMAT_PCM_16_BIT
        devices AUDIO_DEVICE_OUT_REMOTE_SUBMIX
      }
    }
    inputs {
      submix {
          sampling_rates 44100|48000
          channel_masks AUDIO_CHANNEL_IN_STEREO
          formats AUDIO_FORMAT_PCM_16_BIT
          devices AUDIO_DEVICE_IN_REMOTE_SUBMIX
      }
    }
```

7.2 Audio effects

Les développeurs soucieux d'améliorer la qualité d'un flux audio entrant (e.g. une conversation téléphonique) peuvent désormais appliquer des filtres de pré-traitement, tels que l'annulation de bruit (NS) ou d'écho (AEC), le contrôle automatique du gain (AGC)... Ces différents effets sont fournis par la librairie **audiofx**, mais leur application doit être spécifiée au sein du fichier **audio_effects.conf**, spécifique à chaque appareil. Voyons donc pour son contenu et sa configuration.

Le format est fort simple, comme illustré ci-dessous :

```
    pre_processing {
        <source d'entrée> {
            <effet> {
                <param 1> {
                    param {

int|short|float|bool|string <valeur>
                            [
int|short|float|bool|string <valeur> ]
                            ...
                        }
```

Les sources d'entrée possibles sont **mic**, **camcorder**, **voice_recognition** et **voice_communication**, tandis que les différents effets possibles sont **bassboost**, **virtualizer**, **equalizer**, **volume**, **reverb_env_aux**, **reverb_env_ins**, **reverb_pre_aux**, **reverb_pre_ins**, **visualizer**, **downmix** et **aec**.

L'exemple ci-dessous active donc l'annulation d'écho et la suppression de bruit pour le composant de VoIP, ainsi que le contrôle automatique du gain pour l'entrée ligne :

```
pre_processing {
   voice_communication {
       aec {}
       ns {}
   }
   camcorder {
       agc {}
   }
}
```

Enfin, et pour en conclure avec l'audio, notez que Jelly Bean améliore de manière significative la diffusion audio à faible latence (*low-latency*) au travers notamment des API OpenSL ES. Ceci requiert cependant un support matériel adéquat, qui doit être déclaré au travers des API. Je renverrai le lecteur intéressé au sein d'un guide explicatif de design matériel fourni par Google au sein du document **[I]**.

8 Multimédia

À l'exception d'une refonte complète de l'interface de gestion de la caméra (avec l'apparition de la HAL v3), Jelly Bean apporte peu de changements de fond quant à la couche multimédia, hormis de nouvelles fonctionnalités applicatives. D'une manière générale, je me rends compte avec cet article que nous n'avons jamais abordé les détails de la gestion de la caméra sous Android. Je vous propose donc de découvrir cette dernière au sein d'un prochain article dédié.

Mais revenons à Jelly Bean et aux changements multimédias de cette version :

- **Accès de bas niveau aux codecs**. Il est désormais possible, aussi bien au niveau applicatif qu'au travers du NDK, d'interroger le framework StageFright de l'existence de tel ou tel codec (e.g. H.264) au niveau de la plateforme (que son implémentation soit matérielle ou logicielle). Il devient également possible d'instancier un codec et d'y chaîner des buffers en entrée comme en sortie pour créer son propre *pipeline*, à la manière de GStreamer, offrant un contrôle complet des ressources offertes par chaque plateforme, y compris la possibilité de jouer du contenu protégé par DRM.
- **Routage multimédia**. En lien direct avec les nouvelles capacités d'affichage et de restitution audio vues précédemment, de nouvelles API **MediaRouter**, **MediaRouteActionProvider** et **MediaRouteButton** permettent désormais la diffusion sur des périphériques distants.

- **Mixage**. Les API de StageFright offrent désormais la capacité de créer ses propres fichiers multimédias (flux PS ou TS) à partir de flux élémentaires (ES) audio et vidéo.
- **Framework modulaire de gestion de droits numériques** (DRM). Ce dernier offre aux développeurs la capacité (malsaine) d'intégrer leur propre format de DRM au sein de protocoles de transport tels que MPEG DASH (*Dynamic Adaptive Streaming*), au-dessus d'HTTP.
- **Encodeur VP8**. Le décodeur étant disponible depuis bien longtemps (logiciellement du moins), place désormais à l'encodeur VP8. Les extensions OpenMAX 1.1.2 offertes au travers du NDK permettent de configurer finement les différents profils et niveaux de codec VP8 à utiliser.
- **Encodage vidéo depuis une surface**. Il est désormais possible (sans recopie de buffer) de passer directement une surface OpenGL ES à l'encodeur vidéo de StageFright. Ceci permet par exemple d'enregistrer facilement le contenu de votre écran.

9 Support multi-utilisateur

Si, comme tous les ans, on peut entendre l'éternelle promesse « Cette année est l'année de Linux sur desktop », cela n'a jamais été aussi vrai qu'avec Android et Jelly Bean. Ce dernier apporte en effet le support de multiples utilisateurs pour les utilisateurs de tablettes (ce n'est pas encore le cas pour les utilisateurs de smartphones).

Plusieurs utilisateurs peuvent ainsi se partager une même tablette, chacun n'ayant accès qu'à ses propres applications, données, bureaux... Le passage d'un utilisateur à l'autre se fait naturellement par une simple sélection au sein de l'écran de verrouillage. En réfléchissant un peu, vous comprendrez vite que ce cloisonnement des utilisateurs et de leurs données n'est pas anecdotique côté système et entraîne de nombreux impacts au niveau de la sécurité (nous y reviendrons dans quelques instants).

Le principal changement induit concerne la problématique de stockage externe. Que votre tablette dispose ou non d'un port pour carte SD, cette dernière y stocke les données utilisateurs. Si vous n'en disposez pas, une partition de votre flash eMMC sera utilisée à la place. Oui mais voilà, cette partition doit être lisible sous Windows/Linux/OS X, lorsque votre tablette est connectée via USB. Elle est donc formatée en FAT32. Maintenant essayez donc de gérer des droits utilisateurs sur un système FAT32 ;-)

Chaque utilisateur doit en effet disposer de son propre espace cloisonné sur ce système et ne doit pas pouvoir voir celui des autres utilisateurs. Le point de montage dynamique **/sdcard** de votre tablette est ainsi amené à être modifié en fonction de l'utilisateur actif.

Jelly Bean a donc recours à un nouveau daemon système naturellement dénommé **sdcard**, dont vous trouverez les sources dans le dossier **externals/core/sdcard**. Ce dernier utilise le système de fichiers FUSE pour émuler un système FAT et les permissions de fichiers/répertoires nécessaires. Ce daemon est lancé par root, mais rapidement dégradé vers un UID/GID privilégié (généralement **media_rw** ou **1023**) dès qu'un système de fichiers est monté.

Ce service est instancié directement au sein du fichier de configuration **init.rc** :

```
# virtual sdcard daemon running as media_rw (1023)
service sdcard /system/bin/sdcard /data/media /mnt/shell/emulated 1023 1023
    class late_start
```

Lorsqu'un utilisateur se connecte, le daemon **sdcard** monte le sous-répertoire spécifique à l'utilisateur au sein du point de montage défini par la variable **EMULATED_STORAGE_TARGET**. L'ensemble se configure facilement au sein du fichier **init.rc** par les paramètres suivants :

```
on init
    # See storage config details at http://source.android.com/tech/storage/
    mkdir /mnt/shell/emulated 0700 shell shell
    mkdir /storage/emulated 0555 root root

    export EXTERNAL_STORAGE /storage/emulated/legacy
    export EMULATED_STORAGE_SOURCE /mnt/shell/emulated
    export EMULATED_STORAGE_TARGET /storage/emulated

    # Support legacy paths
    symlink /storage/emulated/legacy /sdcard
    symlink /storage/emulated/legacy /mnt/sdcard
    symlink /storage/emulated/legacy /storage/sdcard0
    symlink /mnt/shell/emulated/0 /storage/emulated/legacy

on post-fs-data
    mkdir /data/media 0770 media_rw media_rw
```

Notez qu'il est désormais également possible de chiffrer les données utilisateurs simplement en paramétrant une variable d'environnement qui sera utilisée par le daemon de montage **vold** :

```
on fs
    setprop ro.crypto.fuse_sdcard true
```

Enfin, vous noterez que si un cloisonnement s'opère sur votre carte SD entre les données de chaque utilisateur, le système reste suffisamment intelligent pour mutualiser l'éventuel dossier **Android/obb** (*Opaque Binary Blob*), contenant de grosses archives binaires avec les données des applications téléchargées (ces fichiers peuvent être d'une taille conséquente, plusieurs centaines de Mo, voire Go).

10 Sécurité avancée

Finissons ce tour d'horizon des changements apportés par Jelly Bean par une revue des nouvelles fonctionnalités de sécurité introduites :

- **Vérification des applications**. L'utilisateur a désormais la possibilité d'activer la vérification des applications avant l'installation. Cette fonctionnalité, très utile, surtout si vous installez des applications par un biais autre que le PlayStore, permet de notifier l'utilisateur que l'application à installer pourrait être nuisible au système, qui a alors la possibilité de refuser l'installation.
- **Contrôle des SMS**. Android notifie désormais l'utilisateur si une application essaie d'envoyer des SMS vers un numéro surtaxé (à l'insu de l'utilisateur).
- **Garantie VPN**. Les applications peuvent désormais être configurées pour nécessiter l'utilisation d'une connexion VPN, empêchant tout accès au réseau si tel n'était pas le cas, évitant toute possible fuite de données sur un réseau non sécurisé ou autorisé.
- **Rejet de certificats SSL invalides**. Le framework permet enfin de rejeter des certificats SSL dont l'authenticité ne peut être validée, protégeant ainsi de possibles compromissions d'autorités de certifications (CA).
- **Affichage explicite des permissions applicatives requises**. Chaque application Android requiert un certain sous-ensemble de permissions (e.g. accès aux contacts, à votre carte SD...). Les permissions sont désormais triées, groupées et peuvent être détaillées à l'installation, donnant à l'utilisateur une meilleure compréhension des conséquences de son acceptation.
- **Renforcement du daemon installd**. Ce dernier, responsable de l'installation des packages applicatifs, tourne désormais sous un compte utilisateur Unix dégradé (précédemment root), empêchant ainsi toute attaque par escalade de privilèges.
- **Renforcement du script d'initialisation**. Ce dernier utilise désormais la politique `O_NOFOLLOW`, coupant ainsi court aux possibles attaques par déréférencement de liens symboliques.
- **Sécurisation des ContentProviders**. Ces derniers sont désormais configurés pour bloquer par défaut leur capacité d'export de données, empêchant l'accès aux applications non autorisées.
- **Isolation renforcée via SELinux**. L'isolation des applications est désormais renforcée par l'emploi possible de vérifications de contrôles d'accès (ou *Mandatory Access Control*, MAC) offertes par SELinux au niveau du noyau. Transparent au niveau applicatif tout comme pour l'utilisateur, ceci renforce d'autant la politique de sécurité d'Android. Si la politique par défaut reste permissive (et ne fait que tracer les violations de sécurité), il est possible de configurer le système pour devenir absolument intransigeant. L'implémentation de cette fonctionnalité et son impact système est un sujet en soi et si cette dernière vous passionne, je ne saurai que trop vous recommander la documentation constructeur fournie par Google **[2]**.
- **Suppression des droits setuid/setgid**. Ces droits ont été purement et simplement retirés de l'ensemble des binaires du système, empêchant toute attaque par escalade de privilèges. La partition `/system` est également montée avec le paramètre `nosuid`, empêchant les applications Android d'exécuter des programmes `setuid`.
- **Authentification ADB**. Souvenez-vous d'ADB (*Android Debug Bridge*), cet utilitaire magique de connexion distante (USB ou IP) à votre système. Depuis la version 4.2.2, une authentification client/serveur par échange de clés RSA est désormais nécessaire, empêchant tout un chacun avec un câble USB de se connecter à votre téléphone.
- **Restriction des capacités**. Aussi bien Zygote (le super-daemon, père de tous les processus Java) qu'ADB forcent désormais l'usage de `prctl(PR_CAPBSET_DROP)` avant de lancer une application, empêchant toute escalade de privilèges.
- **KeyStore provider dédié**. Le framework autorise désormais chaque application à créer son propre KeyStore par le biais du fournisseur (*provider*) associé et d'y déclarer des clés privées exclusives, empêchant d'autres applications d'y accéder.

Conclusion

Voici donc qui conclut notre première partie du dossier « Android Inside », présentant les changements significatifs apportés par Android Jelly Bean (versions 4.1 à 4.3). Vous l'aurez compris, l'OS grandit et gagne en maturité évidente au fur et à mesure des nouvelles versions. C'est ainsi avec plaisir que nous continuerons sur cette lancée, très prochainement, avec la découverte des changements liés à Android 4.4, lancé le 31 octobre 2013 sous le nom de Kit Kat, au sein d'une très forte campagne marketing. ■

Références

[1] http://source.android.com/devices/latency_design.html

[2] http://source.android.com/devices/tech/security/se-linux.html

PLANIFICATEUR DE REQUÊTES DE POSTGRESQL – LES PARCOURS

par Guillaume Lelarge

Un moteur de bases de données est composé de plusieurs parties, ne serait-ce qu'au niveau des requêtes : un analyseur syntaxique, un planificateur, un exécuteur. Le planificateur est certainement l'un des composants les plus importants : de ses capacités vont dépendre les performances du moteur. Un mauvais plan peut rapidement ralentir tout un système, alors qu'un bon plan permettra à la fois de bonnes performances et une grande interaction des différentes sessions. Cet article va tenter d'expliquer le planificateur de PostgreSQL, ses capacités, ses points forts, comme ses points faibles, pour permettre à tout un chacun de comprendre pourquoi telle requête utilise tel plan d'exécution.

1 Schéma et données de tests

Commençons par créer une base, y créer une table et lui ajouter quelques données :

```
$ psql postgres
psql (9.3.3)
Type "help" for help.

postgres=# CREATE DATABASE planif;
CREATE DATABASE
postgres=# \c planif
You are now connected to database "planif" as user
"guillaume".
planif=# CREATE TABLE t1 (c1 integer, c2 text);
CREATE TABLE
planif=# ALTER TABLE t1 SET (autovacuum_enabled=off);
ALTER TABLE
planif=# INSERT INTO t1
planif-# SELECT i, md5(random()::text) FROM generate_
series(1, 1000000) AS i;
INSERT 0 1000000
```

La commande **ALTER TABLE** sert à désactiver l'autovacuum sur la table, le but étant de contrôler quand nous allons faire un **ANALYZE** ou un **VACUUM** sur cette table. Le reste des commandes est suffisamment simple pour ne pas nécessiter d'explications.

2 Parcours de table

On va continuer avec la requête la plus simple possible, une extraction de toutes les données d'une table :

```
SELECT * FROM t1;
```

Pour récupérer le plan d'exécution, il nous faut utiliser l'instruction **EXPLAIN**. Par défaut, elle ne renvoie que le plan et des informations sur les estimations du planificateur. Il est possible d'y ajouter quelques options comme :

- **COSTS**, pour afficher les coûts estimés par le planificateur (ils sont affichés par défaut) ;
- **ANALYZE**, pour récupérer les informations réelles provenant de l'exécution de la requête (attention, dans ce cas, la requête est réellement exécutée) ;
- **TIMING**, pour récupérer le chronométrage de chaque opération du plan d'exécution (ils sont affichés par défaut, une fois que l'option **ANALYZE** est activée) ;
- **BUFFERS**, pour récupérer l'utilisation des blocs disque au niveau du cache de PostgreSQL (ces informations ne sont disponibles que si la requête est exécutée, donc que si l'option **ANALYZE** est activée) ;
- **VERBOSE**, pour renvoyer plein d'autres informations.

Maintenant, testons la requête :

```
planif=# EXPLAIN SELECT * FROM t1;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on t1  (cost=0.00..13991.02 rows=771702 width=36)
(1 row)
```

PostgreSQL nous indique que l'exécuteur va faire un parcours séquentiel de la table (c'est le nœud **Seq Scan**).

Note

Attention, l'option **ANALYZE** cause vraiment l'exécution de la requête. Donc, si vous exécutez un **EXPLAIN ANALYZE** sur une requête de modification de données (**INSERT**, **UPDATE**, **DELETE**, **COPY**, **TRUNCATE**), ces modifications auront bien lieu. Pour éviter cela, vous devez embarquer votre **EXPLAIN ANALYZE** dans une transaction que vous annulerez par la suite, comme ceci :

```
BEGIN;
EXPLAIN ANALYZE DELETE FROM t1 WHERE c1<500;
ROLLBACK;
```

Un parcours séquentiel revient à lire les fichiers de la table, bloc par bloc, sans chercher à filtrer (pas de clause **WHERE**), ou à trier le résultat (pas de clause **ORDER BY**).

Il nous indique aussi quelques informations statistiques entre parenthèses :

- **cost=0.00..13991.02** le premier nombre correspond au coût de récupération de la première ligne, alors que le deuxième indique le coût de récupération de toutes les lignes ;
- **rows=771702** nous montre l'estimation du nombre de lignes renvoyées par l'exécution du parcours séquentiel ;
- **width=36** précise la moyenne du nombre d'octets renvoyés par une seule ligne.

Ces données sont calculées à partir des statistiques récupérées par PostgreSQL sur les données des tables (par exemple la largeur moyenne d'une colonne), ou sur les propriétés d'une table (par exemple le nombre de lignes ou le nombre de blocs). Ces statistiques sont principalement calculées lors de l'exécution de la commande **ANALYZE**. Comme nous avons désactivé l'autovacuum et que nous n'avons pas exécuté de **VACUUM** ou d'**ANALYZE** manuel, le planificateur ne dispose d'aucune information statistique qu'il pourrait utiliser. Du coup, il utilise des valeurs assez basses, mais pas à zéro, car il y a peu de chances qu'une table n'ait réellement aucune ligne. De plus, il sait grâce au collecteur de statistiques qu'un million de lignes a été inséré dans la table. Il est aussi possible que d'autres transactions aient commencé d'autres modifications sur les données de la table.

Exécutons **ANALYZE** pour voir le changement au niveau des statistiques et du plan d'exécution :

```
planif=# ANALYZE t1;
ANALYZE
planif=# EXPLAIN SELECT * FROM t1;
                        QUERY PLAN
-----------------------------------------------
 Seq Scan on t1  (cost=0.00..16274.00 rows=1000000
width=16)
(1 row)
```

Le plan en lui-même n'a pas changé, mais les informations statistiques sont bien différentes. Cette fois-ci, le nombre de lignes est beaucoup plus précis. Il est même exact dans notre cas. Détaillons maintenant chaque information.

Si on regarde les coûts, récupérer la première ligne est globalement gratuit, tout simplement parce que le premier bloc a de fortes chances d'être déjà en mémoire (dans le cache disque de PostgreSQL, ou dans le cache disque du système d'exploitation) et que le décodage de la première ligne sera très rapide. Par contre, récupérer toutes les lignes dépend directement de la taille de la table, plus exactement du nombre de blocs à lire, ainsi que du nombre de lignes à déchiffrer. Le coût total revient donc à multiplier le coût unitaire d'un bloc avec le nombre de blocs de la table et y additionner le coût de décodage d'une ligne multiplié avec le nombre de lignes.

Le fichier **postgresql.conf** comprend un paramètre indiquant le coût de lecture d'un bloc séquentiel. Il s'appelle **seq_page_cost** et vaut **1** par défaut. L'autre paramètre s'appelle **cpu_tuple_cost** et vaut **0,01** par défaut. Autrement dit, cela sous-entend que le traitement d'une ligne est 100 fois plus rapide que la lecture d'un bloc séquentiel, ce qui paraît assez logique (un traitement par le CPU est en règle générale plus rapide qu'une lecture sur disque).

Le nombre de blocs et de lignes sont des statistiques disponibles dans les colonnes du catalogue système **pg_class**, respectivement **relpages** et **reltuples**.

Pour calculer le coût total, on peut donc utiliser cette requête :

```
planif=# SELECT current_setting('cpu_tuple_
cost')::float4*reltuples
planif-#      + current_setting('seq_page_
cost')::float4*relpages
planif-# AS cout_total
planif-# FROM pg_class
planif-# WHERE relname='t1';
 cout_total
------------
      16274
(1 row)
```

On retrouve bien le coût de 16274 calculé par le planificateur.

Concernant le nombre de lignes, on a déjà vu qu'il s'agissait de la colonne **reltuples** du catalogue système **pg_class** :

```
planif=# SELECT reltuples FROM pg_class WHERE relname='t1';
 reltuples
-----------
     1e+06
(1 row)
```

La largeur moyenne est récupérée grâce aux statistiques sur les données des colonnes. Ces statistiques sont stockées dans le catalogue système **pg_statistic**, mais nous allons utiliser la vue système **pg_stats**, bien plus simple à comprendre et à utiliser. Dans cette vue, il y a un enregistrement par colonne de chaque table analysée de la base. Cet enregistrement contient une colonne appelée **avg_width**, correspondant à la largeur moyenne des données de la colonne. Nous avons fait un **SELECT ***, autrement dit nous récupérons toutes les colonnes. Il faut donc récupérer la somme des valeurs de la colonne **avg_width** pour chaque colonne de la table **t1** :

```
planif=# SELECT sum(avg_width) FROM pg_stats
WHERE tablename='t1';
 sum
-----
  16
(1 row)
```

Là aussi, on retrouve bien l'information fournie par la commande **EXPLAIN**. Cette information peut sembler inutile. En fait, elle permet de connaître le volume de données que PostgreSQL va devoir traiter ou va devoir envoyer à l'application. Si on récupère par exemple un million de lignes, cela représentera dans notre cas 16 Mo. Si par contre, on n'est intéressé que par la colonne **c1**, de largeur 4 octets, on n'aurait à récupérer que 4 Mo. Cela fait une grosse différence au niveau des performances. Donc, il est vraiment nécessaire, pour des raisons de performance, de ne demander que les colonnes importantes.

3 Petit retour sur ANALYZE

Comme nous l'avons dit, la commande **ANALYZE** permet de calculer des statistiques sur les données des tables. Voyons ce qu'elle fait plus précisément.

Cette commande lit des lignes aléatoires dans la table. Elle récupère un ensemble de lignes dont le nombre dépend directement du paramètre **default_statistics_target**. En multipliant la valeur de ce paramètre par 300, nous obtenons la taille de l'échantillon de lignes récupéré par la commande **ANALYZE**. Par défaut, ce paramètre vaut 100 depuis la version 8.4 (il était à 10 auparavant), ce qui nous donne 30000 lignes pour l'échantillon.

Sur ces 30000 lignes, la commande **ANALYZE** va calculer un ensemble de statistiques : pourcentage de valeurs **NULL**, largeur moyenne d'une ligne, nombre de valeurs distinctes, histogramme des valeurs, valeurs les plus fréquentes, fréquences de ces valeurs, etc. Toutes ces statistiques sont stockées dans la table **pg_statistic**. Cette table est utilisée par le planificateur. Elle est optimisée pour lui, ce qui veut dire qu'elle est difficilement lisible par un humain. Ces derniers vont plutôt utiliser la vue **pg_stats** qui leur fournira à peu près les mêmes informations, mais d'une façon beaucoup plus lisible. Voici le contenu de cette vue pour la table **t1** (filtre **tablename='t1'**) et pour la colonne **c1** (filtre **attname='c1'**) :

```
planif=# \x
Expanded display is on.
planif=# SELECT * FROM pg_stats WHERE tablename='t1'
and attname='c1';
-[ RECORD 1 ]----------+--------------------------
--------------------------
schemaname             | public
tablename              | t1
attname                | c1
inherited              | f
null_frac              | 0
avg_width              | 4
n_distinct             | -1
most_common_vals       |
most_common_freqs      |
histogram_bounds       | {73,9600,20383,30580,40857,
51339,60926,70514,80550,89841,99380,108575,118008,12
8606,137609,147939,158548,168055,178221,187765,19748
3,207907,217755,228317,239142,249235,259247,269100,2
78745,288511,298666,309525,320046,330255,339519,3484
03,359142,369203,380109,391577,401896,411652,421849,
431225,441261,451518,462070,471488,481683,491112,501
204,511317,520419,530983,540787,551583,560671,570313
,579400,589584,599529,609140,619184,628314,638931,64
8341,658563,667822,678849,688979,699496,709984,72000
1,730149,740637,750343,760662,770458,780672,790034,8
01167,811755,821786,832554,842784,853873,864090,8727
31,882503,892648,901636,910741,921145,930439,939706,
950361,961169,971079,980753,990644,999987}
correlation            | 1
most_common_elems      |
most_common_elem_freqs |
elem_count_histogram   |
```

Et voici la raison d'être de chaque colonne : voir tableau ci-dessous.

Du résultat de la requête ci-dessus, on peut déduire que la colonne **c1** a des données de quatre octets de largeur en moyenne (normal pour une colonne de type **integer**). Il n'y a pas de valeurs **NULL**, toutes les valeurs sont distinctes, la colonne n'est pas héritée, etc.

Sur certaines tables ou colonnes, on aimerait récupérer un échantillon plus important. Il est possible de le faire grâce à un **ALTER TABLE** :

```
ALTER TABLE t1 ALTER c1 SET STATISTICS 1000 ;
```

Il est parfois préférable de le faire directement au niveau de la colonne. En effet, le faire de façon globale va avoir un impact sur la commande **ANALYZE** et sur le planificateur. La commande **ANALYZE** devant lire plus de lignes, elle sera automatiquement plus lente. Elle aura aussi plus de statistiques à stocker, ce qui la ralentira mais fera aussi grossir la table **pg_statistic**. Enfin, le planificateur, ayant beaucoup plus d'informations, aura tendance à être plus lent sur son travail. Par contre, ce dernier sera certainement plus optimisé et le plan résultant plus adéquat.

La bonne configuration du paramètre **default_statistics_target** dépend de la balance que vous voulez obtenir entre la durée d'exécution du **ANALYZE**, la rapidité du planificateur de requêtes et la justesse du plan d'exécution.

Colonne	Raison d'être
schemaname	Nom du schéma contenant la table
tablename	Nom de la table contenant la colonne
attname	Nom de la colonne pour laquelle on a des statistiques
inherited	Booléen indiquant si la colonne est héritée d'une table mère
null_frac	Ratio de valeurs **NULL** par rapport aux valeurs non **NULL**
avg_width	Largeur moyenne de la colonne en octets. Fixe dans le cas où le type de données a une largeur fixe (**integer** par exemple), variable (et donc potentiellement plus intéressant) pour les types de données de taille variable (**char**, **varchar**, **text**, **bytea**)
n_distinct	Nombre de valeurs distinctes. -1 si elles sont toutes distinctes.
most_common_vals	Tableau des valeurs les plus fréquentes
most_common_freqs	Tableau de fréquence des valeurs les plus fréquentes
histogram_bounds	Histogramme des valeurs, découpé en X morceaux, X dépendant de la valeur du paramètre **default_statistics_target**
correlation	Corrélation des données
most_common_elems	Si la colonne contient un tableau, tableau des éléments les plus fréquents
most_common_elems_freqs	Si la colonne contient un tableau, tableau de fréquence des éléments les plus fréquents
elem_count_histogram	Si la colonne contient un tableau, histogramme des éléments

L'un des inconvénients de l'analyseur de PostgreSQL est qu'il n'est pas capable de calculer des statistiques sur la corrélation entre deux colonnes, ce qui rend les statistiques du planificateur peu fiables dans le cadre d'une recherche de type **c1=une_valeur AND c2=une_autre_valeur** et pour laquelle les valeurs de **c1** et **c2** ont une forte corrélation.

4 Parcours de table avec filtre

Maintenant, essayons une autre requête, cette fois avec un filtre :

```
planif=# EXPLAIN SELECT * FROM t1 WHERE c1>500;
                        QUERY PLAN
-----------------------------------------------------
 Seq Scan on t1  (cost=0.00..18774.00 rows=999552 width=16)
   Filter: (c1 > 500)
(2 rows)
```

Ce plan montre que l'exécuteur va faire un parcours séquentiel de la table et qu'il va filtrer les lignes suivant une condition : **c1>500**. Plus la table est grosse, plus le travail sera important, étant donné que l'exécuteur va devoir récupérer chaque ligne et les tester sur la condition indiquée pour savoir s'il doit les conserver dans l'ensemble de lignes en résultat.

On peut remarquer que, même si le nombre de lignes renvoyées est plus petit que celui de la requête précédente, le coût total est lui plus important. En fait, en plus de lire chaque bloc de la table et de décoder chaque ligne pour savoir si elle est visible ou non, il faut aussi tester la condition pour chacune des lignes. L'utilisation d'un opérateur a un coût spécifié par le paramètre **cpu_operator_cost**, qu'il faut multiplier par le nombre de lignes à tester. Cela nous donne :

```
planif=# SELECT current_setting('cpu_tuple_cost')::float4    * reltuples
planif-#      + current_setting('seq_page_cost')::float4    * relpages
planif-#      + current_setting('cpu_operator_cost')::float4 * reltuples
planif-# AS cout_total
planif-# FROM pg_class
planif-# WHERE relname='t1';
 cout_total
------------
      18774
(1 row)
```

Il est à noter que le nombre de lignes en retour n'a aucune importance dans le calcul du coût. Pour PostgreSQL, la durée d'exécution de la requête sera identique qu'on termine avec une ligne ou avec un million de lignes. Ça ne sera pas la même chose au niveau de l'application qui va envoyer la requête, car elle devra récupérer les lignes du résultat. Récupérer une ligne est clairement beaucoup plus rapide qu'en récupérer un million.

Le nombre de lignes estimé est tombé à 999552. Ce n'est pas le nombre exact, mais cela reste très proche de la réalité. Le planificateur dispose d'un histogramme des valeurs qu'il va mettre à contribution dans ce cas. Cet histogramme a été récupéré par la commande **ANALYZE**, exécutée précédemment. Voici ce que contient l'histogramme pour la colonne **c1** de cette table :

```
planif=# SELECT histogram_bounds
planif-# FROM pg_stats
planif-# WHERE tablename='t1' AND attname='c1';
                                histogram_bounds
------------------------------------------------------------------------------
{73,9600,20383,30580,40857,51339,60926,70514,80550,89841,99380,108575,118008,128606,
137609,147939,158548,168055,178221,187765,197483,207907,217755,228317,239142,249235,
259247,269100,278745,288511,298666,309525,320046,330255,339519,348403,359142,369203,
380109,391577,401896,411652,421849,431225,441261,451518,462070,471488,481683,491112,
501204,511317,520419,530983,540787,551583,560671,570313,579400,589584,599529,609140,
619184,628314,638931,648341,658563,667822,678849,688979,699496,709984,720001,730149,
740637,750343,760662,770458,780672,790034,801167,811755,821786,832554,842784,853873,
864090,872731,882503,892648,901636,910741,921145,930439,939706,950361,961169,971079,
980753,990644,999987}
(1 row)
```

L'histogramme contient 100 valeurs (ce qui correspond à la configuration du paramètre **default_statistics_target**). Il y a autant de valeurs entre deux points côte à côte de l'histogramme. Par exemple, il y a autant de valeurs entre les valeurs 30580 et 40857 qu'entre les valeurs 431225 et 441261. L'estimation du nombre de lignes étant d'un million, on peut conclure qu'il y a 10000 valeurs (un million divisé par 100) entre deux points côte à côte. Autrement dit, comme nous cherchons toutes les lignes pour lesquelles **c1** est supérieur à 500 et que 500 se trouve dans le premier bloc de valeurs (i. e., entre 73 et 9600), l'estimation du nombre de lignes peut se basculer en faisant le ratio pour ce premier bloc auquel on va additionner le nombre de valeurs pour chaque autre bloc. Cela revient à cette requête :

```
planif=# SELECT round(
planif-# (
planif-#  (9600::float4-500)/(9600-73)
planif-#  +
planif-#  (current_setting('default_statistics_target')::int4 - 1)
planif-# )
planif-# * 10000.0
planif-# ) AS rows;
  rows
--------
 999552
(1 row)
```

On retrouve bien la valeur du nombre estimé de lignes. Attention à bien faire des divisions à virgule flottante (d'où la conversion de type en **float4**), sinon l'estimation ne sera pas juste.

Essayons une autre requête, cette fois avec l'opérateur d'égalité :

```
planif=# EXPLAIN SELECT * FROM t1 WHERE c1=500;
                        QUERY PLAN
-----------------------------------------------------
 Seq Scan on t1  (cost=0.00..18774.00 rows=1 width=16)
   Filter: (c1 = 500)
(2 rows)
```

L'estimation ici est exacte. En effet, nous n'avons inséré que des valeurs différentes dans la colonne **c1**. Du coup, si

on cherche les lignes dont la colonne **c1** est égale à une certaine valeur, nous ne pouvons récupérer qu'une seule ligne. Il n'y a pas de contrainte unique sur cette colonne, alors comment le planificateur peut estimer le nombre de lignes si précisément ?

En fait, le planificateur sait que les valeurs de cette colonne sont toutes distinctes grâce à une autre statistique. La commande **ANALYZE** calcule le nombre de valeurs distinctes pour chaque colonne. Elle stocke cette information dans la colonne **n_distinct** de la vue **pg_stats** :

```
planif=# SELECT n_distinct FROM pg_stats WHERE tablename='t1' AND
attname='c1';
 n_distinct
------------
         -1
(1 row)
```

La valeur **-1** est une valeur spéciale indiquant que chaque valeur est différente pour la colonne **c1** de la table **t1**. De ce fait, le planificateur en déduit que la requête ne peut renvoyer qu'une seule ligne.

Il est à noter que, même si l'estimation du nombre de lignes a dramatiquement diminué, le coût total n'a pas bougé d'un pouce. En effet, l'exécuteur doit toujours lire toutes les lignes de la table et toutes les traiter.

Il est quand même dommage de devoir lire toutes les lignes pour trouver la seule qui nous intéresse. La commande **EXPLAIN** peut d'ailleurs nous indiquer le nombre de lignes ignorées lorsqu'on utilise l'option **ANALYZE** :

```
planif=# EXPLAIN (ANALYZE) SELECT * FROM t1 WHERE c1=500;
                        QUERY PLAN
-----------------------------------------------------
 Seq Scan on t1  (cost=0.00..18774.00 rows=1 width=16)
                 (actual time=23.663..108.773 rows=1 loops=1)
   Filter: (c1 = 500)
   Rows Removed by Filter: 999999
 Total runtime: 108.833 ms
(4 rows)
```

La ligne intéressante est la ligne « Rows Removed by Filter: 999999 ». Avec un tel ratio de lignes ignorées, il serait intéressant de pouvoir accéder directement à la ligne qui nous intéresse.

5 Parcours d'index

En fait, il faudrait pouvoir avoir un index des valeurs et de leur position dans la table. C'est exactement le but d'un index. Créons-en un :

```
planif=# CREATE INDEX ON t1(c1);
CREATE INDEX
```

Maintenant que l'index est créé, PostgreSQL va pouvoir l'utiliser pour accélérer la récupération des données qui nous intéressent :

```
planif=# EXPLAIN SELECT * FROM t1 WHERE c1=500;
                           QUERY PLAN
-----------------------------------------------------------------
 Index Scan using t1_c1_idx on t1  (cost=0.42..8.44 rows=1 width=16)
   Index Cond: (c1 = 500)
(2 rows)
```

Ici, l'exécuteur parcourt l'index (nœud **Index Scan**) et utilise la condition (ligne **Index Cond**) pour ne parcourir que la partie de l'index contenant les données recherchées. C'est pour cela qu'il récupère très rapidement la ligne qui l'intéresse. De ce fait, le coût est très bas par rapport à l'ancienne exécution : seulement 8,44 au lieu de 18774.

Que se passe-t-il dans le cas d'un parcours d'index ? L'exécuteur va lire le premier bloc de l'index. Dans le cas de notre index, c'est un arbre balancé, ce qui lui évitera d'avoir à lire tous les blocs. Ce premier bloc va donc lui indiquer le prochain bloc à lire, et ainsi de suite jusqu'à trouver la bonne valeur. Dès qu'il retrouve une valeur intéressante, il va lire le bloc de la table contenant cette ligne et vérifiera si cette ligne est visible par notre transaction. Il ne renverra la ligne que si cette condition est vérifiée. Une fois qu'il a examiné tous les blocs intéressants de l'index, il aura terminé son travail.

Pour vérifier qu'il y a bien lecture de l'index et de la table, il suffit de regarder les statistiques d'entrées/sorties pour les tables :

```
planif=# SELECT heap_blks_read, heap_blks_hit, idx_blks_read, idx_blks_hit
planif-# FROM pg_statio_user_tables
planif-# WHERE relname='t1';
 heap_blks_read | heap_blks_hit | idx_blks_read | idx_blks_hit
----------------+---------------+---------------+--------------
           6277 |       1025090 |             1 |            0
(1 row)
planif=# EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM t1 WHERE c1=500;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using t1_c1_idx on t1  (cost=0.42..8.44 rows=1 width=16)
                                   (actual time=0.108..0.109 rows=1 loops=1)
   Index Cond: (c1 = 500)
   Buffers: shared hit=1 read=3
 Total runtime: 0.148 ms
(4 rows)

planif=# SELECT heap_blks_read, heap_blks_hit, idx_blks_read, idx_blks_hit
planif-# FROM pg_statio_user_tables
planif-# WHERE relname='t1';
 heap_blks_read | heap_blks_hit | idx_blks_read | idx_blks_hit
----------------+---------------+---------------+--------------
           6277 |       1025091 |             4 |            0
(1 row)
```

Autrement dit, il y a eu lecture de trois blocs au niveau de l'index (**idx_blks_read** passant de 1 à 4) et d'un bloc de la table (**heap_blks_hit** passant de 1025090 à 1025091). Quatre blocs sont lus en tout (un dans le cache de PostgreSQL et 3 en dehors), ce qu'accrédite d'ailleurs la commande **EXPLAIN** avec l'option **BUFFERS** (ligne « Buffers : shared hit=1 read=3 »). Si on ré-exécute la commande, on s'aperçoit que l'exécuteur de PostgreSQL lit le même nombre de blocs, à la différence près qu'ils sont maintenant tous en cache :

ABONNEZ-VOUS !

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

NOUVEAU ! Abonnez-vous (réabonnez-vous) en ligne sur :

boutique.ed-diamond.com

Vous pouvez ainsi :
- ➧ Avoir accès à votre suivi personnalisé d'abonnement
- ➧ Profiter des promos réservées à nos abonnés
- ➧ Vous réabonner facilement sans interruption d'abonnement

Pour plus d'informations, veuillez nous contacter via e-mail : cial@ed-diamond.com ou par téléphone : +33 (0)3 67 10 00 20

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 newsletters des Éditions Diamond.

❑ Je souhaite recevoir les offres promotionnelles des partenaires des Éditions Diamond.

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

ABONNEMENT GLMF

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

offre LM

60€* au lieu de **86,90€**** en kiosque

Économie 26,90€

INCLUS : GNU/Linux Magazine (11nos)

offre LM+

+

115€* au lieu de **164,30€**** en kiosque

Économie 49,30€

INCLUS : GNU/Linux Magazine (11nos) + ses 6 Guides

NOUVEAUTÉ 2014
TOUS LES HORS-SÉRIES PASSENT EN **GUIDES !**

offre T

198€* au lieu de **277,10€**** en kiosque

Économie 79,10€

INCLUS : GNU/Linux Magazine (11nos), Open Silicium (4nos), MISC (6nos), Linux Pratique (6nos) et Linux Essentiel (6nos)

offre T+

299€* au lieu de **411,20€**** en kiosque

Économie 112,20€

INCLUS : GNU/Linux Magazine (11nos) + ses 6 Guides, Linux Pratique (6nos) + ses 3 Guides, MISC (6nos) + ses 2 Hors-Séries, Open Silicium (4nos) et Linux Essentiel (6nos)

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

NOUVEAU ! Abonnez-vous (réabonnez-vous) en ligne sur :

boutique.ed-diamond.com

Vous pouvez ainsi : ➡ Avoir accès à votre suivi personnalisé d'abonnement ➡ Profiter des promos réservées à nos abonnés ➡ Vous réabonner facilement sans interruption d'abonnement

Pour plus d'informations, veuillez nous contacter via e-mail : cial@ed-diamond.com ou par téléphone : +33 (0)3 67 10 00 20

➡ 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 françaises

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 de 415 euros.

Je choisis de régler par :

☐ Chèque bancaire ou postal à l'ordre des Éditions Diamond (uniquement France et DOM TOM)

☐ Pour les règlements par virements, veuillez nous contacter via e-mail : cial@ed-diamond.com ou par téléphone : +33 (0)3 67 10 00 20

Date et signature obligatoires

```
planif=# EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM t1 WHERE c1=500;
                                QUERY PLAN
-------------------------------------------------------------------
 Index Scan using t1_c1_idx on t1  (cost=0.42..8.44 rows=1 width=16)
                      (actual time=0.024..0.025 rows=1 loops=1)
   Index Cond: (c1 = 500)
   Buffers: shared hit=4
 Total runtime: 0.054 ms
(4 rows)

planif=# SELECT heap_blks_read, heap_blks_hit, idx_blks_read, idx_blks_hit
planif-# FROM pg_statio_user_tables
planif-# WHERE relname='t1';
 heap_blks_read | heap_blks_hit | idx_blks_read | idx_blks_hit
----------------+---------------+---------------+--------------
           6277 |       1025092 |             4 |            3
```

Donc, il y a eu lecture de trois blocs au niveau de l'index (**idx_blks_hit** passant de 0 à 3) et d'un bloc de la table (**heap_blks_hit** passant de 1025091 à 1025092). Ce qu'accrédite encore la commande **EXPLAIN** grâce à l'option **BUFFERS** (ligne « Buffers: shared hit=4 »). De plus, la lecture a bien été partielle, car la table comme l'index comprennent bien plus que quatre blocs :

```
planif=# SELECT pg_table_size('t1')/8192        AS "Nb blocs table",
planif-#        pg_table_size('t1_c1_idx')/8192 AS "Nb blocs index";
 Nb blocs table | Nb blocs index
----------------+----------------
           6279 |           2745
(1 row)
```

Au niveau des statistiques, la largeur moyenne d'une ligne n'a pas changé vu qu'on récupère toujours les deux colonnes. Le nombre de lignes est de 1, car PostgreSQL sait que chaque ligne contient une valeur différente pour la colonne **c1**.

Maintenant, essayons de lire un plus grand nombre de lignes :

```
planif=# EXPLAIN SELECT * FROM t1 WHERE c1>500;
                          QUERY PLAN
---------------------------------------------------------
 Seq Scan on t1  (cost=0.00..18774.00 rows=999480 width=16)
   Filter: (c1 > 500)
(2 rows)
```

Le planificateur est passé à un parcours séquentiel. La raison est assez simple : quand il s'agit de lire une proportion importante d'une table, une lecture séquentielle sera probablement plus rapide (notamment avec la technologie du *read-ahead* du noyau) qu'une lecture aléatoire qui demandera de fréquents déplacements de la tête de lecture. Désactivons les parcours séquentiels pour voir le plan que suggèrera le planificateur :

```
planif=# SET enable_seqscan TO off;
SET
planif=# EXPLAIN SELECT * FROM t1 WHERE c1>500;
                                QUERY PLAN
---------------------------------------------------------------------------
 Index Scan using t1_c1_idx on t1  (cost=0.42..34741.33 rows=999480 width=16)
   Index Cond: (c1 > 500)
(2 rows)
```

Note

Les paramètres **enable_*** permettent de fortement défavoriser un certain type de nœuds. Par exemple, **enable_seqscan** n'empêche pas réellement l'utilisation d'un parcours séquentiel (sans index, on ne peut pas faire autrement). Il va donc ajouter un surcoût très fort à son utilisation pour qu'il apparaisse très peu intéressant.

Deux observations : le planificateur passe bien à un parcours d'index et le coût final est un peu plus élevé. Il est donc logique que PostgreSQL utilise un parcours séquentiel dans ce cas. Si on essaie d'exécuter la requête dans les deux cas, on obtient les temps d'exécution suivants :

```
planif=# \o /dev/null
planif=# \timing
Timing is on.
planif=# SET enable_seqscan TO on;
Time: 0.161 ms
planif=# SELECT * FROM t1 WHERE c1>500;
Time: 518.373 ms
planif=# SET enable_seqscan TO off;
Time: 0.166 ms
planif=# SELECT * FROM t1 WHERE c1>500;
Time: 566.167 ms
planif=# \o
```

Soit 518 millisecondes en parcours séquentiel et 566 millisecondes en parcours d'index. Le planificateur ne s'est pas trompé dans ce cas-ci. Cependant, cela ne sera pas tout le temps le cas. Il peut arriver ceci :

```
planif=# \o /dev/null
planif=# SET enable_seqscan TO on;
Time: 0.170 ms
planif=# SELECT * FROM t1 WHERE c1>400000;
Time: 356.721 ms
planif=# SET enable_seqscan TO off;
Time: 0.183 ms
planif=# SELECT * FROM t1 WHERE c1>400000;
Time: 350.970 ms
planif=# \o
```

Il a surévalué le coût de l'index. C'est assez logique quand la table et l'index tiennent en mémoire et qu'ils finissent inévitablement à rester en cache. Dans ce cas, un accès séquentiel et un accès aléatoire ont le même coût. Cela permet de baisser fortement le paramètre **random_page_cost** :

```
planif=# SET enable_seqscan TO on;
SET
Time: 0.182 ms
planif=# SET random_page_cost to 2;
SET
Time: 0.234 ms
planif=# EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM t1 WHERE c1>400000;
                                    QUERY PLAN
-----------------------------------------------------------------------------
 Index Scan using t1_c1_idx on t1  (cost=0.42..17615.57 rows=601837 width=16)
                         (actual time=0.029..104.273 rows=600000 loops=1)
   Index Cond: (c1 > 400000)
```

```
   Buffers: shared hit=5466
 Total runtime: 135.202 ms
(4 rows)

Time: 135.760 ms
```

Après avoir baissé **random_page_cost**, le planificateur préfère passer par un parcours d'index.

L'astuce consistant à abaisser la valeur du paramètre **random_page_cost** fonctionne très bien. Il faut cependant faire cela en toute conscience. Cela n'a un intérêt que dans trois cas principaux :

- vous disposez de disques très rapides ;
- vous disposez de disques SSD (sur ces disques équivalents à de la mémoire, l'accès à une page aléatoire est aussi rapide que l'accès à une page séquentielle, car il n'y a pas de déplacement physique, mécanique de la tête de lecture) ;
- la base est suffisamment petite pour tenir en mémoire.

Dans tous les autres cas, il peut être dangereux de descendre fortement la valeur de ce paramètre, car on obtiendrait l'effet contraire (utilisation d'un parcours d'index, alors qu'un parcours séquentiel serait gagnant).

6 Parcours avec deux filtres

Essayons maintenant d'ajouter un filtre sur la deuxième colonne :

```
planif=# EXPLAIN (ANALYZE) SELECT * FROM t1 WHERE c1<500 AND c2 LIKE
'abcd%';
                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using t1_c1_idx on t1  (cost=0.42..26.41 rows=1 width=37)
                           (actual time=0.314..0.314 rows=0 loops=1)
   Index Cond: (c1 < 500)
   Filter: (c2 ~~ 'abcd%'::text)
   Rows Removed by Filter: 499
 Total runtime: 0.343 ms
(5 rows)
```

La planificateur réalise un parcours d'index en utilisant la seule condition **c1<500**, ce qui lui permet de rejeter un grand nombre de lignes. Pendant ce parcours, il lit aussi la table pour exécuter le filtre supplémentaire (**c2 LIKE 'abcd%'**) et s'assurer que les enregistrements finaux sont bien visibles pour la transaction. Ce filtre supplémentaire sur **c2** lui fait rejeter ici 499 lignes supplémentaires.

Essayons maintenant avec un index sur **c2** :

```
planif=# CREATE INDEX ON t1(c2);
CREATE INDEX
planif=# EXPLAIN (ANALYZE) SELECT * FROM t1 WHERE c1>500 AND c2 LIKE
'abcd%';
                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using t1_c2_idx on t1  (cost=0.42..8.45 rows=100 width=37)
                           (actual time=0.066..0.112 rows=22 loops=1)
   Index Cond: ((c2 >= 'abcd'::text) AND (c2 < 'abce'::text))
   Filter: ((c1 > 500) AND (c2 ~~ 'abcd%'::text))
 Total runtime: 0.147 ms
(4 rows)
```

Cette fois, le planificateur a choisi d'utiliser le nouvel index, ce qui permet une amélioration très importante de la durée d'exécution de la requête. Il faut aussi noter l'astuce du planificateur qui, au lieu de chercher les textes qui commencent par la chaîne « abcd », choisit d'utiliser les opérateurs **>=** et **<=** pour transformer la condition en **c2 >= 'abcd' AND c2 < 'abce'**. Cela lui permet de profiter de la pleine puissance des index. Il n'est plus nécessaire de faire un parcours complet, il peut ne faire qu'un parcours partiel.

Changeons maintenant un peu le filtre :

```
planif=# EXPLAIN (ANALYZE) SELECT * FROM t1 WHERE c1<500 OR c2 LIKE 'abcd%';
                                QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on t1  (cost=107.43..16749.07 rows=6241 width=37)
                        (actual time=0.275..0.855 rows=651 loops=1)
   Recheck Cond: ((c1 < 500) OR (c2 ~~ 'abcd%'::text))
   Filter: ((c1 < 500) OR (c2 ~~ 'abcd%'::text))
   ->  BitmapOr  (cost=107.43..107.43 rows=5241 width=0)
                 (actual time=0.220..0.220 rows=0 loops=1)
         ->  Bitmap Index Scan on t1_c1_idx  (cost=0.00..99.74 rows=5241 width=0)
                                    (actual time=0.147..0.147 rows=499 loops=1)
               Index Cond: (c1 < 500)
         ->  Bitmap Index Scan on t1_c2_idx1  (cost=0.00..4.57 rows=1 width=0)
                                  (actual time=0.071..0.071 rows=152 loops=1)
               Index Cond: ((c2 ~>=~ 'abcd'::text) AND (c2 ~<~ 'abce'::text))
 Total runtime: 0.967 ms
(9 rows)
```

Cette fois, nous voulons toutes les lignes pour lesquelles la valeur de la colonne **c1** est inférieure à 500 et toutes les lignes pour lesquelles la valeur de la colonne **c2** commence par la chaîne « abcd ». PostgreSQL dispose d'un index sur **c1** et d'un index sur **c2**. Comme le nombre de lignes à récupérer est très faible dans les deux cas, il va utiliser les index pour créer deux bitmaps et combiner ces bitmaps en mémoire. Le bitmap est un champ de bits : un bit à 1 indique que le test est vrai, un bit à zéro qu'il est faux. Il suffit ensuite de combiner ces deux champs de bits en mémoire suivant le lien entre les filtres (opérations binaires AND ou OR) pour obtenir rapidement les lignes qui correspondent à l'expression globale. Comme ce n'est pas simple à expliquer ainsi, quelques dessins seront peut-être plus parlants.

Un extrait du contenu des deux index est donné en figure 1.

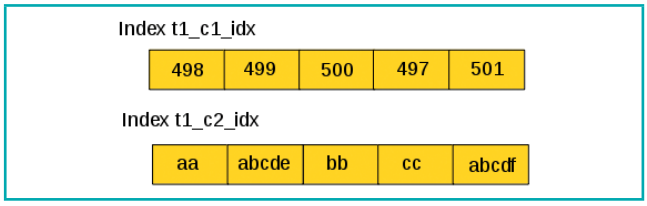


Fig. 1 : Contenu des index c1 et c2 : la ligne X contient les valeurs 498 et « aa ». La ligne X+1 contient les valeurs 499 et « abcde ». Et ainsi de suite...

PostgreSQL doit combiner les deux index avec ce filtre : **`c1<500 OR c2 LIKE 'abcd%'`**. Il va donc créer deux représentations bitmap en mémoire où un 1 signifie que la valeur correspond au filtre et 0 l'inverse. La figure 2 montre ce que cela donne pour le premier index et la figure 3 montre ce que cela donne pour le deuxième index.

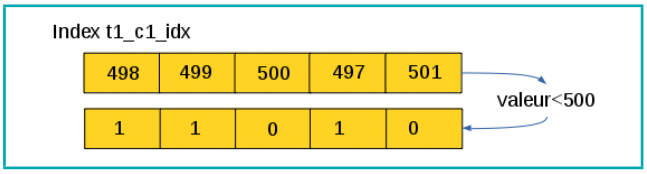


Fig. 2 : Représentation bitmap du premier index. La valeur 498 est bien inférieure à 500, sa correspondance dans le bitmap est donc la valeur 1. La valeur 500 est égale à 500, pas inférieure, donc le bitmap contient un 0 pour cet enregistrement. Cette opération correspond au nœud « Bitmap Index Scan on t1_c1_idx ».

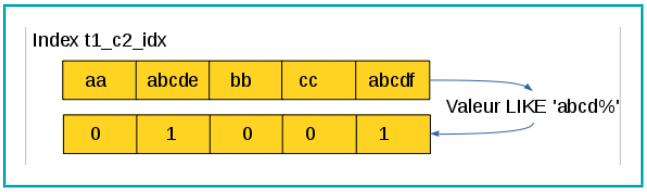


Fig. 3 : Représentation bitmap du deuxième index. Cette opération-là correspond au nœud « Bitmap Index Scan on t1_c2_idx ».

Maintenant, PostgreSQL n'a plus qu'à combiner les deux bitmaps en mémoire pour savoir quels enregistrements satisfont les deux filtres (voir figure 4).

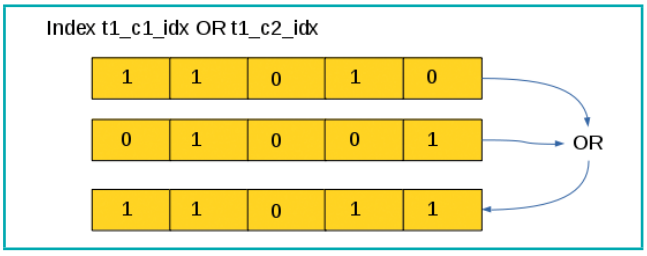


Fig. 4 : Combinaison des deux bitmaps en mémoire. PostgreSQL sait que les lignes 1, 2, 4 et 5 satisfont au filtre, contrairement à la ligne 3. Ce travail correspond au nœud « BitmapOr ».

Le concept d'index bitmap est très intéressant quand on veut combiner plusieurs index rapidement.

Cependant, ce n'est pas le seul intérêt de ces index. L'autre intérêt est la façon dont ils sont parcourus : l'exécuteur parcourt tous les éléments de l'index (ou des index, comme dans le cas ci-dessus), puis va lire la table pour chaque enregistrement sélectionné. Cela occasionne moins de déplacements de la tête de lecture, ce qui fait qu'un Bitmap Heap Scan / Bitmap Index Scan est souvent utilisé quand le nombre estimé de lignes à lire est à la fois trop important pour qu'un simple parcours d'index ne soit intéressant, et trop petit pour justifier un parcours séquentiel de la table.

7 Parcours d'index couvrant

Il existe un dernier parcours d'index. Ce dernier est intéressant quand les colonnes filtrées et renvoyées sont dans l'index. On appelle ce type d'index un index couvrant (car il couvre tous les besoins de la requête). C'est le cas avec cette requête :

```
planif=# EXPLAIN SELECT c1 FROM t1 WHERE c1<500;
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Only Scan using t1_c1_idx on t1  (cost=0.42..13.53 rows=520 width=4)
   Index Cond: (c1 < 500)
(2 rows)
```

Pour que ce type de parcours soit possible, il faut que le **`VACUUM`** ait enregistré les informations nécessaires dans le fichier FSM. Un **`VACUUM`** fréquent est une bonne chose, mais un **`ANALYZE`** peut aussi faire le travail. Ceci explique le fait qu'on n'ait pas eu besoin d'exécuter un **`VACUUM`** pour que la magie opère. Mais en règle générale, on aura besoin d'un **`VACUUM`**.

8 Revue des différents types de parcours de relations

Cet article nous a permis de voir que le planificateur de PostgreSQL disposait de plusieurs types de parcours de relations, suivant les besoins. Nous avons vu aussi que le nombre de lignes à récupérer déterminait assez fortement le type de parcours sélectionné par le planificateur. Voici un autre exemple le montrant un peu plus simplement.

Commençons par créer une table, sur laquelle nous allons désactiver l'autovacuum :

```
planif=# CREATE TABLE t2 (lettre char(1), contenu text);
CREATE TABLE
planif=# ALTER TABLE t2 SET (autovacuum_enabled=off);
ALTER TABLE
```

Ajoutons-lui un index sur la colonne **`lettre`** :

```
planif=# CREATE INDEX ON t2(lettre);
CREATE INDEX
Insérons quelques données en exécutant cinq fois la requête ci-dessous :
INSERT INTO t2
  SELECT substring(relname, 1, 1), repeat('x', 250)
  FROM pg_class
  ORDER BY random();
```

Cela nous fera une table de 1510 lignes. Maintenant, nous allons installer une extension très intéressante. Cette extension, appelée **`explanation`**, permet par l'intermédiaire d'une procédure stockée de récupérer un plan d'exécution avec une colonne pour chaque information. Elle est disponible sur le

site d'échange d'extensions de PostgreSQL, pgxn.org (http://www.pgxn.org/dist/explanation/). En fait, nous voulons récupérer seulement le nœud principal du plan d'exécution avec l'estimation du nombre de lignes et celui du coût total. Une fois installée au niveau du système d'exploitation, il faut l'installer sur la base de données pour accéder à sa procédure stockée. Cela se fait avec une requête SQL très simple :

```
planif=# CREATE EXTENSION explanation;
CREATE EXTENSION
```

Voici ce que donne son utilisation à partir d'une requête CTE :

```
planif=# WITH lettres (lettre, nombre) AS (
planif-#   SELECT lettre, COUNT(*)
planif-#   FROM t2
planif-#   GROUP BY 1
planif-# )
planif-# SELECT lettre, nombre, node_type, total_cost, plan_rows
planif-# FROM lettres,
planif-# LATERAL explanation('SELECT * FROM t2 WHERE lettre =''' ||
lettre ||''''')
planif-# WHERE parent_id IS NULL
planif-# ORDER BY nombre;
 lettre | nombre |    node_type     | total_cost | plan_rows
--------+--------+------------------+------------+-----------
 k      |      5 | Bitmap Heap Scan |      56.27 |        33
 i      |     10 | Bitmap Heap Scan |      56.27 |        33
 e      |     10 | Bitmap Heap Scan |      56.27 |        33
 a      |     15 | Bitmap Heap Scan |      56.27 |        33
 d      |     20 | Bitmap Heap Scan |      56.27 |        33
 v      |     20 | Bitmap Heap Scan |      56.27 |        33
 u      |     25 | Bitmap Heap Scan |      56.27 |        33
 _      |     25 | Bitmap Heap Scan |      56.27 |        33
 f      |     30 | Bitmap Heap Scan |      56.27 |        33
 r      |     40 | Bitmap Heap Scan |      56.27 |        33
 s      |     50 | Bitmap Heap Scan |      56.27 |        33
 t      |     55 | Bitmap Heap Scan |      56.27 |        33
 c      |     60 | Bitmap Heap Scan |      56.27 |        33
 p      |   1165 | Bitmap Heap Scan |      56.27 |        33
(14 rows)
```

Le planificateur a travaillé sans statistiques, et son analyse est donc identique quelle que soit la requête exécutée. Mettons à jour les statistiques et relançons la grosse requête :

```
planif=# ANALYZE t2;
ANALYZE
planif=# WITH lettres (lettre, nombre) AS (
planif-#   SELECT lettre, COUNT(*)
planif-#   FROM t2
planif-#   GROUP BY 1
planif-# )
planif-# SELECT lettre, nombre, node_type, total_cost, plan_rows
planif-# FROM lettres,
planif-# LATERAL explanation('SELECT * FROM t2 WHERE lettre =''' ||
lettre ||''''')
planif-# WHERE parent_id IS NULL
planif-# ORDER BY nombre;
 lettre | nombre |    node_type     | total_cost | plan_rows
--------+--------+------------------+------------+-----------
 k      |      5 | Index Scan       |      18.98 |         5
 i      |     10 | Bitmap Heap Scan |      31.91 |        10
 e      |     10 | Bitmap Heap Scan |      31.91 |        10
 a      |     15 | Bitmap Heap Scan |      39.77 |        15
 d      |     20 | Bitmap Heap Scan |      46.34 |        20
 v      |     20 | Bitmap Heap Scan |      46.34 |        20
 _      |     25 | Bitmap Heap Scan |      50.54 |        25
 u      |     25 | Bitmap Heap Scan |      50.54 |        25
 f      |     30 | Bitmap Heap Scan |      54.17 |        30
 r      |     40 | Bitmap Heap Scan |      59.79 |        40
 s      |     50 | Bitmap Heap Scan |      63.01 |        50
 t      |     55 | Bitmap Heap Scan |      64.31 |        55
 c      |     60 | Bitmap Heap Scan |      64.97 |        60
 p      |   1165 | Seq Scan         |      76.12 |      1165
(14 rows)
```

Nous constatons que le type de parcours dépend fortement du nombre de lignes :

- très peu de lignes, et c'est un parcours d'index ;
- un peu plus de lignes et c'est un parcours de bitmap construit en mémoire ;
- beaucoup de lignes et c'est un parcours séquentiel.

9 Autres types de parcours

PostgreSQL ne parcourt pas que les relations. Il existe plusieurs autres types de parcours, bien moins courants, concernant d'autres types d'objets :

- **TidScan**, parcours d'identifiants de lignes ;
- **SubqueryScan**, parcours des résultats d'une sous-requête ;
- **FunctionScan**, parcours de lignes de résultats d'une fonction renvoyant plusieurs lignes (voir plus bas pour quelques détails) ;
- **ValuesScan**, parcours de valeurs (voir plus bas pour quelques détails) ;
- **CteScan**, parcours d'une table CTE (voir plus bas pour quelques détails) ;
- **WorkTableScan** ;
- **ForeignScan**, parcours d'une table distante.

Nous allons illustrer trois cas.

Le premier cas concerne le nœud appelé **Values Scan**. Son nombre de lignes est exact étant donné que les lignes sont fournies à la commande, et son coût est très petit vu qu'aucune opération de lecture sur disque n'est nécessaire.

```
planif=# EXPLAIN VALUES (1), (2);
                          QUERY PLAN
--------------------------------------------------------------
 Values Scan on "*VALUES*"  (cost=0.00..0.03 rows=2 width=4)
(1 row)
```

Pour le deuxième cas (les fonctions SRF), nous obtenons un nœud **Function Scan**. Le nombre estimé de lignes est

une valeur par défaut (1000), sauf dans le cas des fonctions utilisateurs, où il est possible d'indiquer le nombre de lignes renvoyées (généralement, ce sera une moyenne, et donc pas une valeur exacte... Mais une valeur moyenne est toujours préférable à une valeur interne sans rapport avec le fonctionnement de cette procédure stockée).

Comme exemple, nous allons créer une procédure stockée renvoyant toujours trois lignes (les valeurs entières 1 à 3) :

```
planif=# CREATE FUNCTION f1()
planif-# RETURNS SETOF integer
planif-# LANGUAGE plpgsql AS
planif-# $$
planif-# BEGIN
planif-# RETURN NEXT 1;
planif-# RETURN NEXT 2;
planif-# RETURN NEXT 3;
planif-# END
planif-# $$;
CREATE FUNCTION
planif=# EXPLAIN SELECT * FROM f1();
                        QUERY PLAN
-----------------------------------------------------------
 Function Scan on f1  (cost=0.25..10.25 rows=1000 width=4)
(1 row)
```

Nous obtenons bien le nœud **Function Scan**, mais le nombre de lignes estimé est à 1000, ce qui est très loin de la vérité. Sans indication, le planificateur utilise cette valeur codée en dur. Il est cependant possible de lui indiquer que la fonction renvoie généralement plutôt X lignes. Pour cela, il faut utiliser la clause **ROWS** (disponible depuis la version 8.3) :

```
planif=# ALTER FUNCTION f1() ROWS 3;
CREATE FUNCTION
planif=# EXPLAIN SELECT * FROM f1();
                      QUERY PLAN
-------------------------------------------------------
 Function Scan on f1  (cost=0.25..0.28 rows=3 width=4)
(1 row)
```

Le nombre de lignes est bien de 3, l'estimation du coût total a fortement chuté.

Bien sûr, en cas de filtre, le planificateur n'a aucun moyen d'estimer correctement le nombre de lignes suite à l'application du filtre.

Enfin, en ce qui concerne le troisième cas, un **CTE Scan** n'apparaît qu'à l'utilisation d'une requête CTE (acronyme de *Common Table Expression*). Ce type de requête apparaît avec la version 8.4 de PostgreSQL. Elle permet de définir une ou plusieurs pseudo-table(s) dans une requête et de l'(es) utiliser dans la même requête. En voici un exemple :

```
planif=# EXPLAIN WITH tmp AS (SELECT c1, count(*) AS nb FROM t1 GROUP
BY c1)
planif-# SELECT * FROM tmp WHERE nb>2;
                                QUERY PLAN
-------------------------------------------------------------------
 CTE Scan on tmp  (cost=49317.43..71817.43 rows=333333 width=12)
   Filter: (nb > 2)
   CTE tmp
     ->  GroupAggregate  (cost=0.42..49317.43 rows=1000000 width=4)
           ->  Index Only Scan using t1_c1_idx on t1
                         (cost=0.42..34317.43 rows=1000000 width=4)
(5 rows)
```

La pseudo-table a pour nom **tmp**. Elle « contient » les données correspondant à la requête **SELECT c1, count(*) AS nb FROM t1 GROUP BY c1**, ce qui permet de ne récupérer que les lignes dont le **count(*)** est supérieur à 2. Pour être franc, l'exemple est un peu simpliste et il n'y a pas réellement besoin d'une requête CTE pour obtenir le bon résultat. Cette requête permet d'obtenir le même résultat :

```
SELECT c1, count(*) AS nb FROM t1 GROUP BY c1 HAVING count(*) > 2;
```

Pour en terminer avec ces différents types de parcours, il est à noter que le parcours de table distante est optimisable directement par le *Foreign Data Wrapper*. Ce dernier peut en plus fournir des indications au planificateur, comme le nombre estimé de lignes en retour, ou le coût de récupération des lignes. Mais là, c'est clairement au niveau du Foreign Data Wrapper que cela se joue et les performances seront évidemment différentes entre un Foreign Data Wrapper bien écrit et un mal écrit.

Conclusion

C'est ainsi que se termine cette première partie sur l'étude du planificateur. Nous n'avons vu que les parcours, notamment de relations. La prochaine partie se concentrera sur les différents types de jointures. ■

Événement

pgday.fr 2014 les 5 et 6 juin à Toulon

Après le gros succès de l'édition 2013 à Nantes, l'association PostgreSQL.fr propose sa nouvelle édition du pgday.fr à Toulon. Contrairement à l'année dernière où l'événement n'avait duré que le temps d'une journée, cette édition 2014 se tiendra sur deux jours, les 5 et 6 juin. Une centaine de visiteurs sont attendus pour échanger autour de PostgreSQL et de ses projets associés. Bruce Momjian sera présent et lancera l'événement avec la keynote d'ouverture.

Il est possible de s'enregistrer pour participer à cet événement important, vu qu'il s'agit des dix ans de l'association française (http:///www.pgday.fr).

Pour suivre l'actualité sur cet événement, abonnez-vous au flux Twitter **@PGDAY_FR** ou au flux RSS (http://www.pgday.fr/rss.xml).

ASTERISK EN ROUTEUR D'APPELS : PAR TOUTATIS, ÇA CAPTE MAL DANS CE TUNNEL !

par Emile iMil Heitor [pour GCU-Squad! canal historique et la secte des serviettes de main oranges]

« Bonjour, êtes-vous équipé en fenêt*clic*-Bienvenue chez iMil, si vous êtes un démarcheur, restez en ligne, un opérateur va vous répondre... ». Avouez, avouez que vous n'en pouvez plus de recevoir des coups de fil anonymes de vendeurs de fenêtres et autres démarcheurs téléphoniques. Eh bien pour ça aussi, UNIX a une solution, elle s'appelle Asterisk.

1 La VoIP, c'est pas si compliqué

Bon ok, c'est pas si compliqué, mais ça peut vite devenir agaçant. J'y reviendrai plus tard dans l'article. N'en reste pas moins qu'il existe, sur nos UNIX modernes, un logiciel qui, de plus en plus, dame le pion à nombre de solutions de Voix sur IP propriétaires, je veux bien entendu parler d'Asterisk.

Asterisk est essentiellement connu pour sa fonctionnalité d'IPBX, ou commutateur téléphonique privé utilisant le protocole IP, mais le logiciel est capable de bien plus ; en particulier, Asterisk peut se comporter comme un routeur d'appels, un *softswitch*, n'en déplaise aux vendeurs de tels serveurs propriétaires dont le dédain envers les solutions libres me rappelle des souvenirs... [/rant]

Dans cet article, nous mettrons l'accent sur la fonctionnalité phare, nous utiliserons Asterisk comme l'IPBX du foyer et nous évertuerons à faire de la vie des démarcheurs téléphoniques un véritable cauchemar.

2 Mais la théorie, c'est... pffffff

Je vous l'accorde. Mais dans le monde tortueux de la voix, il est nécessaire de disposer d'un minimum de notions, car on ne communique pas avec la voix comme on affiche une page web. Malheureusement.

LE protocole inévitable lorsqu'on évoque la VoIP, c'est SIP. Abréviation de *Session Initiation Protocol*, SIP, normalisé par l'IETF, a pris le pas sur le protocole H323 issu du monde des télécoms. Il est important de comprendre que SIP, seul, ne transporte pas les données en soi ; aucun morceau de voix n'est véhiculé par ce protocole, cette tâche est reléguée au protocole RTP, et nous verrons que les pires casse-tête émanent de ce fonctionnement bicéphale. Ainsi, lorsqu'on compose un numéro de téléphone avec un terminal SIP, une foule d'informations sont échangées, et parmi elles, l'adresse IP des serveurs de média, dans le meilleur des cas les terminaux eux-mêmes. S'établit alors une connexion entre les terminaux ; c'est via cette dernière que vous pouvez écouter les dernières péripéties de votre grand-mère autour du monde.

Rentrons un peu plus loin dans le vif du sujet.

2.1 L'enregistrement

Un terminal SIP, pour être reconnu par le logiciel de commutation, doit « s'enregistrer », comprendre que ce dernier dispose le plus souvent d'un identifiant, classiquement un couple *login / password*. L'authentification s'effectue par le biais du protocole SIP vers un *SIP Registrar*, l'équipement en charge de tenir la base de données (quelle que soit sa forme) des terminaux. Voici à quoi ressemble un échange d'enregistrement :

- Le terminal fait une demande d'enregistrement : *REGISTER* ;
- Le SIP Registrar renvoie un code 401 (*Unauthorized*) et annonce la méthode d'authentification, ainsi qu'une chaîne de caractères qui permettra de créer un hash et sécuriser l'échange ;
- Le terminal, une fois ces éléments connus, va renvoyer une requête de type *REGISTER* avec les informations

nécessaires à son enregistrement (utilisateur, IP, mot de passe hashé, algorithme...) ;

- Le registrar renvoie alors une réponse *200 OK*, le terminal est désormais connu.

Voici une trace SIP classique qui illustre ce comportement :

- 192.168.1.10 est le terminal, ici un téléphone SIP de marque SNOM ;

- 192.168.2.1 est le SIP registrar.

2.1.1 Le terminal envoie la demande d'enregistrement

```
REGISTER sip:192.168.2.1 SIP/2.0
Via: SIP/2.0/UDP 192.168.1.10:2051;branch=z9hG4bK-gmdpiz3f19hu;rport
From: "iMil" <sip:snom300@192.168.2.1>;tag=g52o3lkawt
To: "iMil" <sip:snom300@192.168.2.1>
Call-ID: 3c4210ba5a55-09spjcv2gxwh@snom300-000413254E2B
CSeq: 2394 REGISTER
Max-Forwards: 70
Contact: <sip:snom300@192.168.1.10:2051;line=1gvc26v8>;flow-
id=1;q=1.0;+sip.instance="<urn:uuid:c049091f-8a4a-433c-9253-1c5aba144
02d>";audio;mobility="fixed";duplex="full";description="snom300";acto
r="principal";events="dialog";methods="INVITE,ACK,CANCEL,BYE,REFER,OP
TIONS,NOTIFY,SUBSCRIBE,PRACK,MESSAGE,INFO"
User-Agent: snom300/6.0.3
Supported: gruu
Allow-Events: dialog
X-Real-IP: 192.168.1.10
WWW-Contact: <http://192.168.1.10:80>
WWW-Contact: <https://192.168.1.10:443>
Expires: 0
Content-Length: 0
```

2.1.2 Le registrar refuse l'enregistrement et donne les informations nécessaires dans l'entête WWW-Authenticate

```
SIP/2.0 401 Unauthorized
Via: SIP/2.0/UDP 192.168.1.10:2051;branch=z9hG4bK-gmdpiz3f19hu;received=192.168.1
.10;rport=2051
From: "iMil" <sip:snom300@192.168.2.1>;tag=g52o3lkawt
To: "iMil" <sip:snom300@192.168.2.1>;tag=as32c7322b
Call-ID: 3c4210ba5a55-09spjcv2gxwh@snom300-000413254E2B
CSeq: 2394 REGISTER
Server: Asterisk PBX 10.12.4
Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, SUBSCRIBE, NOTIFY, INFO, PUBLISH
Supported: replaces, timer
WWW-Authenticate: Digest algorithm=MD5, realm="asterisk", nonce="59c26ba4"
Content-Length: 0
```

2.1.3 Le terminal utilise l'algorithme et le hash fournis afin de renouveler la demande

```
REGISTER sip:192.168.2.1 SIP/2.0
Via: SIP/2.0/UDP 192.168.1.10:2051;branch=z9hG4bK-1uu4fahipil5;rport
From: "iMil" <sip:snom300@192.168.2.1>;tag=g52o3lkawt
To: "iMil" <sip:snom300@192.168.2.1>
Call-ID: 3c4210ba5a55-09spjcv2gxwh@snom300-000413254E2B
CSeq: 2395 REGISTER
Max-Forwards: 70
Contact: <sip:snom300@192.168.1.10:2051;line=1gvc26v8>;flow-
id=1;q=1.0;+sip.instance="<urn:uuid:c049091f-8a4a-433c-9253-1c5aba144
02d>";audio;mobility="fixed";duplex="full";description="snom300";acto
r="principal";events="dialog";methods="INVITE,ACK,CANCEL,BYE,REFER,OP
TIONS,NOTIFY,SUBSCRIBE,PRACK,MESSAGE,INFO"
User-Agent: snom300/6.0.3
Supported: gruu
Allow-Events: dialog
X-Real-IP: 192.168.1.10
WWW-Contact: <http://192.168.1.10:80>
WWW-Contact: <https://192.168.1.10:443>
Authorization: Digest username="snom300",realm="asterisk",nonce="59c2
6ba4",uri="sip:192.168.2.1",response="deadbeeff00b42",algorithm=md5
Expires: 0
Content-Length: 0
```

2.1.4 Le registrar accepte l'enregistrement

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP 192.168.1.10:2051;branch=z9hG4bK-o0jvzcoumuk8;received=192.168.1.10;rpo
rt=2051
From: "iMil" <sip:snom300@192.168.2.1>;tag=s91dzemvls
To: "iMil" <sip:snom300@192.168.2.1>;tag=as5a6ef160
Call-ID: 3c421e6f5f37-y33h2d0xes2s@snom300-000413254E2B
CSeq: 2396 REGISTER
Server: Asterisk PBX 10.12.4
Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, SUBSCRIBE, NOTIFY, INFO, PUBLISH
Supported: replaces, timer
Expires: 3600
Contact: <sip:snom300@192.168.1.10:2051;line=1gvc26v8>;expires=3600
Date: Sun, 16 Feb 2014 18:23:08 GMT
Content-Length: 0
```

Dans le monde de la VoIP, on représente classiquement ces échanges à l'aide d'un *Call Flow*. Considérant la verbosité d'une trace SIP, l'analyse et l'éventuel débogage d'un dysfonctionnement sont rendus plus accessibles par la visualisation d'un tel graphique. En outre, le fameux logiciel d'analyse réseau Wireshark, permet de générer de tels graphes à l'aide du menu Telephony. Le rendu étant quelque peu étriqué, nous y préférerons, pour cet article, un graphe un peu plus explicite (Fig. 1).

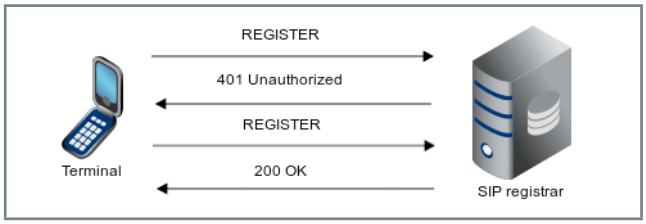


Fig. 1

2.2 L'appel

Une fois nos terminaux enregistrés, la seconde étape délicate consiste en l'appel lui-même. En simplifiant, il se déroule ainsi :

- Une requête *INVITE* est émise par le terminal. Elle est munie d'une SDP, ou *Session Description Protocol*, dans

laquelle le terminal annonce ses capacités en termes de codecs, ainsi que l'adresse IP sur laquelle auront lieu les communications média (ici, la voix) ;

- Le serveur SIP répond immédiatement avec l'entête *Trying* et transmet l'invitation du terminal appelant au terminal appelé. Du fait de ce mode de fonctionnement, on nomme le serveur SIP Proxy ;

- Le terminal appelé (et évidemment enregistré) renvoie une réponse *Ringing* au SIP Proxy, qui relaye ce message au terminal appelant.

La trace de cet échange est plus complexe :

```
INVITE sip:1022@192.168.2.1;user=phone SIP/2.0
Via: SIP/2.0/UDP 192.168.1.10:2051;branch=z9hG4bK-kal713klt5hg;rport
From: "iMil" <sip:snom300@192.168.2.1>;tag=8gad22euu0
To: <sip:1022@192.168.2.1;user=phone>
Call-ID: 3c424be8975e-plr7wrnrxtzq@snom300-000413254E2B
CSeq: 2 INVITE
Max-Forwards: 70
Contact: <sip:snom300@192.168.1.10:2051;line=1gvc26v8>;flow-id=1
P-Key-Flags: keys="3"
User-Agent: snom300/6.0.3
Accept: application/sdp
Allow: INVITE, ACK, CANCEL, BYE, REFER, OPTIONS, NOTIFY, SUBSCRIBE, PRACK,
MESSAGE, INFO
Allow-Events: talk, hold, refer
Supported: timer, 100rel, replaces, callerid
Session-Expires: 3600;refresher=uas
Authorization: Digest username="snom300",realm="asterisk",nonce="deadbeef",uri=
"sip:1022@192.168.2.1;user=phone",response="5949cbdf9e041889612f9da3d03e50f9",a
lgorithm=md5
Content-Type: application/sdp
Content-Length: 370

v=0
o=root 1779652087 1779652087 IN IP4 192.168.1.10
s=call
c=IN IP4 192.168.1.10
t=0 0
m=audio 50056 RTP/AVP 0 8 9 2 3 18 4 101
a=rtpmap:0 pcmu/8000
a=rtpmap:8 pcma/8000
a=rtpmap:9 g722/8000
a=rtpmap:2 g726-32/8000
a=rtpmap:3 gsm/8000
a=rtpmap:18 g729/8000
a=rtpmap:4 g723/8000
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-16
a=ptime:20
a=sendrecv
```

L'*INVITE* est le message initial émis par le terminal. L'entête *Content-Type* indique que la SDP est contenue dans la suite de ce message, cette dernière contient toutes les informations nécessaires à l'établissement de la session média, en particulier :

- **o** permet d'identifier l'initiateur de l'appel,

- **c** présente les informations de connexion,

- **a** tous les attributs média, par exemple les codecs que le terminal supporte.

```
SIP/2.0 100 Trying
Via: SIP/2.0/UDP 192.168.1.10:2051;branch=z9hG4bK-kal713klt5hg;received=192.168
.1.10;rport=2051
From: "iMil" <sip:snom300@192.168.2.1>;tag=8gad22euu0
To: <sip:1022@192.168.2.1;user=phone>
Call-ID: 3c424be8975e-plr7wrnrxtzq@snom300-000413254E2B
CSeq: 2 INVITE
Server: Asterisk PBX 10.12.4
Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, SUBSCRIBE, NOTIFY, INFO,
PUBLISH
Supported: replaces, timer
Session-Expires: 1800;refresher=uas
Contact: <sip:1022@192.168.2.1:5060>
Content-Length: 0
```

Le message *TRYING* est immédiatement renvoyé par le SIP Proxy qui informe le terminal que le processus est en cours. Dans le même temps, le SIP Proxy relaye le message d'invite vers le terminal appelé, ce dernier contient les informations qui ont été transmises par l'appelant possiblement modifiées par le proxy, comme nous le verrons plus tard. Le terminal appelé informe le proxy qu'il est en train de sonner (message *180 Ringing*), le message est alors transmis au terminal appelant qui entendra alors le fameux *Ringback Tone*, ou *tonalité de retour d'appel* :

```
SIP/2.0 180 Ringing
Via: SIP/2.0/UDP 192.168.1.10:2051;branch=z9hG4bK-kal713klt5hg;received=192.168.1
.10;rport=2051
From: "iMil" <sip:snom300@192.168.2.1>;tag=8gad22euu0
To: <sip:1022@192.168.2.1;user=phone>;tag=as60eb8c3c
Call-ID: 3c424be8975e-plr7wrnrxtzq@snom300-000413254E2B
CSeq: 2 INVITE
Server: Asterisk PBX 10.12.4
Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, SUBSCRIBE, NOTIFY, INFO, PUBLISH
Supported: replaces, timer
Session-Expires: 1800;refresher=uas
Contact: <sip:1022@192.168.2.1:5060>
Content-Length: 0
```

Lorsque l'utilisateur décroche le terminal appelé, un message *200 OK* est envoyé au SIP proxy, qui à son tour, le transmet au terminal appelé. Dans cette réponse, on trouvera également une SDP qui indique au terminal appelant les coordonnées et capacités de l'appelant. Enfin, l'appelant acquittera cette réponse à l'aide d'un message *ACK*. L'établissement de la session média, et donc de la conversation, peut alors avoir lieu :

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP 192.168.1.10:2051;branch=z9hG4bK-kal713klt5hg;received=192.168.1
.10;rport=2051
From: "iMil" <sip:snom300@192.168.2.1>;tag=8gad22euu0
To: <sip:1022@192.168.2.1;user=phone>;tag=as60eb8c3c
Call-ID: 3c424be8975e-plr7wrnrxtzq@snom300-000413254E2B
CSeq: 2 INVITE
Server: Asterisk PBX 10.12.4
Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, SUBSCRIBE, NOTIFY, INFO, PUBLISH
```

```
Supported: replaces, timer
Session-Expires: 1800;refresher=uas
Contact: <sip:1022@192.168.2.1:5060>
Content-Type: application/sdp
Require: timer
Content-Length: 314

v=0
o=root 485727824 485727824 IN IP4 192.168.1.11
s=Asterisk PBX 10.12.4
c=IN IP4 192.168.1.11
t=0 0
m=audio 19010 RTP/AVP 3 0 8 101
a=rtpmap:3 GSM/8000
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-16
a=silenceSupp:off - - - -
a=ptime:20
a=sendrecv

ACK sip:1022@192.168.2.1:5060 SIP/2.0
Via: SIP/2.0/UDP 192.168.1.10:2051;branch=z9hG4bK-cu4e80h6zf8q;rport
From: "iMil" <sip:snom300@192.168.2.1>;tag=8gad22euu0
To: <sip:1022@192.168.2.1;user=phone>;tag=as60eb8c3c
Call-ID: 3c424be8975e-plr7wrnrxtzq@snom300-000413254E2B
CSeq: 2 ACK
Max-Forwards: 70
Contact: <sip:snom300@192.168.1.10:2051;line=1gvc26v8>;flow-id=1
Content-Length: 0
```

La session RTP s'établira sur un codec commun, et dans le cas présent, la communication média s'effectuera directement d'un terminal à l'autre, sans équipement tiers. On représente cette mise en relation avec le très classique schéma représenté en figure 2.

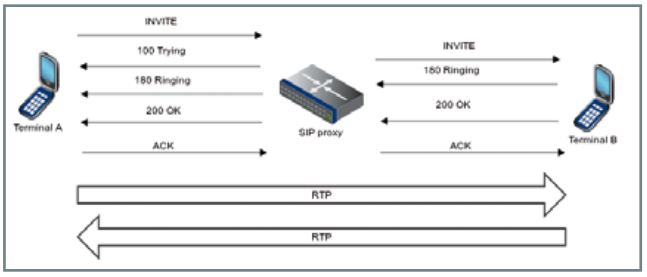


Fig. 2

3 NAT et VoIP == Roquefort et confiture

« Dans le cas présent, la communication média s'effectuera directement d'un terminal à l'autre ». Ah-ah, contemplez avec quelle légèreté je balaye d'un revers de la main une quelconque problématique liée à l'établissement de la session média. Quelle insouciance, quelle naïveté ! Non. La vérité, c'est que l'établissement des médias, dans un réseau domestique, c'est un véritable cauchemar, un instrument de torture du moyen âge, chute de cheveux et assèchement de la peau. Les médias et le NAT, c'est l'ENFER.

Ça y est ? Je vous ai bien motivé là ? On y va.

Les traces SIP précédentes, et en particulier les informations contenues dans les SDP, contiennent les adresses IP des terminaux. Lorsque vous passez nonchalamment un coup de téléphone à travers votre [opérateur]box, l'adresse inscrite dans la SDP est l'IP publique qui vous est affectée par le fournisseur d'accès ; aucun souci en perspective dans ce cas de figure, les équipements communiquent en direct. Mais que se produira-t-il si votre terminal SIP dispose d'une IP de type RFC1918, non routable (dite « privée ») ? Eh bien c'est simple : vous n'entendrez rien. Le terminal appelé sonnera, vous entendrez le *ringback tone*, le « décroché » enverra les bonnes séquences, mais rien. Pas un son. Le néant. Pourquoi ? Parce que le terminal appelé essaiera en vain d'établir une connexion RTP vers une adresse qui n'est pas accessible depuis son réseau. Et c'est là que les ennuis commencent. En effet, comme nous l'avons vu précédemment, le protocole SIP ne transporte que la signalisation, la voix - le média - transite par le biais d'un autre protocole, RTP, qui pour simplifier la tâche, utilisera un port au hasard. Oui parce que sinon c'est trop simple.

Il existe une multitude de techniques, plus ou moins efficaces, pour contourner ce casse-tête. La plus simple consiste à utiliser du *tunneling* IP de bout en bout ; cette solution est convenable dans le cadre d'une téléphonie privée, mais n'adresse pas la téléphonie publique.

Une autre consiste en l'utilisation d'un proxy RTP sur l'équipement public faisant office de passerelle sur votre réseau ; c'est par exemple cette technique que j'ai utilisé pour bypasser la RougeBox fournie par mon opérateur. Dans le cadre de notre article, j'ai opté pour une solution assez

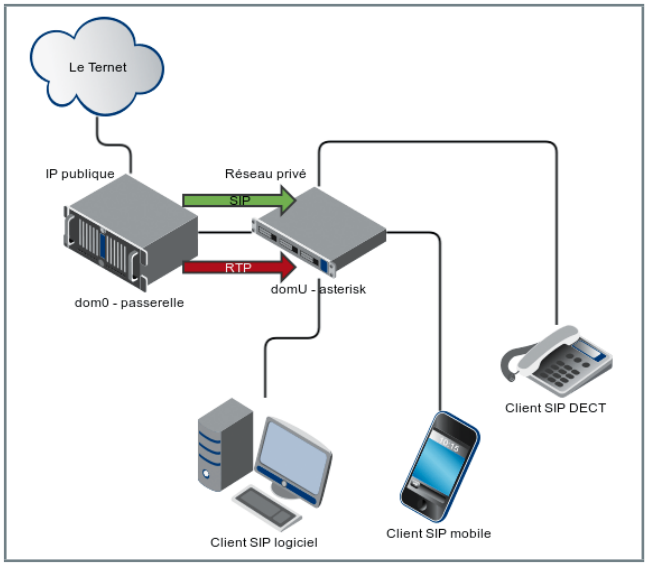


Fig. 3

commune à base d'iptables et de directives appropriées d'Asterisk (Fig. 3). Concrètement, nous allons faire arriver le port 5060 (le port SIP standard) de la passerelle sur une machine virtuelle numérotée sur un adressage « privé ». C'est sur cette machine virtuelle que tourne notre IPBX.

De la même façon, Asterisk définit une plage de ports dédiés au RTP, nous allons rediriger ces ports depuis la passerelle vers notre VM.

Ces règles se placent dans la chaîne **PREROUTING** et sont relativement simples :

- Redirection du protocole SIP :

```
# iptables -t nat -A PREROUTING -i eth0
-p udp -m udp --dport 5060 -j DNAT --to-
destination 192.168.2.1
```

- Redirection du flux RTP, que le fichier **rtp.conf** d'Asterisk définit comme transitant sur les ports 10000 à 20000 :

```
# iptables -t nat -A PREROUTING -i eth0 -p
udp -m udp --dport 10000:20000 -j DNAT --to-
destination 192.168.2.1
```

C'est tout ? Eh-eh-eh. Non. Quand bien même le flux RTP est-il transporté jusqu'à notre réseau domestique, la problématique de l'adresse média présente dans la SDP reste entière. Comment faire ? Modifier la SDP. Point de techniques barbares, Asterisk gère cela nativement à l'aide d'un paramètre bien nommé : **media_address**. Cette variable, que nous allons placer dans le fichier de configuration **sip.conf**, permettra à l'IPBX de remplacer l'adresse média présente dans la SDP par notre adresse publique.

4 Et sinon on fait des trucs un peu ?

Oui, oui, on arrête - presque - là la théorie pour manipuler du fichier de configuration et passer à la pratique, mais comme je vous l'exposais au début de cet article, on n'aborde pas la téléphonie sur IP sans en comprendre les fondements, car si au détour d'un forum quelconque on trouvera probablement des paramètres adaptés, il est toutefois plus agréable de comprendre les objets que l'on manipule.

Asterisk existe sous forme de paquet pour l'intégralité des systèmes UNIX (libres) modernes. Mon IPBX fonctionne sous NetBSD, mais il va sans dire que FreeBSD, OpenBSD, Debian et dérivés, ainsi que Red Hat et dérivés disposent tous d'un paquet (plus ou moins) à jour du logiciel. Ainsi, choisissez la commande qui vous sied parmi **pkgin**, **pkg**, **apt-get**, **aptitude**, **yum** et installez le paquet **asterisk**. De mon côté, il s'agit de la version 10.12.4.

Nous nous concentrerons sur deux fichiers de configuration :

- **sip.conf**, qui contient les informations relatives aux inscriptions SIP entrantes et sortantes ;
- **extensions.conf** décrit ce qu'on appelle en VoIP un *dialplan* : il s'agit des règles de numérotation et de leurs traitements. Nous reviendrons en détails sur la puissance que cachent ces règles.

La première étape de la mise en place de notre système de téléphonie consiste en l'enregistrement de nos divers terminaux. J'utiliserai pour la démonstration un téléphone SIP de marque SNOM et un classique téléphone mobile Android, capable depuis quelques temps de s'enregistrer sur un SIP registrar nativement, sans passer par un logiciel tiers. Pour ceux d'entre vous ne disposant pas d'une version d'Android supportant ce fonctionnement, il est également possible de profiter de la voix sur IP à l'aide du logiciel CSipSimple, disponible sur Google Play mais également sous forme de paquet individuel sur le site du projet. CSipSimple est sous licence GNU GPL v3.

À noter que les téléphones IP SNOM fonctionnent sous GNU/Linux et que depuis leur interface web, il est possible d'obtenir des traces SIP, très pratiques pour les séances de débogage.

Le fichier **sip.conf** se présente sous la forme connue de configuration suivante :

```
[section1]
valeur = clé

[section2]
valeur = clé
```

Ici, les sections représentent les interlocuteurs SIP à l'exception de la section **[general]** qui s'applique à l'ensemble des sections. Voici un fichier **sip.conf** minimal :

```
[general]
nat=no

[poste1]
type=friend
context=desk
username=jeanmi
secret=skinautique
host=dynamic

[poste2]
type=friend
context=desk
username=micheline
secret=peinturesurbois
host=dynamic
```

Comme on le comprend aisément, notre configuration comprend deux utilisateurs, dont le login est spécifié par le champ **username** et le mot de passe par le champ **secret**. La directive **nat=no** s'applique à l'ensemble des postes, mais l'on pourra écraser cette valeur dans chaque section si nécessaire en donnant ponctuellement une valeur différente à **nat=**. La directive **type** définit la catégorie de l'interlocuteur SIP, elle peut prendre trois valeurs :

- **peer**, il s'agit d'une entité SIP à laquelle Asterisk envoie des appels. Typiquement, votre opérateur SIP ;
- **user**, c'est une entité qui place des appels à l'aide d'Asterisk, par exemple un terminal qui ne peut qu'émettre des appels ;
- **friend**, cette entité est un **peer** et un **user**, comprendre qu'elle peut recevoir des appels d'Asterisk ou en émettre.

Le champ **host** est ici défini comme dynamique simplement parce que le terminal s'enregistre seul, inutile donc

d'en préciser l'IP, il se présentera lors de son enregistrement. Enfin, le champ **context** définit le contexte auquel est associé le terminal ; cette variable prendra tout son sens lorsque nous manipulerons le fichier **extensions.conf** dans lequel nous définirons un dialplan.

Munis de ces informations, nous pouvons d'ores et déjà démarrer l'IPBX ou encore recharger sa configuration SIP. Asterisk dispose d'une *Command Line Interface* similaire à celle d'un routeur. Cette dernière peut être accédée à l'aide des commandes **asterisk -r** ou encore **rasterisk**. Pour recharger uniquement la configuration SIP, on utilise la commande **CLI** :

```
asterisk*CLI> sip reload
```

À des fins d'observation, il s'avère souvent judicieux d'activer le mode *debug* de la pile SIP d'Asterisk, ceci s'effectue à l'aide de la commande **CLI** :

```
asterisk*CLI> sip set debug on
SIP Debugging enabled
```

Dès lors que cette configuration sera rechargée, les terminaux de jeanmi et micheline pourront s'enregistrer sur le PBX et s'appeler. Les informations à renseigner sur le client SIP sont généralement :

- Nom d'utilisateur,
- Mot de passe,
- Serveur (implicitement *Sip registrar*).

Pour exemple, voir en figure 4 la configuration de mon téléphone SNOM.

Fig. 4

La configuration SIP sur les mobiles Android est cachée dans les paramètres de téléphonie.

- Démarrez l'application « téléphone »,
- Entrez dans les préférences de l'application (trois points en bas à droite sur une Kit Kat officielle),
- Choisissez *Call Settings*,
- En bas de la liste déroulante, choisissez *SIP Accounts*.

Une fois les configurations appliquées sur vos terminaux, ces derniers devraient être enregistrés sur votre IPBX. Pour vous en convaincre, utilisez la commande Asterisk suivante :

```
korriban*CLI> sip show peers
Name/username     Host          Dyn Forcerport ACL Port   Status Description
poste1/jeanmi     192.168.1.10  D                         49731  Unmonitored
poste2/micheline  192.168.1.11  D                         46327  Unmonitored
```

Une dernière opération est nécessaire pour permettre à nos deux utilisateurs de communiquer : leur attribuer un numéro de téléphone. Pour ce faire, nous allons découvrir le fichier **extensions.conf**.

Ce fichier contient toute l'intelligence du routage d'appels. C'est ici, par le biais des contextes d'appel et des extensions déclarées, que nous pouvons diriger un appel entrant vers la bonne destination, qu'il s'agisse d'un terminal, d'un opérateur SIP externe ou encore... d'un script :)

Pour le moment, nous nous contenterons de déclarer un contexte simple dans lequel nous associerons un numéro de ligne interne à nos terminaux précédemment déclarés dans le fichier **sip.conf**. **extensions.conf** est structuré de la même façon que **sip.conf**, il en est de même pour l'ensemble des fichiers de configuration d'Asterisk :

```
[desk]
exten => 10,1,Dial(SIP/poste1,5)
exten => 11,1,Dial(SIP/poste2,5)
```

Ici, on déclare le contexte **desk** et deux extensions : **10** et **11**. Le numéro de poste est suivi d'un numéro de séquence qu'on incrémente si d'autres opérations sont nécessaires. Par exemple, si l'on souhaite simplement diffuser un message lorsque la ligne 12 est appelée, on aura la séquence suivante :

```
exten => 12,1,Answer()
exten => 12,2,Playback(mon_message)
exten => 12,3,Hangup()
```

Notez que pour des raisons de simplicité, il est souvent préférable d'utiliser le caractère **n** après la séquence 1, qui implique l'incrémentation de chaque séquence consécutive. Nous utilisons, pour établir une communication SIP avec le poste appelé, la fonction **Dial()**, dont le premier paramètre est le poste visé, et le second représente le temps maximal, en secondes, pendant lequel le poste sonnera. Ce dernier paramètre est optionnel ; s'il est omis, le poste sonnera tant que l'utilisateur n'a pas décroché.

À la façon des informations SIP, on recharge les paramètres relatifs au dialplan grâce à la CLI Asterisk :

```
asterisk*CLI> dialplan reload
```

Nos deux postes sont maintenant enregistrés et joignables, les appels peuvent être placés.

5 Dehors, le chaos

C'est à cet instant que tout se corse. Nos utilisateurs devront fatalement pouvoir joindre le monde extérieur, et dans notre cas d'école, cela se déroulera par le biais d'un (ou plusieurs) opérateur(s) de téléphonie.

Les combinaisons sont multiples. On pourra par exemple, dans le cas où votre fournisseur d'accès ne propose pas de facilités SIP, installer une carte de type FXO que sait parfaitement interfacer Asterisk. De telles cartes coûtent une dizaine d'euros sur les sites d'enchères connus. Une telle carte vous permettra de raccorder le RJ11 de votre ligne téléphonique classique et de le faire prendre en charge par Asterisk comme s'il s'agissait d'une interconnexion numérique.

Nous considèrerons que votre fournisseur d'accès est de type GratuiBoite, ou encore RougeBoite, ces derniers fournissant des accès SIP. Il est par ailleurs peu onéreux de nos jours de souscrire à une ligne téléphonique SIP ; chez un hébergeur *low cost* à trois lettres, un tel abonnement proposant quarante destinations gratuites coûte 1€ par mois.

Nous démarrons la configuration du raccordement à notre trunk SIP dans le fichier **sip.conf**. La première étape consiste à renseigner une directive d'enregistrement de notre IPBX vers le fournisseur de téléphonie publique. Nous utilisons pour cela la directive **register**, à placer dans la section **[general]** du fichier **sip.conf**. La syntaxe supportée par cette clé est la suivante :

```
register => [peer?][transport://]user[@domain][:secret[:authuser]]@
host[:port][/extension][~expiry]
```

Chez GratuiBoite, cela se traduit par :

```
register => utilisateur:motdepasse@freephonie.net
```

Chez RougeBoite :

```
register => +3399XXXXXXXXX@ims.mnc010.mcc208.3gppnetwork.
org:motdepasse:NDIXXXXXXXXX.LIBERTALK@sfr.fr@internet.p-cscf.sfr.
net:5064~3600
```

où XXXXXXXXX représente votre numéro de téléphone public.

Et chez les low costeurs à trois lettres :

```
register => 003397994XXXX:motdepasse@sip.ovh.net
```

Afin de permettre le transit des appels, il est nécessaire de créer deux sections supplémentaires à notre configuration SIP, l'une pour les appels sortants, l'autre pour les appels entrants :

Pour GratuiBoite :

```
[freephonie-out]
type=peer
host=freephonie.net
username=XXXXXXXXXX
fromuser=XXXXXXXXXX
secret=motdepasse
fromdomain=freephonie.net

[freephonie-in]
type=user
context=fromfree
host=freephonie.net
```

Pour la RougeBoite, c'est un peu plus compliqué. En effet, cet opérateur a fait basculer son infrastructure VoIP dans un monde aussi inutilement complexe que cauchemardesque à appréhender : l'IMS (*IP Multimedia Subsystem*). Sans s'attarder sur le sujet qui prendrait plusieurs numéros de ce magazine pour en faire le tour, IMS est supposément, je cite :

« une architecture standardisée *Next Generation Network* (NGN) pour les opérateurs de téléphonie, qui permet de fournir des services multimédias fixes et mobiles. » ©Wikipédia.

Pour vous donner un aperçu du sac de nœuds, observez en figure 5 le schéma d'architecture qu'on peut admirer sur la page dédiée à ce standard, toujours sur Wikipédia.

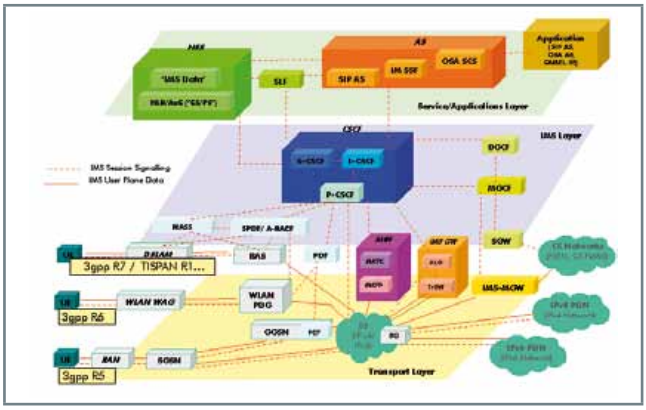


Fig. 5

Ouais. Ça pique.

Fort heureusement, de charitables âmes ont épluché et testé les diverses combinaisons pour enregistrer Asterisk sur un tel réseau et ont proposé les sections suivantes, légèrement modifiées par votre serviteur :

```
[sfr-out]
type=peer
fromdomain=ims.mnc010.mcc208.3gppnetwork.org
fromuser=+3399XXXXXXXXX
defaultuser=NDIXXXXXXXXX.LIBERTALK@sfr.fr
host=internet.p-cscf.sfr.net
insecure=invite,port
remotesecret=motdepasse
auth=NDIXXXXXXXXX.LIBERTALK@sfr.fr:motdepasse@ims.mnc010.
mcc208.3gppnetwork.org
outboundproxy=internet.p-cscf.sfr.net:5064
nat=yes

[sfr-in]
type=user
fromdomain=ims.mnc010.mcc208.3gppnetwork.org
host=internet.p-cscf.sfr.net
insecure=invite,port
context=from-sfr
port=5064
nat=yes
```

Hormis les directives relatives à l'identification, on note une clé inconnue : **`insecure=`**. Ce paramètre vaut **`no`** par défaut et indique à Asterisk que toutes les connexions doivent être authentifiées. Ici, nous spécifions que la requête *INVITE* initiale ne sera pas authentifiée, et que nous ne vérifierons pas le port duquel elle provient.

Enfin, source de tous les maux, nous indiquons qu'il s'agit d'une connexion « natée ». En particulier, afin de simplifier l'établissement de la session RTP, la directive **`nat=yes`** met en place une forme de RTP symétrique, comprendre que le *User Agent* utilisera la même socket et le même port pour émettre et recevoir le RTP.

Mais ce n'est pas la fin de nos ennuis ; telle quelle, cette configuration permettra de placer des appels, mais toujours pas de faire transiter du son. En effet, la SDP fera toujours mention des IP privées de notre réseau interne. La méthode généralement utilisée et fonctionnelle avec la plupart des fournisseurs d'accès SIP consiste à ajouter les paramètres suivants dans la section **`[general]`** du fichier **`sip.conf`** :

```
localnet=192.168.0.0/255.255.0.0 ; notre sous réseau privé
externip=1.2.3.4 ; IP publique de la passerelle
```

Ce qui aura pour effet de masquer les IP incluses dans le sous-réseau **`localnet`** et de les remplacer, dans la SDP, par l'IP **`externip`**. Pour une raison qui m'est inconnue, ce mécanisme pourtant classique ne fonctionne pas avec l'opérateur RougeBox ; si cette configuration permet en effet d'émettre des appels, la réception, elle, ne fonctionne plus. J'ai contourné ce dysfonctionnement en ôtant les directives **`localnet`** et **`externip`** et en leur préférant la variable **`media_address`**, qui laisse intactes les informations VIA des entêtes SIP, mais remplace l'IP média présente dans la SDP par la valeur indiquée.

Cela nous donne une section **`[general]`** de cette forme :

```
context=default
register => +3399XXXXXXXXX@ims.mnc010.mcc208.3gppnetwork.
org:motdepasse:NDIXXXXXXXXX.LIBERTALK@sfr.fr@internet.p-cscf.sfr.net:5064~3600
allowguest=no ; IMPORTANT, n'autorisez pas les accès anonymes !
alwaysauthreject=yes ; On traite les erreurs de mot de passe comme des
utilisateurs inexistants
contactdeny=0.0.0.0/0.0.0.0 ; on refuse toute connexion
contactpermit=91.68.1.28/255.255.255.255 ; à part celles listées ici
contactpermit=192.168.1.0/255.255.255.0 ; et ici
contactpermit=192.168.2.0/255.255.255.0 ; et encore ici
media_address=1.2.3.4 ; l'adresse du proxy RTP
```

```
dtmfmode=rfc2833 ; les codes DTMF respectent la RFC2833
directmedia=no ; par défaut, on définit que les sessions média ne s'établissent
pas directement entre les terminaux
nat=no ; par défaut, on n'active pas de NAT
```

Un poste SIP est alors déclaré de cette façon :

```
[snom300]
type=friend ; le poste émet et reçoit
context=maison ; on utilise le contexte général "maison"
username=snom ; nom d'utilisateur
secret=unpasswordsuperbalaise ; mot de passe
host=dynamic ; le poste s'enregistre seul
directmedia=yes ; de base, pour les appels internes, les sessions RTP
s'établissent directement
```

6 Vous avez demandé le SAMU, ne quittez pas

Nous attaquons la partie à mon sens la plus intéressante de ce setup, le dialplan public et le routage conditionnel des appels.

Nous avons effleuré les possibilités offertes par le fichier **extensions.conf**. En vérité, c'est une stratégie complète et complexe que vous pouvez ici mettre en œuvre par le biais de fonctionnalités somme toute assez simples.

Assurons-nous en premier lieu que nous pouvons désormais passer et recevoir des appels. On compose classiquement un numéro de téléphone public en démarrant la séquence par un 0. C'est ce comportement que nous allons indiquer à Asterisk :

```
exten => _0.,1,Dial(SIP/sfr-out/${EXTEN})
```

Le premier paramètre d'une extension n'est pas obligatoirement un nombre ou une lettre ; en réalité, nous sommes en présence d'une forme de regexp. Précédemment, nous avons utilisé des numéros directs de postes, mais il est également possible d'utiliser des extensions spéciales, par exemple :

- **i** pour une extension invalide,
- **s** lorsque le numéro appelé n'est connu dans aucun contexte, c'est typiquement l'extension utilisée lors de la réception d'appels extérieurs,
- **h** raccrochage,
- **t** *timeout*.

Ou encore, et c'est le cas qui nous occupe ici, d'utiliser des *patterns*. Dans l'exemple ci-dessus, **_0.** signifie n'importe quel numéro qui commence par 0. Un tel numéro sera routé vers le trunk SIP **sfr-out** et c'est toute l'extension qui sera composée (**${EXTEN}**). Si l'on possédait un second opérateur que nous souhaitions utiliser uniquement pour les numéros à destination des États-Unis, nous aurions écrit :

```
exten => _001.,1,Dial(SIP/ovh-out/${EXTEN})
```

Mieux, si l'on souhaite choisir son opérateur en fonction du premier numéro entré :

```
exten => _6.,1,Dial(SIP/ovh-out/${EXTEN:1})
exten => _7.,1,Dial(SIP/freephonie-out/${EXTEN:1})
exten => _8.,1,Dial(SIP/sfr-out/${EXTEN:1})
```

Dans cet exemple, si le numéro tapé commence par 6, on passera par le trunk SIP **ovh-out** et on prendra soin d'effacer le premier chiffre du numéro composé (**${EXTEN:1}**). On devine qu'en composant 7 suivi d'un numéro de téléphone, on passera par **freephonie-out** et 8 par **sfr-out**.

Mais ce n'est pas tout ! Les extensions comprennent une symbolique assez simple mais très puissante :

- **X** un chiffre entre 0 et 9,
- **Y** un chiffre entre 1 et 9,
- **Z** un chiffre entre 2 et 9,
- **.** équivaut à [a-zA-Z0-9]+ en regexp,
- **!** équivaut à [a-zA-Z0-9]* en regexp.

Ces séquences sont obligatoirement préfixées par le symbole _. Voici quelques exemples de séquences possibles :

- **_XX1234** matchera **[0-9][0-9]1234** en regexp,
- **_Z.** matchera **[2-9].+** en regexp,
- **_001!** matchera **001.*** en regexp.

Les possibilités sont énormes.

Nous sommes en mesure de placer des appels extérieurs, mais quid des appels entrants ? Nous l'avons évoqué un peu plus haut, c'est l'extension spéciale **s** qui nous permettra de gérer ce cas :

```
[from-sfr]
exten => s,1,Dial(SIP/snom&SIP/samsungs4)
```

Comprendre que lorsqu'un appel est reçu dans le contexte **from-sfr**, on fait sonner les terminaux **snom** et **samsungs4**. Arrêtez de regarder votre mobile, on doit encore voir deux-trois trucs.

6.1 « Il est aux cabinets »

Mais le nirvana, le summum de la classe américaine, réside dans le fait de totalement informatiser ces appels, gérer de façon fine des appels entrants, et c'est précisément ce que nous permet la puissance du dialplan Asterisk.

Premièrement, il nous est possible de « brancher » un contexte sur un autre. Ceci est très simplement fait grâce à la fonction **Goto()**. Par exemple, si je souhaite renvoyer tout appel entrant vers un SVI (Serveur Vocal Interactif), je remplace notre extension précédente par :

```
exten => s,1,Goto(ivr,s,1)
```

et crée un nouveau contexte **ivr**, traduction anglaise de SVI, mais le nom du contexte n'a aucune importance :

```
exten => s,1,Playback(message_d_accueil)
```

À noter, par défaut les fichiers son disponibles pour Asterisk sont localisés dans **/var/lib/asterisk/sounds** ; le paquet **asterisk** de **pkgsrc** les place dans **/usr/pkg/libdata/asterisk/sounds/en/**. Cet emplacement est évidemment configurable dans le fichier **asterisk.conf**.

L'intérêt du message sonore est évidemment d'attendre un choix de l'utilisateur appelant, ceci est fait à l'aide de la fonction **WaitExten()** :

```
[ivr]
exten => s,1,Answer()
exten => s,n,Playback(message_d_accueil)
exten => s,n,WaitExten()
```

On choisit ensuite le routage de l'appel à l'aide du numéro tapé par l'appelant :

```
exten => 1,1,Goto(callhome,s,1) ; si l'appelant tape 1
exten => 2,1,Goto(spam,s,1) ; s'il tape 2

[callhome]
exten => s,1,Dial(SIP/snom&SIP/samsungs4)
exten => s,1,Hangup()

[spam]
exten => s,1,Playback(spam)
exten => s,n,Hangup()
```

Aah oui, ça commence à prendre forme hein ?

7 Yé né souis pas oune nouméro !

Laissez-moi finir ce tour d'horizon en vous introduisant le concept d'AGI ou *Asterisk Gateway Interface*. Asterisk peut s'interfacer sur bon nombre de langages familiers : C, Python, Perl, PHP, Shell...

Si les possibilités du couple dialplan / AGI sont pratiquement infinies, j'aimerais m'appuyer sur un exemple glané en ligne que j'ai utilisé pour filtrer les appels entrants, chez moi.

L'auteur utilise le système de synthèse vocale de Google, et plus précisément un script AGI, pour « parler » à travers son SVI. L'article étant parfaitement bien écrit, je ne paraphraserai pas l'auteur sur ces colonnes et vous invite à découvrir sa prose, mais vous exposerai simplement une application de ces possibilités à travers un morceau de mon fichier **extensions.conf** :

```
[from-sfr]
exten => s,1,Goto(ivr,s,1)

[ivr]
exten => s,1,Answer()
exten => s,n,agi(googletts.agi,"Pour parler a iMil, tapez 1",fr);
exten => s,n,agi(googletts.agi,"Televendeur, tapez 2.",fr);
exten => s,n,WaitExten()

exten => 1,1,Goto(callhome,s,1)
exten => 2,1,Goto(spam,s,1)
```

Si ce serveur vocal suffit à bloquer la plupart des tentatives de ventes de fenêtres, certains se sont essayés à appuyer sur 2, et ont eu le plaisir d'écouter le fichier **spam.gsm** fourni par défaut dans Asterisk, qui propose une foultitude de produits tels que Vi4gr4, Montres Rxxlex, amitié avec un prince Lybien, ou élargissement de l'entre-jambe.

Ouais, ça m'amuse.

8 On s'rappelle ?

Ce survol du monde fabuleux de la téléphonie libre a simplement effleuré les infinies possibilités que propose cet incroyable outil qu'est Asterisk, et pour finir en beauté, sachez que l'excellent livre « Asterisk, The Definitive Guide » est sous licence *Creative Common* et consultable en ligne, téléchargeable et bien évidemment achetable. À l'aide de ce dernier, qui vit depuis de nombreuses années, vous aurez le loisir de plonger dans des configurations épatantes qui dépassent de très loin les modestes pistes de réflexion exposées dans cet article. ■

RETOUR D'EXPÉRIENCE SUR L'OPTIMISATION D'UN PROJET LIBRE

par Frédéric Le Roy

Aujourd'hui, je vais partager avec vous l'expérience vécue sur un projet libre dans le cadre de ses optimisations. Cet article ne sera pas technique, mais plus un pot-pourri des leçons que nous avons tirées lors de l'évolution du projet.

1 Le projet

Le projet qui sert de base pour cet article est Domogik, une solution libre de domotique. Afin de mieux appréhender la suite, voici quelques détails importants :

- Le langage du projet est Python.
- Le projet a été conçu initialement pour être modulaire via l'utilisation de plugins pour gérer les différentes technologies liées à la domotique.
- À cause du nombre de technologies différentes et surtout de leurs différences, il n'a pas été simple de créer un modèle de données suffisamment souple dès les premières versions.
- Le projet bénéficie d'une communauté vivante avec des personnes francophones et anglophones.
- Le niveau et l'éventail des compétences varient énormément entre les différentes personnes qui ont participé au projet, que ça soit des développeurs du *core*, des développeurs IHM, ou des créateurs de plugins.

2 Un peu d'histoire

À l'heure où j'écris ces lignes, la version 0.4.0 de Domogik est quasiment terminée. Cette version est une sorte de « reboot » du projet : une grande partie du code a été réécrite, le modèle de données a été changé en profondeur et le format des plugins a également fortement évolué. Cet article va donc couvrir en grande partie les choix faits pour cette version. Cette version a pour vocation d'être une base pérenne et solide pour la suite.

3 Les 4 axes

Les optimisations peuvent se classer en 4 axes :

- Celles qui concernent la qualité de la solution (choix techniques, fiabilité du code) ;
- Celles qui concernent la communauté (tout ce qui peut simplifier le travail des contributeurs) ;
- Celles qui concernent le support utilisateur (tout ce qui peut rendre les gens autonomes et nous faire consacrer moins de temps au support) ;
- Celles qui concernent l'infrastructure (maintenance de nos sites).

4 La qualité de la solution

4.1 La base de données

4.1.1 Les moteurs de bases de données

À l'origine du projet, le choix avait été fait de supporter 3 modèles de bases différents : SQLite, MySQL et PostgreSQL.

Afin de permettre ceci, SQLAlchemy avait été choisi comme framework Python pour manipuler la base de données.

Au bout de quelques mois d'utilisation de Domogik par certains utilisateurs, il s'est avéré que le moteur SQLite qui était utilisé par défaut n'était pas assez performant pour gérer des historiques de données importants (par exemple des relevés de température chaque minute). Fonctionnellement, ça marchait, mais en termes de performance, les utilisateurs pouvaient attendre parfois jusqu'à 20 secondes pour afficher une courbe d'historique de la température.

À ce jour, nous ne supportons plus que MySQL et PostgreSQL.

4.1.2 L'optimisation du stockage en base

Ne plus supporter un moteur ne fait pas tout ! Même si sur des machines robustes MySQL et PostgreSQL permettent de très bonnes performances, sur de petites configurations telles des Raspberry Pi avec leurs cartes SD, il s'agit d'une autre histoire (et ce genre de petites configurations a le vent en poupe dans le monde de la domotique).

Si on ne tient pas compte de l'hérésie qui consiste à stocker énormément de données dans une base de données stockée sur une carte SD (que ça soit pour des questions de performance ou de fiabilité dans le temps), il y a une réelle attente de la part de la communauté.

Les principales difficultés (que ça soit en termes de performance ou d'espace disque) sont liées au volume des données stockées, ainsi qu'à la fréquence d'écriture de ces données sur le support physique.

Dans les versions précédentes, nous stockions en base chaque changement de valeur et ce, pour l'éternité. Or, il s'avère que certaines informations n'ont pas besoin d'être historisées pour la majorité des gens, par exemple les changements d'état des interrupteurs et lumières. D'autre part, certaines informations n'ont pas besoin d'avoir un historique important et pour finir, certaines données peuvent être altérées sans que cela ne dérange l'utilisateur.

Nous avons donc implémenté plusieurs fonctionnalités pour optimiser les données en base, chacune étant optionnelle : chaque plugin propose des réglages par défaut et l'utilisateur peut pour chaque *device* adapter les réglages à son besoin.

4.1.2.1 Suppression des doublons

La première méthode implémentée (et surtout non destructive) a été de supprimer les doublons. Prenons le jeu de données suivant (des températures) :

Id	Valeur
1	20,5
2	20,4
3	20,4
4	20,4
5	20,4
6	20,2
7	20,3

Supprimer les doublons consiste à ne garder que la première et la dernière valeur d'une suite de valeurs identiques. Nous obtiendrons donc ici :

Id	Valeur
1	20,5
2	20,4
5	20,4
6	20,2
7	20,3

4.1.2.2 Paramétrage de la durée de rétention des valeurs

Nous avons ajouté 2 paramètres pour configurer la politique de rétention des données :

- Le nombre de valeurs que l'on souhaite garder en base. Par exemple, si on ne souhaite garder que les 100 dernières valeurs, s'il y en a déjà 100 en base, lors de l'ajout d'une nouvelle valeur, la plus ancienne sera supprimée ;
- La durée de rétention en jours. Seuls N jours de données seront stockés et chaque jour, les données d'avant le jour J-N seront purgées.

4.1.2.3 Arrondi des valeurs

Une valeur d'arrondi est définie pour un device (par exemple 1). Pour chaque valeur reçue pour ce device, un calcul est réalisé pour vérifier si la différence entre la valeur reçue et la dernière valeur stockée dépasse l'arrondi. Si l'arrondi est dépassé, on stocke la valeur, sinon on considère qu'elle n'a pas bougé. Ceci altère les données, mais permet de réaliser un important nettoyage dans le cas de capteurs où les données peuvent avoir d'infimes changements. Il est ainsi possible d'économiser plus de 50 % de place pour certains capteurs.

4.1.2.4 Optimisation de la configuration du moteur

Une tâche qui n'a pas encore été faite (mais initiée par la communauté) est l'optimisation du moteur MySQL pour son usage sur un Raspberry Pi.

4.1.3 Le modèle de données

Pendant les premières versions du projet, nous avons essayé d'identifier les faiblesses de notre modèle de données et nous les avons listées avec des exemples liés. Les détailler ici serait inutile, notre modèle de donné étant très spécifique. Toutefois, en plus de l'optimisation du modèle de données, nous avons aussi choisi de déplacer des données qui étaient présentes auparavant dans des fichiers XML à l'intérieur du modèle de données.

Ces fichiers étaient liés à la description des messages xPL (un protocole domotique ouvert, qui a pour but de permettre l'interaction entre différentes technologies), qui décrivent comment intercepter les valeurs des devices et aussi comment émettre des commandes vers les devices. D'autres données, liées aux fonctionnalités proposées par les plugins, étaient aussi stockées dans des fichiers. Le fait d'intégrer ces informations en base de données a permis :

- de clarifier la gestion de ces informations,
- de simplifier le développement des plugins pour les développeurs,
- de simplifier la mise à jour d'un plugin pour un utilisateur,
- de nous débarrasser de la librairie `pyinotify` qui ne fonctionnait que sur Linux (et pas sur les systèmes BSD par exemple) et était utilisée pour vérifier la mise à jour de ces fichiers et leur rechargement.

4.2 Le langage

Le langage choisi initialement pour le projet est Python. À l'origine du projet, Python 2.6 était utilisé.

4.2.1 Le choix des composants majeurs

Dans les premières versions, le serveur REST du projet était basé sur les librairies `BaseHTTPServer` et Cie. Pour le passage à Python 3.x, il était nécessaire de modifier pas mal de code. Afin d'éviter de futurs désagréments (et comme nous devions refondre notre serveur REST), nous avons choisi d'utiliser le micro-framework Flask. La simplicité de mise en œuvre, que ça soit pour créer un vrai service REST ou une IHM, (nous l'avons aussi utilisé pour créer l'interface d'administration du projet) a été un vrai plus.

Nous avons aussi changé d'autres librairies importantes afin de rendre le code compatible Python 2.x et Python 3.x, telle `optparse` qui a été remplacée par `argparse`. À ce jour, seul notre hub xPL en Python reste à convertir pour être compatible Python 3.x (nous avions misé sur la librairie Twisted, mais malgré une longue attente et espérance, la roadmap pour migration en Python 3.x n'est à ce jour réalisée qu'à 55 %. Il va nous falloir choisir rapidement une autre librairie).

4.2.2 Optimisation des compétences humaines

Dans la cadre de la future version 0.4.1, nous allons créer un dépôt de plugins en ligne. Ce dépôt nous permettra de gérer les versions des plugins et sera utilisé par les solutions installées pour trouver et installer des plugins. Un tel dépôt existe déjà pour les versions antérieures et a été créé avec le framework PHP Yii. Ce dépôt marche bien, mais nous manquons de compétences sur le projet pour le maintenir et le faire évoluer. Dès que le développeur compétent est absent ou occupé, cet outil devient un point de blocage pour des évolutions.

Nous avons donc fait le choix de re-développer ce dépôt en Python avec Flask, le même framework que celui utilisé pour le projet. Nous ferons également de même pour toutes les éventuelles applications web que nous devrons développer.

Ceci va permettre à plus de personnes de pouvoir intervenir sur l'écosystème : chaque développeur ayant déjà touché au cœur de la solution aura un minimum de compétences sur le framework Flask et pourra donc participer à la maintenance, voire aux évolutions.

4.3 L'installateur

Les scripts d'installation sont quelque chose de très important : c'est le premier contact de l'utilisateur avec votre projet. Une installation qui se passe mal est souvent synonyme d'abandon de la part de l'utilisateur.

La souplesse des scripts d'installation est aussi très importante pour le packaging de votre projet sur les différentes distributions ! Voici quelques points que nous avons améliorés lorsque nous avons réécrit tous les scripts d'installation :

- Chaque élément de la configuration est désormais paramétrable via une option du script d'installation (éléments de configuration, création d'un compte utilisateur ou pas, lancement de l'installation de la base de données, ...).
- Les éléments paramétrables des fichiers de configuration sont gérés dynamiquement en fonction des fichiers exemples ; ainsi, il suffit de mettre à jour ces fichiers dans les sources pour que l'installateur propose automatiquement de nouvelles options.

Le fait d'avoir revu les scripts d'installation nous a permis notamment :

- D'avoir une intégration continue,
- Pour les développeurs, de pouvoir rejouer une installation complète en une seule commande.

4.4 Les tests

La qualité d'un projet passe entre autres par les tests. Parler de tests mériterait bien plusieurs pages tellement le sujet est vaste... Je n'aborderai pas ici les tests unitaires (TU) sur lesquels nous ne sommes malheureusement pas exemplaires, mais c'est un point sur lequel nous tâcherons de faire des efforts.

En termes de tests (excepté donc les TU), il reste donc :

- les tests fonctionnels : nous allons en parler ;
- les tests de performance : non encore implémentés ;
- les tests de charge : non justifiés ;
- les tests de robustesse : non encore implémentés ;
- les tests de vulnérabilité : planifiés.

La grande difficulté dans notre projet est que beaucoup de tests sont liés aux plugins et la majorité des plugins nécessitent du matériel. Pour réaliser ces tests, il faut donc que la personne ait le matériel, mais surtout, puisse faire les tests dessus : il n'est pas envisageable de faire certains tests si le matériel est utilisé en production par exemple. De plus, le matériel domotique coûte cher et il n'est pas envisageable pour les développeurs d'avoir le matériel en double.

4.4.1 Au début : les tests manuels

À l'origine du projet, nous avons rapidement installé la solution de test **Testlink**. Il s'agit d'une solution qui permet de rédiger des cahiers de tests, de les hiérarchiser et de gérer des campagnes de tests. Ça a été un gros plus pour la validation de chaque version du projet, mais rapidement plusieurs problèmes se sont dévoilés :

- réaliser les tests prend du temps ;
- peu de gens de la communauté ont participé aux tests et l'effort dépensé l'a surtout été par les développeurs (ce qui est contre-productif) ;
- peu de gens aiment/veulent rédiger des tests ;
- les tests étaient seulement passés en fin de version et il pouvait se passer plusieurs semaines sans que l'on voie un bug introduit pendant les développements.

4.4.2 Maintenant : les tests automatisés

Avec Domogik 0.4, nous avons créé plusieurs librairies pour automatiser et tester certaines tâches : création de devices, configuration d'un plugin, lancement et arrêt d'un plugin, vérification des données reçues (reçue, insérée en base)... En plus de ceci, nous avons créé des fonctions pour interagir avec le réseau xPL (protocole domotique sur lequel s'appuie notre projet) : attente d'un message suivant un pattern pendant N secondes, émission d'un message puis attente d'une réponse, ...

Tout ceci nous a permis d'automatiser la création de devices (et de la valider), la configuration d'un plugin (et la valider), son lancement et son bon fonctionnement (en agissant/écoutant sur le réseau xPL).

Malheureusement, seuls les plugins ne nécessitant pas de matériel pouvaient être testés de manière automatique quelle que soit la machine. Comme la majorité des interfaces domotiques sont accessibles via un port série, nous avons donc créé une librairie qui permet de simuler des périphériques série en chargeant des scénarios écrits en JSON. Ces scénarios peuvent envoyer des données *ad vitam eternam* et envoyer des réponses spécifiques en fonction des messages écrits sur le port série.

Nous sommes par exemple capables de tester en moins de 30 minutes et de manière automatique un plugin basé sur une technologie série avec plus de 50 matériels différents.

4.4.3 Les tests automatisés « from scratch » et à chaque commit !

Les tests automatisés c'est bien, mais encore faut-il idéalement les lancer sur une machine vierge :

- On se retrouve dans le cas d'un utilisateur lambda qui n'a jamais rien installé ;
- Ça permet de tester les problèmes éventuellement liés à une anomalie dans l'installation (dépendances non satisfaites, ...).

Par contre, pour réaliser ceci, il faut :

- Pouvoir recréer une machine virtuelle (VM) vierge à chaque commit (gourmand en ressources) ;
- Lancer l'installation et les tests de manière automatique ;
- Récupérer le statut des tests ;
- Remonter l'information.

Ici, nous nous sommes appuyés sur **Travis-CI**, un site qui permet de faire de l'intégration continue. Si vous avez un dépôt GitHub, l'usage est très simple : il faut autoriser l'accès à Travis-CI sur le dépôt, puis dans vos sources, ajouter un simple fichier de description. Ce fichier décrira les actions à réaliser (appel des scripts d'installation, des scripts de préparation des tests, des scripts de tests, des scripts de fin de test).

À chaque commit, le fichier sera interprété par Travis-CI et une nouvelle VM sera créée et les traitements définis seront lancés. À la fin, vous aurez un statut, consultable sur l'interface web de Travis-CI. En cas d'échec, vous recevrez un mail, mais vous pouvez aussi configurer des notifications IRC ou toute autre action !

4.4.4 Ce que nous a apporté concrètement Travis-CI

La mise en place de l'intégration continue nous a apporté plusieurs avantages :

- La validation de chaque commit pour les dépôts du projet et pour chaque dépôt des plugins ;
- La détection des anomalies liées à des problèmes d'installation ;
- L'obligation de rendre Domogik compatible avec les virtualenv Python ;
- La détection rapide (et l'identification du commit) de plusieurs régressions ;
- Le début de l'automatisation des tests !

Il nous reste par contre encore pas mal de travail : tout le code du projet n'est pas encore couvert par les tests, mais maintenant que l'infrastructure est en place, il s'agit d'un travail de fond qui va continuer tout doucement.

4.4.5 Les inconvénients de Travis-CI

Le principal inconvénient de cet outil est que nous n'avons pas la main dessus. Si le service s'arrête ou devient payant, nous n'aurons plus d'intégration continue et il nous faudra en monter une nouvelle.

Cet inconvénient est finalement assez faible : l'outil se contentant (pour faire court) de créer des VM dans certaines configurations (la partie la plus complexe), puis d'enchaîner des commandes, il ne devrait pas être trop complexe d'adapter notre intégration continue sur une autre solution.

Mais surtout, nous n'avons pas à ce jour à nous préoccuper d'une infrastructure lourde en termes de machines virtuelles (et ça vaut bien cet inconvénient).

4.5 Prise en compte des architectures « lentes »

Quand le projet a débuté, les plateformes de type Raspberry n'étaient pas encore démocratisées et recherchées par la communauté domotique. Leur arrivée et surtout, leur utilisation par certains membres de la communauté, a permis de mettre en évidence des problèmes de performances divers. À ce jour, par exemple, l'utilisation de Domogik 0.3 sur un Raspberry est loin d'être une partie de plaisir : l'interface est lente et certaines fonctionnalités de l'administration buggent. Voici quelques exemples des soucis que nous rencontrons :

- Les performances de la base de données (nous en avons déjà parlé).
- Les performances de l'interface Domoweb qui utilise Django (certes, il restait des optimisations à réaliser, mais autant c'est fluide sur un

serveur classique, autant sur un Raspberry c'est leeeeeeent).

- Les performances d'outils externes à Domogik. Par exemple, la commande `pip freeze`, qui permet de lister les packages Python installés, peut prendre plus de dix secondes sur certaines configurations : ceci a pour effet de faire planter la gestion des plugins dans l'interface (un timeout de 10 secondes était configuré en cas d'erreur).

Dans le cadre du développement des versions 0.4.0 et plus, nous avons donc choisi des frameworks plus légers (Flask au lieu de Django) et la gestion des plugins/packages va être réalisée différemment, afin de prendre en compte les lenteurs liées aux outils externes.

Nous avons aussi prévu de réaliser, d'ici une ou deux versions, des tests de performance sur Raspberry afin de pouvoir comparer au fur et à mesure des commits les performances et pouvoir identifier au mieux les sources de lenteur. Ceci se traduira sûrement par une réinstallation *from scratch* chaque nuit de la dernière version de développement, puis par le lancement de tests simulant une activité utilisateur et la mesure du temps de traitement des différentes pages. Notez que ces tests ne pourront pas être faits avec Travis-CI ou tout autre service distribué d'intégration continue : des tests de performance impliquent des serveurs stables d'un point de vue architecture et charge (sinon les résultats peuvent varier) ! Nous utiliserons donc probablement une plateforme Raspberry hébergée chez un des développeurs afin de réaliser ces tests.

4.6 Mesure des performances

Un travail a été initié sur le composant REST, afin de pouvoir logger le temps d'exécution de chaque appel à une URL. Ceci permettra de faire des tests de performance et de voir l'évolution des performances dans le temps. Ce travail sera probablement à généraliser sur les autres composants du core.

4.7 La sécurité

Il existe des outils qui permettent de faire du *fuzzing* sur des services web. Nous avons prévu de mettre en place ce type d'outils et de les utiliser sur chaque version alpha/bêta/candidate/finale des versions à venir. Ceci permettra de mettre en évidence (et surtout de corriger) d'éventuelles failles.

5 Le support aux utilisateurs

5.1 Il y a utilisateurs lambda et utilisateurs bêta

Loin de moi l'idée de dénigrer des utilisateurs, mais il faut bien reconnaître et prendre en compte qu'il y a d'un côté :

- L'utilisateur lambda, qui recherche un minimum d'informations avant de poser une question et a 2/3 bases sur le monde de GNU/Linux. L'utilisateur lambda a une vie à côté de notre projet et s'attend à être guidé (au minimum par sa machine...).
- L'utilisateur bêta qui a un comportement bêta ! D'un côté il ne va pas chercher, de l'autre il ne va pas écouter ce qu'on lui dit, ou ne comprendra rien malgré une patience digne d'éloges de la part de la personne qui sera à son écoute. Ils sont rares, mais ils existent et nous avons déjà perdu plusieurs heures de travail à dépanner de telles personnes pour des raisons souvent très drôles (avec beaucoup de recul).

Dans les versions 0.1 à 0.3, nous avons consacré de nombreuses heures à réaliser du support aux utilisateurs. Ce temps est malheureusement précieux et a été pris soit sur nos loisirs, soit sur le temps de développement... En 0.4, nous avons pris en compte un certain nombre de problématiques récurrentes directement au niveau du code. Notez que chaque composant du projet hérite d'une même classe et que c'est cette classe qui va gérer les différentes problématiques.

Dans tous les cas, il faut garder en tête que malheureusement :

- L'utilisateur ne va pas souvent aller lire les logs et se contente des informations présentes dans la sortie standard ou l'interface ;
- L'utilisateur n'a très souvent rien fait de mal (hummm) et s'il n'a pas suivi la documentation, c'est qu'elle est mal faite (ce qui peut être vrai d'ailleurs) ;
- L'utilisateur vous indiquera dans la majorité des cas le problème de manière très précise : « Ça ne marche pas », alors qu'il aurait pu vous noyer de détails : « J'ai lancé le composant X, je n'ai pas accès au service à l'URL http://192.168.1.1:40404 et dans les logs, je trouve cette erreur : ... ». Ces détails ne servent à rien évidemment ! Si ça « ne marche pas », vous savez forcément d'où ça vient ;)

5.2 Le niveau des logs

Avant de parler du niveau des logs, nous avons réalisé une première modification importante dans notre gestion des logs : alors qu'auparavant dans la sortie standard nous retrouvions uniquement les résultats des commandes `print()` et dans les logs les résultats des commandes `self.log.info`/`debug`/`warning`/`error()`, maintenant nous retrouvons le même contenu dans les logs et la sortie standard. Les commandes `print()` sont désormais proscrites du code.

Certains logs étaient mal catégorisés et il y avait par exemple trop de choses en niveau `ERROR` et certaines informations importantes pour le support étaient en niveau `DEBUG` (valeur des paramètres de configuration des plugins par exemple). Voici ce qui a été fait et ce que je vous conseille :

- **DEBUG** : tout log jugé utile par le développeur, ce niveau ne sera pas activé dans les packages finaux.
- **INFO** : tout log important à la compréhension du statut d'un composant : les phases importantes du démarrage d'un composant, la valeur des éléments configurables par l'utilisateur, ...
- **WARNING** : tout événement qui ne devrait pas se passer, mais qui n'est pas bloquant pour le fonctionnement (par exemple, une trame reçue par un compteur électrique et que l'on n'arrive pas à décoder car elle est corrompue).
- **ERROR** : tout point entraînant dans la majorité des cas un dysfonctionnement total ou partiel pour l'utilisateur.

5.3 Relancer deux fois un composant

Pour un utilisateur lambda, si un composant (plugin ou composant core), ne réagit pas de suite, c'est probablement qu'il ne fonctionne pas, il faut donc le relancer une seconde fois, voire plus (mais sans vérifier via un petit **ps** qu'il ne tourne pas déjà). Ce type de comportement débouche forcément sur une impasse dans le cas d'un serveur réseau (le port étant déjà ouvert par exemple) et peut déboucher sur des fonctionnements étranges.

Ici, nous avons simplement vérifié au démarrage des composants que le composant n'est pas déjà démarré. Et si c'est le cas, on arrête le composant après avoir gentiment affiché le pid des process déjà actifs : l'utilisateur n'a plus qu'à arrêter le composant (et s'il est motivé, à regarder dans les logs qu'est-ce qui s'est mal passé).

5.4 Vérification de la configuration

Les plugins de notre projet peuvent nécessiter une configuration. Ces éléments peuvent être obligatoires ou non et sont typés. Ils sont décrits dans un fichier au format JSON.

Nous avons créé une fonction, appelée par le développeur au lancement du plugin et qui va (en fonction du paramètre) récupérer de manière automatique les éléments de configuration, les caster en fonction de leur type et surtout, vérifier si les éléments obligatoires sont configurés ou non. S'il y a un oubli ou une erreur de la part de l'utilisateur, le plugin se coupe automatiquement avec un message approprié. Tout sera également loggé : valeurs récupérées ou valeurs par défaut utilisées (pour les paramètres optionnels), afin que le développeur ait toutes les informations importantes lorsque l'utilisateur lui fournira les logs.

6 La maintenance de l'écosystème

Un projet, ce n'est pas seulement du code et une communauté, c'est aussi plein d'autres tâches qui ne font pas avancer le projet lui-même, mais qui sont nécessaires :

- La maintenance des dépôts de code,
- La création et la maintenance d'un site,
- Faire vivre le site,
- Gérer un forum, une *mailing list*, un chan IRC, ...,
- Rédiger et publier la documentation (et un jour la traduire),
- Faire des sauvegardes,
- Maintenir les machines.

Tout ça prend de plus en plus de temps et d'importance quand le projet avance !

6.1 Les dépôts de code

À l'origine, le projet utilisait Mercurial et le dépôt était hébergé sur un de nos serveurs en ligne. Ce dépôt nécessitait peu de maintenance, mais il fallait régulièrement donner des droits à de nouveaux utilisateurs sur le dépôt.

Afin que des utilisateurs anonymes contribuent, il leur fallait créer un patch et le proposer par mail, ticket ou forum. Ce n'était confortable pour personne.

6.1.1 GitHub : un choix à faire

Il y a quelques mois, un des membres du projet a proposé de migrer le code vers **GitHub**. Au début, peu de gens étaient convaincus :

- Migration nécessaire de Mercurial vers Git ;
- Apprentissage de Git pour les utilisateurs ;
- Nous ne maîtrisions plus l'hébergement du code.

La personne ayant été convaincante, nous avons migré et nous ne l'avons pas regretté !

6.1.2 Plus de gestion de comptes compliquée

Notez tout d'abord que là où nous utilisions un dépôt commun au core et aux plugins, depuis la 0.4, le core a son dépôt et chaque plugin a également son propre dépôt.

Là où auparavant nous devions gérer les comptes des utilisateurs sur le dépôt par profil (développeur core, développeur IHM, développeur de plugin) afin de limiter la casse en cas de mauvaise manipulation, maintenant une équipe « core » a la main sur le dépôt du core et ensuite, chaque plugin peut avoir ses propres contributeurs.

N'importe qui peut également créer un fork du dépôt, le modifier, puis proposer une *pull request* qui pourra être vérifiée via le site web de GitHub, puis validée et incluse. Il n'est donc plus besoin de faire une demande à notre équipe pour proposer un patch, tout est géré directement via GitHub.

6.1.3 Une charge en moins sur notre infrastructure

La dépôt n'étant plus hébergé sur nos machines, il s'agit d'une charge en moins à gérer (que ça soit en puissance ou en maintenance).

6.1.4 Et le risque de perdre les données ?

Si jamais GitHub fermait du jour au lendemain ou désactivait notre compte, nous ne perdrions pas les données : Git étant décentralisé, chaque utilisateur a une copie du dépôt. Et nous pourrions toujours installer un GitLab, clone de GitHub à installer sur nos serveurs.

À noter que nous n'utilisons pas le tracker de GitHub pour le core, donc nous avons toujours la main sur nos tickets et notre roadmap (en effet, ce tracker est trop limité pour la gestion des tickets du core).

6.2 Le plan de reprise d'activité

Dans les sociétés, on appelle PRA (plan de reprise d'activité) un processus qui permet de mettre instantanément (ou tout au moins rapidement) une 2ème infrastructure en ligne. Une tel plan est à envisager : nul n'est à l'abri de voir ses serveurs crasher et même malgré la présence de sauvegardes, il peut être long de réinstaller des serveurs (d'autant plus si vous avez un écosystème important hébergé dessus).

Un membre de l'équipe est en train de travailler à la création de nouvelles VM configurées grâce à **Puppet**. Ceci permettra de recréer l'écosystème de manière automatisée, par exemple : installation et configuration de WordPress, installation de ses plugins, chargement de la sauvegarde de la base, ...

Ce sont des actions coûteuses en temps, ingrates, et la communauté ne les voit même pas, mais le jour où vous en aurez besoin, vous pleurerez de joie !

7 La communauté

7.1 À chacun son dépôt

Afin de simplifier le processus de création d'un plugin, chaque plugin/package aura désormais son propre dépôt GitHub (voire un autre hébergeur, pourquoi pas !). Ceci permet à un contributeur de créer un dépôt en son propre nom et de commencer le développement de son plugin sans avoir besoin de demander quoi que ce soit à notre équipe (auparavant, il fallait un accès en écriture aux sources du dépôt principal).

Par contre, afin que la documentation du plugin puisse être générée sur notre site et que le plugin soit référencé, l'utilisateur devra enregistrer son plugin sur notre gestionnaire de plugins.

Ceci permet à chacun de gérer ses propres plugins de manière autonome, voire de créer des plugins personnels et de ne pas les diffuser à la communauté.

En plus du dépôt, nous conseillons aux développeurs des plugins d'utiliser le tracker fourni avec le dépôt de leur plugin, afin de gérer les tickets liés au plugin. Ceci permet d'alléger le tracker du core en nombre de tickets, mais surtout de retrouver facilement les tickets liés à un plugin.

7.2 Une bonne gestion des branches

La gestion des branches sur le gestionnaire de versions est un point très important. Jusque-là nous n'avions pas encore trouvé le modèle idéal. Maintenant, c'est chose faite. Il n'est toutefois pas appliqué : il le sera seulement à partir de la sortie de la version 0.4.0.

Je vous invite à lire la description complète de ce modèle **[git-branching-model]**. Mais je vais vous donner les quelques points qui nous ont séduits dedans :

- La branche *master* correspond toujours à une version finie et stable du projet. Ainsi, n'importe quel utilisateur qui clone le dépôt pourra utiliser sans réfléchir une version stable de suite.
- La branche *develop* contient toutes les fonctionnalités de la prochaine version.
- Chaque évolution ou dossier sera, avant de passer dans la branche *develop*, développé dans une branche dédiée, appelée branche de fonctionnalité (*feature branch*). Ainsi, chaque dossier est développé indépendamment et n'impacte pas les autres dossiers : si un dossier prend du retard, il sera toujours possible de sortir une version avec les autres dossiers finis.
- Des branches dédiées au support (pour les patchs, corrections de documentation) peuvent être créées pour chaque version ; elles doivent être utilisées uniquement pour du support.

7.3 Une documentation claire

Nous avions déjà une documentation assez conséquente pour les précédentes versions de Domogik, mais il s'est avéré que ça ne suffisait pas pour tout le monde. Voici les reproches que nous avions :

- L'information recherchée n'était pas évidente à trouver pour qui ne connaissait pas déjà bien la solution ;
- La limite entre documentation utilisateur, documentation technique et documentation pour les développeurs de plugins était parfois floue ;
- Pour le développement des plugins il était plus facile de s'inspirer d'un plugin existant que de partir de la doc (ceci dit, quel que soit le sujet, il est souvent plus facile

de partir d'un exemple que de commencer de zéro avec une documentation, quelle que soit la qualité de la documentation) ;

- L'API pour les développeurs d'interface web n'était pas claire et manquait d'exemples.

Pour répondre à ces différentes demandes (et comme nous avons réécrit une grande partie du projet et revu le modèle des plugins), nous avons repris toute la documentation existante de zéro !

7.3.1 Le choix du format de la documentation

Choisir le bon support/format pour votre documentation est très important ! Il y a notamment un point qu'il faudra prendre en compte : est-ce que votre documentation sera traduite dans d'autres langues un jour ? La gestion des traductions est en effet un point très important et qui demande un investissement initial.

Le second point important est le *versioning* de la documentation : est-ce que votre documentation n'existera que pour la dernière version du projet, ou souhaitez-vous avoir une version de documentation par version du projet ?

À l'origine du projet, nous avions choisi d'utiliser un wiki (**Tikiwiki** pour le nommer), qui permet de gérer les traductions de pages. Outre une certaine lourdeur du wiki, il s'est avéré compliqué de gérer l'avancement des traductions, mais surtout, de gérer la documentation pour différentes versions du projet. À ce jour, nous sommes en train d'abandonner ce wiki, mais il s'agit d'une action délicate, car la documentation de nombreux plugins est encore hébergée dessus... Bref, choisissez bien si vous ne désirez pas vous traîner un boulet aux pieds pendant plusieurs mois (et compliquer la vie des utilisateurs qui cherchent la documentation).

À ce jour, nous gérons toute la documentation via **Sphinx**. Cet outil permet d'écrire de la documentation au format reST (*reStructuredText*) dans des fichiers texte (.txt) et lisibles humainement. Ensuite, il est possible de générer la documentation dans différents formats (PDF, HTML, ...), mais aussi d'appliquer de nouveaux templates. La documentation est gérée dans un dossier spécifique des sources du projet et il existe donc une version de la documentation pour chaque branche (et donc version) du projet. La documentation bénéficie donc de tous les avantages de Git (versioning, merge, ...) et nous sommes capables de générer la documentation pour chaque version du projet.

Sphinx peut s'interfacer facilement avec les méthodes classiques de traduction et il est donc possible de s'appuyer sur des services en ligne comme **Transiflex** pour aider les utilisateurs à contribuer aux traductions.

Pour information, la documentation en ligne de Python est gérée par Sphinx.

Note

Même si Tikiwiki n'a finalement pas été l'outil de nos rêves, il peut très bien correspondre aux besoins d'autres personnes ! Il ne s'agit pas ici d'une critique de ce wiki de manière générale, mais tout simplement, il ne correspondait pas à nos besoins.

7.3.2 Un peu de scripting pour automatiser la diffusion de la documentation

Écrire de la documentation c'est bien, mais il reste encore à la diffuser ! L'utilisation d'un wiki permet d'avoir les corrections diffusées instantanément à l'utilisateur, mais dans le cadre d'une documentation gérée dans les sources, c'est plus compliqué : il faut rafraîchir le dépôt, passer sur la branche à diffuser, générer la documentation au format voulu et la mettre au bon endroit.

Toutefois, ici, le script shell est votre ami ! Nous avons, pour notre part, sur notre serveur un script (qui prend en paramètre un nom de branche) qui va rafraîchir le contenu d'une branche, générer la documentation en lui appliquant un template et la mettre en ligne. Ce script est appelé pour chaque branche

Note

Attention, reprendre une documentation de zéro n'est pas une action anodine ! À titre personnel, j'estime à plusieurs dizaines d'heures le temps passé sur la refonte de la documentation pour la version 0.4.0 du projet. Écrire de la documentation a un coût, est épuisant et même parfois, démotivant. Voici quelques conseils :

- Avant d'attaquer une partie de la documentation, mettez-vous dans la peau de la personne qui la lira et demandez-vous ce qu'elle cherchera à savoir en premier et quelles sont les informations importantes que vous ne voulez pas qu'elle loupe. Vous pouvez demander par exemple à des futurs utilisateurs ce qu'ils en pensent et ce qu'ils désirent. Une fois ceci fait, posez un premier squelette de chapitres. Et régulièrement, remettez-vous dans la peau des utilisateurs afin d'avoir un œil critique sur votre travail. N'hésitez surtout pas à vous appuyer sur des utilisateurs pour leur demander leur avis de temps à autre !
- Lorsque vous écrivez une page, il vous vient l'idée de créer une seconde page pour détailler un point quelconque. Dans cette seconde page, il peut arriver aussi la même chose et vous vous retrouvez rapidement perdu dans les limbes de votre documentation. La manière la plus efficace de gérer ceci est, selon moi, de créer la nouvelle page vide et de faire le lien vers cette page ; ensuite, créez un ticket ou ajoutez dans une *todo list* qu'il faudra remplir cette page. Cela vous permet de rester concentré sur la page en cours.

de version chaque nuit et nous avons donc notre documentation publiée à jour chaque nuit de manière totalement automatique.

Il nous reste encore à créer les scripts qui s'occuperont de pousser sur Transiflex les dernières évolutions de la documentation et récupéreront les dernières contributions des utilisateurs, afin de générer de manière automatique toutes les documentations traduites.

Note

Dans la pratique, nos scripts de déploiement de la documentation sont un peu plus complexes : nous générons également un *header* commun à tous nos sites (forum, tracker, documentation) et il faut l'inclure dans la documentation. Nous générons aussi un index global créé dynamiquement et des pages statiques (erreur 404, ...). Tout ceci pourrait mériter un article dédié. Un jour, qui sait...

7.3.3 Et le contenu dans tout ça ?

Le contenu qu'il faut mettre dans la documentation est très variable d'un projet à l'autre. Voici quelques pistes :

- Bien séparer la documentation utilisateur des autres documentations (techniques, contributeurs).
- Pour les documentations techniques, essayez de réaliser des schémas d'architecture, de flux (nous avons encore du travail en ce sens à faire). Ceci permettra aux gens de mieux comprendre et donc, de s'impliquer plus et qui sait, peut-être de rejoindre le projet !
- Pour les documentations des contributeurs (développeurs d'IHM, de plugins), pensez à donner des exemples avec des explications et ne les noyez pas sous des avalanches de notions techniques. Il vaut mieux détailler des points spécifiques dans des pages dédiées, afin de se concentrer sur le sujet courant dans la page en cours.
- Préparez aussi des pages « guide » pour les contributeurs. Ces pages ne contiendront pas de technique, mais donneront les étapes à franchir pour réaliser une contribution (plugin ou IHM par exemple). Ceci permet au contributeur d'avoir une vue globale de ce qu'il va devoir faire et ensuite seulement il pourra piocher les informations techniques dans la documentation.

7.4 Une démo en ligne

Les développeurs d'IHM n'auront en général pas forcément la possibilité (ou l'envie) de monter une installation complète de votre projet afin d'étudier chaque cas de figure. Si vous pouvez leur mettre à disposition une démo en ligne avec des données virtuelles (mais qui peuvent être manipulées), cela leur permettra de jouer, faire des tests, des expériences et surtout, de participer en développant une IHM.

7.4.1 Facile ou pas ?

Monter une démo en ligne peut être facile ou compliqué. La démo d'une fonctionnalité JavaScript ou CSS nécessitera simplement la publication d'une page HTML. La démo d'un projet domotique est beaucoup moins simple :

- Il faut installer le projet,
- Il faut gérer la base de données,
- Il faut gérer du matériel (ou en simuler),
- Il faut fournir un historique de données pour les fonctionnalités d'historique (affichage des températures sur l'année par exemple),
- Il faut gérer la recréation périodique du projet (désinstaller, réinstaller, reconfigurer la base). Pendant cette phase, il faut aussi prévoir de mettre une page d'indisponibilité pour indiquer que la démo n'est pas cassée (ce qui donne une mauvaise image), mais en cours de reconstruction.

Tout ceci implique donc :

- de pouvoir automatiser une installation,
- de pouvoir importer ou générer un historique virtuel de données,
- de pouvoir simuler du matériel de manière transparente pour l'utilisateur.

7.4.2 Une démo... que pour les utilisateurs/contributeurs ?

Soyons un peu égoïstes... Monter une démo en ligne prend du temps. Y a-t-il moyen de trouver un usage pour le projet ? Oui !

Si on sait monter une démo en ligne pour les utilisateurs, on sait en monter une qui pourra servir à lancer des tests automatiques sur les IHM !

7.4.3 Vous l'avez fait ?

Non, nous n'avons pas encore créé la démo en ligne, mais c'est dans la roadmap. Techniquement, il ne nous manque qu'un plugin dédié qui simule des événements et réagit aux interactions de l'utilisateur.

Conclusion

Et voilà, je pense avoir fait un tour assez complet de ce qui a été amélioré sur notre projet. J'espère sincèrement que ce retour d'expérience pourra être bénéfique à d'autres personnes, notamment sur des projets en train d'éclore. Bien entendu, il faut garder à l'esprit que chaque projet est différent et a donc des contraintes différentes ! Il n'existe donc pas de solution universelle ;) ■

Référence

[git-branching-model] : http://nvie.com/posts/a-successful-git-branching-model/

JE PROGRAMME

LE GUIDE POUR APPRENDRE À PROGRAMMER EN 7 JOURS SEULEMENT ET SANS (TROP) D'EFFORTS !

C'EST DÉCIDÉ, AUJOURD'HUI JE M'Y METS !

GNU/LINUX MAGAZINE HORS-SÉRIE N°71

ACTUELLEMENT DISPONIBLE

CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

boutique.ed-diamond.com

LA DOCUMENTATION DE CODE NE SERT À RIEN !

par Dr KissCool [Attention au deuxième effet...]

Le programme est achevé et tout fonctionne correctement... Mission accomplie ! Est-ce vraiment certain ? Des informaticiens s'imposent ou se voient imposer une contrainte aberrante : documenter leur code. Mais pourquoi donc ?

Je vous propose dans cet article un argumentaire imparable pour expliquer à vos collègues pourquoi vous ne documentez pas votre code et pourquoi il s'agit d'une complète perte de temps.

1 Plusieurs types de commentaires

Les concepteurs de langages se sont montrés hésitants lors de l'intégration de la syntaxe permettant de documenter un code : pas moins de trois types de commentaires différents ! L'indécision est flagrante et pour comprendre une telle débauche de syntaxes, il faut faire un petit rappel historique.

La première syntaxe est apparue à la suite d'une soirée fortement arrosée, où quelqu'un s'est dit : « Wouah ! On va ajouter une syntaxe pour insérer des caractères qui ne servent à rien... Ça va être marrant ! ». Cet illustre personnage a méthodiquement effacé toute trace permettant de remonter jusqu'à lui lorsqu'il s'est aperçu du détournement sordide qui avait été fait de son invention géniale, qui pouvait même permettre d'obscurcir un programme pour éviter que l'on ne vienne trop farfouiller dans votre code. Le mécanisme mis en place s'appelle « commentaire en ligne » : tout caractère se situant après un caractère spécial dépendant du langage ne sera pas pris en compte. Voici un exemple illustrant le bon usage des commentaires en PHP :

```
<?php
  // AE 1F 00 BF 24
  print "Hello
  // 2C AA 1F 76 82
  world !"; // 00 FF 15 FF 97
```

Dans son usage détourné, qui est malheureusement de plus en plus en vogue à l'heure actuelle, nous aurions :

```
<?php
  // Affichage d'un message
  print "Hello world !";
```

Suite à cette infamie, les tenants du verbiage ont créé une deuxième syntaxe permettant d'ajouter des commentaires plus simplement : le « commentaire multiligne ». Grâce à cet apport, on peut maintenant décrire longuement ce que fait un code tout à fait compréhensible... Le début de la perte de temps :

```
/*
  Dans la ligne suivante
  je vais ajouter du code pour
  afficher un message. Ce message
  sera "Hello world !"
*/
print "Hello world !";
/*
  Normalement quand on arrive ici le
  message est apparu à l'écran
*/
```

Enfin, l'aberration ultime apparaît avec la troisième syntaxe : les blocs de commentaires multilignes expliquant le fonctionnement et quels paramètres sont attendus par une fonction. Là, on répète simplement le code que l'on vient d'écrire :

```
/**
 * Ce que fait la fonction
 *
 * @param  int  $n  Un entier
 * @param  string  $s  Une chaîne
 */
function maFonction($n, $p)
{
  ...
}
```

Bien sûr, les plus pervers n'hésiteront pas à utiliser et mêler les trois syntaxes pour insérer de véritables romans

à l'intérieur de leur code, alors qu'ils auraient pu réaliser un obscurcissement (ou obfuscation) très satisfaisant.

Voilà pour l'aspect historique. Ne cherchez pas d'où proviennent ces calamités, je vous l'ai dit : il est impossible de remonter jusqu'à leur initiateur (et non, Denis Bodor n'était pas encore né, j'avais également pensé à lui). Voyons maintenant pourquoi les gens commentent leur code.

2 Mémoire de poisson rouge

C'est connu, lorsqu'un poisson rouge fait le tour de son bocal, il a déjà oublié d'où il vient (et donc, qu'il a déjà fait un tour). Même si d'un point de vue purement scientifique la référence est loin d'être exacte, elle sert tout à fait mon propos. Certains informaticiens ont autant de mémoire qu'un poisson rouge et se voient donc obligés de commenter leur code pour gagner du temps lorsqu'ils reprennent un programme après s'être arrêtés quelques jours, voire quelques semaines. Les nuls ! Franchement, vous est-il déjà arrivé de revenir sur un code en vous disant, au choix :

1. Quoi ? Mais c'est moi qui ai écrit ça ? Mais c'est nul ! Il faut tout reprendre !
2. Alors là, je suis vraiment impressionné... Comment ai-je pu être aussi bon ? Je ne sais pas pourquoi j'ai choisi cette architecture, je ne comprends pas le code, mais c'est beau !

Bien sûr que cela ne vous est jamais arrivé ! Pourtant, pensez que certains malheureux luttent comme ils peuvent contre cette pathologie. Durant ma longue carrière, j'ai même vu des informaticiens particulièrement atteints, qui utilisaient dans leurs commentaires des termes étranges tels que « TODO » ou « FIXME ». On arrive là au paroxysme du trouble mémoriel et c'est pour cela que pour la plupart des individus il est totalement inutile de documenter le code.

3 Intelligence sociale

Sur de gros projets, on est souvent malheureusement contraints de travailler avec des gens (un peu comme moi qui suis obligé de travailler avec « vous-savez-qui »). Pour que chacun puisse apporter sa pierre à l'édifice, les gens se sentent obligés de comprendre ce qu'ont fait les autres développeurs... D'où les commentaires pour s'entraider. Mais on s'en fout ! Qu'ils se débrouillent ! S'ils ne sont pas capables de comprendre le code, qu'ils changent de boulot ! Moi, j'écris tous mes articles au stylo sur post-its non numérotés : ça m'est bien égal de savoir que Denis passe plus de temps à tout reconstitu... aïeuh ! Ça recommence avec les électrochocs... Je pense qu'il n'apprécie pas. Il va vraiment falloir que les auteurs trouvent une solution à ce problème ! Bref, pour revenir au sujet qui nous occupe, moi, lorsque je suis obligé de travailler avec d'autres personnes, je ne fais aucun effort pour expliquer mon travail. Avec un peu de chance, ça les ralentit énormément ou même mieux, ils ne comprennent rien, ils se font virer et moi je reste seul sur le projet... Mouahahah ! Pardon, je m'emporte et je risque encore une sanction de D. Le Sadique.

Il nous reste à voir un dernier point pour clore cet article. Que les plus sensibles d'entre vous s'arrêtent là, car ce qu'ils risquent de découvrir est totalement intolérable : le summum de la perte de temps avec la documentation technique.

4 Documentation technique

L'utilisation des commentaires multilignes avec une syntaxe particulière issue le plus souvent de JavaDoc permet de générer des documents totalement inutiles, qui constituent ce que l'on appelle la « documentation technique ». Imaginez donc un document PDF ou HTML expliquant ce que fait un morceau de code... Mais à quoi voulez-vous que cela serve ? On ne va quand même pas l'imprimer pour la lire dans le RER ! Quand on pense en plus au temps qu'il faut passer pour écrire cette documentation, on se demande vraiment comment des gens peuvent s'infliger cette peine lorsqu'ils n'y sont pas contraints.

L'exemple le plus impressionnant de perte de temps vient comme toujours du côté des adorateurs du Grand Serpent : ils utilisent un système nommé Sphinx et, en plus de l'écriture des commentaires en suivant une syntaxe ésotérique, ils doivent passer des heures pour initier un projet de documentation à l'intérieur de leur projet de code. Je serai tenté de dire que c'est bien fait pour eux ! Choisissez donc un véritable langage au lieu de vous amuser à saisir des espaces ou des tabulations au kilomètre.

Conclusion

Nous avons vu différents niveaux de commentaires qui représentent tous un temps qui ne sera pas passé efficacement à écrire du code. Un peu comme si avant de commencer à programmer vous preniez un temps de réflexion devant une feuille blanche pour définir a minima l'architecture de votre code. Plus personne ne fait ça, pourquoi ne pas vous amuser alors à programmer en perforant des cartes ? Si l'objectif est d'être le moins efficace possible, cela serait encore plus radical que de perdre son temps en écriture de commentaires. La documentation de code ne sert à rien !

Mais n'oubliez pas :

« Il n'y a pas de mauvais langage... Il n'y a que de mauvais développeurs ». ■

COLLECTD ET PERFWATCHER : DÉCOUVERTE

par Yves Mettier [Auteur de C en action (2ième édition) paru chez ENI
Auteur du guide de survie Langage C paru chez Pearson]

Planté ! On va le rebooter. Mais que s'est-il passé sur ce serveur ? Encore une fois, trop de processus qui prennent trop de mémoire ? Lequel était-ce cette fois ? Et a-t-on assez de mémoire sur nos machines ? A-t-on assez de machines ?

1 Présentation de Collectd et PerfWatcher

Avant de se lancer dans une session de débogage avec Collectd et PerfWatcher, présentons ces deux logiciels.

Collectd est un démon qui collecte des informations de performance sur les serveurs et qui peut ensuite les stocker de diverses façons, dont celle qui nous intéressera dans cet article, en les écrivant dans des fichiers RRD. Collectd dispose également d'un mécanisme pour transférer ces informations via le réseau à une autre instance Collectd capable de les recevoir. Cela nous amène donc à imaginer de nombreux agents Collectd effectuant leur travail de collecte et envoyant les informations à un serveur centralisé, chargé de les stocker sur disque.

Lorsque les données sont écrites, dans notre cas dans des fichiers RRD, le travail de Collectd est fini jusqu'à la prochaine collecte, au prochain cycle (60 secondes par défaut, mais il est possible de tomber sans problème à 10, voire 1 seconde). Pourtant, vous voudrez pouvoir consulter ces fichiers RRD facilement. C'est le rôle de PerfWatcher, une application web, qui se charge de générer un graphe à partir du contenu de fichiers RRD. PerfWatcher n'est rien d'autre que l'interface utilisateur pour consulter cet historique que vous allez construire au fil du temps.

N'oublions pas de présenter Collectd-pw, sans lequel PerfWatcher ne peut pas déployer toute sa puissance. Collectd-pw est un ensemble de *patchs* pour Collectd, ayant plusieurs vocations :

- implémenter des fonctionnalités sur Collectd nécessaires à PerfWatcher pour une bonne communication entre les deux. Ces fonctionnalités étant plutôt spécifiques à une utilisation avec PerfWatcher, nous pouvons penser que ces patchs ne seront jamais intégrés à Collectd ;
- proposer de nouvelles fonctionnalités aux utilisateurs de PerfWatcher et Collectd-pw en avance de phase par rapport à Collectd. En effet, le modèle de développement de Collectd-pw permet d'intégrer des patchs développés par ailleurs, qui ne seront intégrés à Collectd que plus tard ;
- corriger des bugs de façon plus réactive que Collectd, dont le rythme de sortie des versions est relativement lent ;
- rétroporter des fonctionnalités disponibles dans les sources les plus récentes de Collectd sur son dépôt Git.

Les patchs de Collectd-pw sont disponibles sur GitHub, ainsi qu'une version dite packagée, ce qui peut donner l'impression d'un fork. Mais il n'en est rien. Les patchs de Collectd-pw sont disponibles pour être intégrés directement à Collectd et chaque nouvelle version de Collectd-pw est reconstruite à partir d'un Collectd d'origine. En cela, il ne s'agit donc pas d'un fork.

2 Voyons l'état et l'historique d'un serveur

Nous voici sur PerfWatcher, un PerfWatcher tel que vous allez l'installer à la lecture de l'article suivant. Nous voulons voir un serveur nommé « ixion » et observer sa charge, sa mémoire... S'il avait déjà été ajouté à l'arborescence de gauche, vous pourriez cliquer dessus. Il existe un champ de recherche en haut, si vous ne le retrouviez plus.

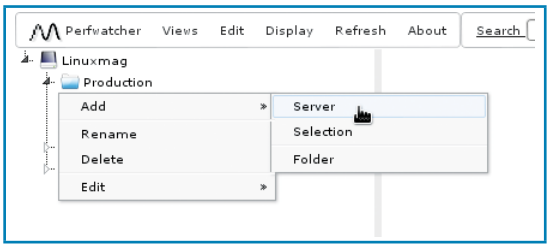


Fig. 1 : Ajout d'un serveur dans PerfWatcher

Mais il n'y est pas encore. Ajoutons-le : clic droit sur un répertoire, ***Add*** puis ***Server*** (Fig. 1). Indiquons son nom, validons, et le voilà ! Son icône à gauche, en gris, devrait passer en vert rapidement s'il est actuellement actif.

Cliquons maintenant sur le nom du serveur que nous venons d'ajouter. Apparaît alors à droite une barre avec des onglets sous laquelle vous trouvez un résumé succinct du serveur, charge et mémoire, ainsi qu'un graphe temps réel de la charge (Fig. 2). Les aiguilles bougent un peu, mais c'est l'effet du vent (ou plutôt un algorithme de simulation du vent). C'est rafraîchi seulement toutes les demi-minutes. Cliquons sur les onglets, par exemple ***load*** (Fig. 3) ou ***memory*** (Fig. 4)...

Fig. 2 : Onglet « Informations »

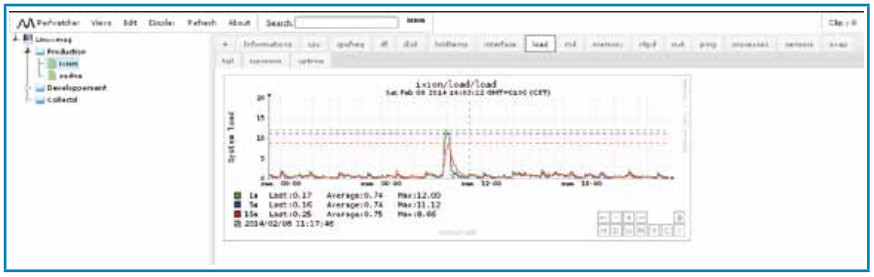


Fig. 3 : Onglet « load »

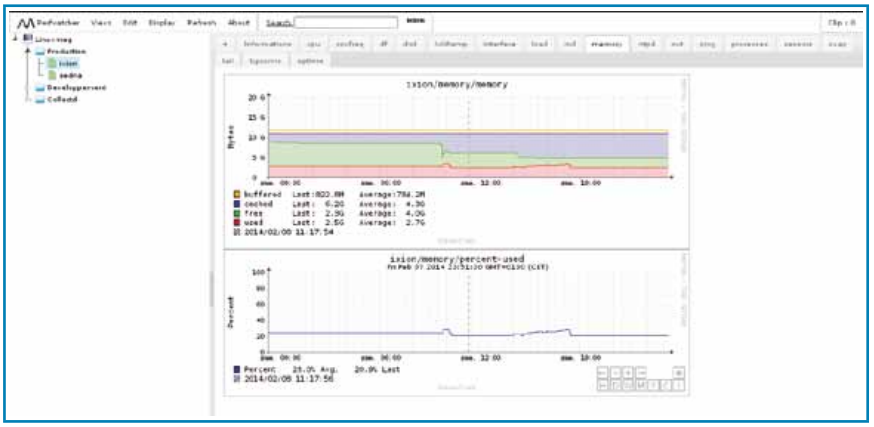


Fig. 4 : Onglet « memory »

Puisque le pointeur de la souris s'est déplacé sur un graphe, des boutons apparaissent en bas à droite (Fig. 5) :

- flèches : pour se déplacer dans le passé ou le futur ;
- ⊕ : pour ajouter le graphe courant dans le *clipboard* ;
- **+** et **-** : pour zoomer ou dé-zoomer ;
- ***H***, ***D***, ***W***, ***M***, ***Y*** : pour afficher la dernière heure (pratique pour voir les dernières données), la journée (vue par défaut), la semaine, le mois ou l'année ;
- ***C*** : vous choisissez vous-même la date ;
- ***!*** : le plus important quand vous avez plusieurs graphes sur la même page, surtout s'ils sont nombreux. Cela modifie tous les autres graphes pour avoir les mêmes dates de début et de fin que le graphe sous le pointeur de la souris. Il est important, car sans lui, impossible de mettre plusieurs graphes à la même échelle.

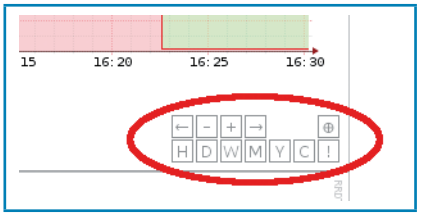


Fig. 5 : Boutons de zoom et de positionnement

Enfin, sachez qu'en cliquant sur le graphe, en restant appuyé sur le bouton de la souris, puis en se déplaçant d'un côté ou de l'autre, une ombre rectangulaire apparaît. Lorsque vous relâchez le bouton, le graphe se met alors à l'échelle de ce que contenait le rectangle. C'est ainsi que vous pouvez zoomer précisément sur la zone qui vous intéresse.

Note

Pour ceux qui connaissaient PerfWatcher avant la version 2.0, le zoom s'effectuait de façon différente. N'en soyez pas surpris.

3 Séance de débogage

Contrairement à ce que nous disions au début, le serveur n'a pas planté. Mais le serveur était un peu lent. En observant la courbe d'utilisation de la mémoire (Fig. 6), nous constatons qu'elle a été entièrement consommée. Le graphe de swap (Fig. 7) explique les lenteurs. Mais que s'est-il passé ?

Un clic droit, puis ***Top process*** ouvre une fenêtre avec la liste des processus qui tournaient au moment du problème (Fig. 8). Classons les entrées du tableau

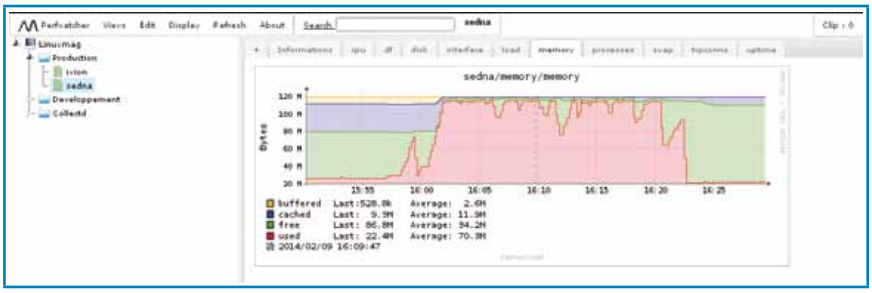


Fig. 6 : Courbe d'utilisation de la mémoire

Fig. 7 : Courbe d'utilisation du swap

par ordre décroissant d'utilisation de la mémoire, et nous obtenons le nom du coupable (vu le nom et son UID, vous vous doutez que ce programme a été conçu spécialement pour simuler une application ayant une fuite mémoire). Pour garder toute proportion, notez que notre serveur (une machine virtuelle) ne dispose que de 128 Mo de mémoire vive, ce que vous aviez déjà dû remarquer sur un graphe précédent (Fig. 6).

Un clic droit, puis ***Timeline*** nous donne le détail de ce qui s'est passé (Fig. 9) : ce processus a été lancé plusieurs fois. La timeline ne donne pas (pas encore ?) d'informations sur les processus pères et fils, mais nous pouvons supposer que notre processus coupable a fait des petits.

4 Capacity planning

Grâce à ses capacités d'historisation, PerfWatcher nous permet de faire du *capacity planning*. Cela n'est certes pas très évolué comparativement à certains produits du marché, mais avec un agrégateur, il vous est possible d'évaluer une ressource sur une partie de votre parc. Par exemple, vous pouvez agréger l'utilisation des CPU ou la charge (Fig. 10) et voir s'il atteint régulièrement le plafond ou si vous avez de la marge. De la même façon, une charge à zéro sur certaines heures signifie que vous avez des heures creuses et pouvez y lancer des traitements. Certains disposant des bons capteurs s'amuseront à calculer la puissance électrique consommée par leurs machines en fonction du temps.

Un agrégateur se configure sur la page des dossiers (Fig. 11). Vous pouvez y voir également d'autres informations et d'autres modes de remplissage, comme le remplissage dit manuel, en indiquant une liste (pratique pour copier-coller une liste de serveurs) ou par expressions régulières (regex si tous vos serveurs ont par exemple le même préfixe).

5 Le clipboard

Le clipboard ressemble à un caddie d'un site marchand, dans lequel vous allez placer des graphes. Pour cela, lorsque vous survolez un graphe et que les boutons contextuels apparaissent, cliquez sur ⊕. En haut à droite, le nombre de graphes dans le clipboard devrait s'incrémenter.

Le clipboard a plusieurs objectifs. Le premier est de fournir une sorte de comparateur. Lorsque vous cliquez sur le bouton en haut à droite, une fenêtre apparaît avec le contenu du clipboard sous forme d'une liste de graphes. Vous allez ainsi pouvoir comparer les graphes, les mettre en relation, et ainsi peut-être repérer l'indice qui vous manquait pour

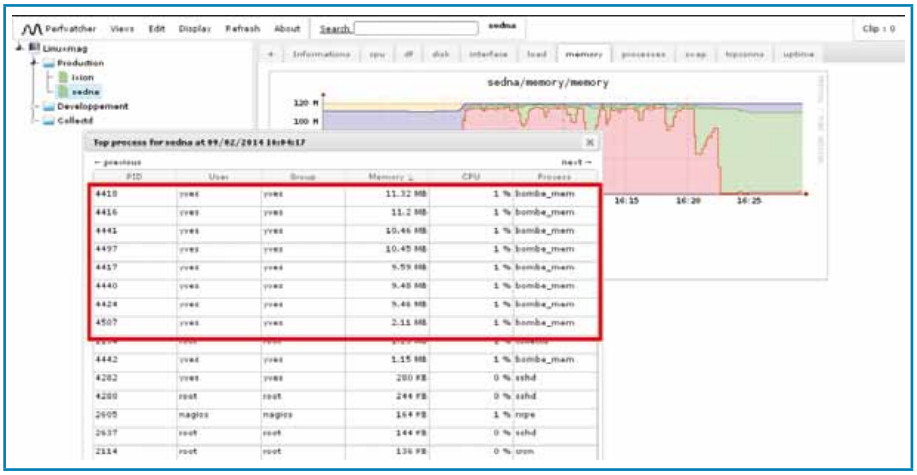


Fig. 8 : Liste des processus actifs lors du « problème »

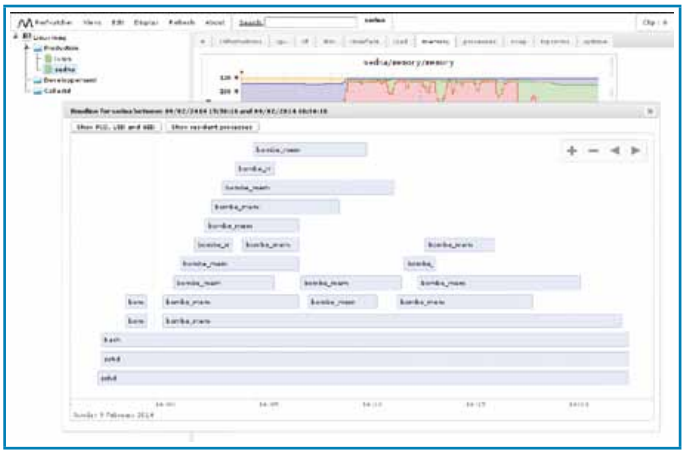


Fig. 9 : Timeline

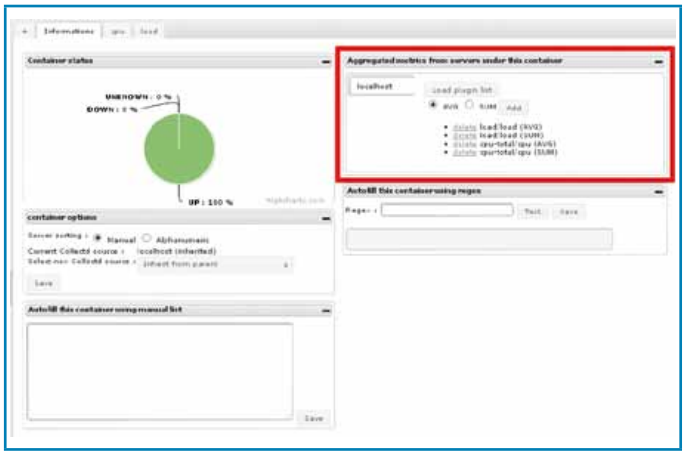


Fig. 11 : Vue d'un dossier

expliquer un dysfonctionnement sur un serveur. Sur la figure 12, nous voyons l'état de la mémoire et du swap sur la même page.

Lorsque vous visualisez le contenu du clipboard, vous pouvez déplacer (classer ?) les graphes entre eux pour les mettre dans l'ordre que vous souhaitez. Vous pouvez également basculer dans la vue *markdown*, dont le principal avantage est d'être moins gourmande en espace vertical. Dans la vue des graphes, vous pouvez supprimer un graphe. Comme vous travaillez en réalité sur une copie du clipboard, si vos changements (déplacements, suppressions) vous déplaisent, un bouton ***Cancel changes*** vous permet de les annuler et de revenir au contenu du clipboard initial. Lorsque vous fermez la fenêtre du clipboard, le contenu de la fenêtre (la copie) remplace le clipboard original. Enfin, un clic droit sur le bouton en haut à droite vous déroule un menu ; c'est là que vous pourrez vider d'un coup tout le clipboard.

Notez que le fait de recharger la page vide également le clipboard. Il est temporaire, comme cela est indiqué dans le cadre rouge en haut de la fenêtre affichant son contenu...

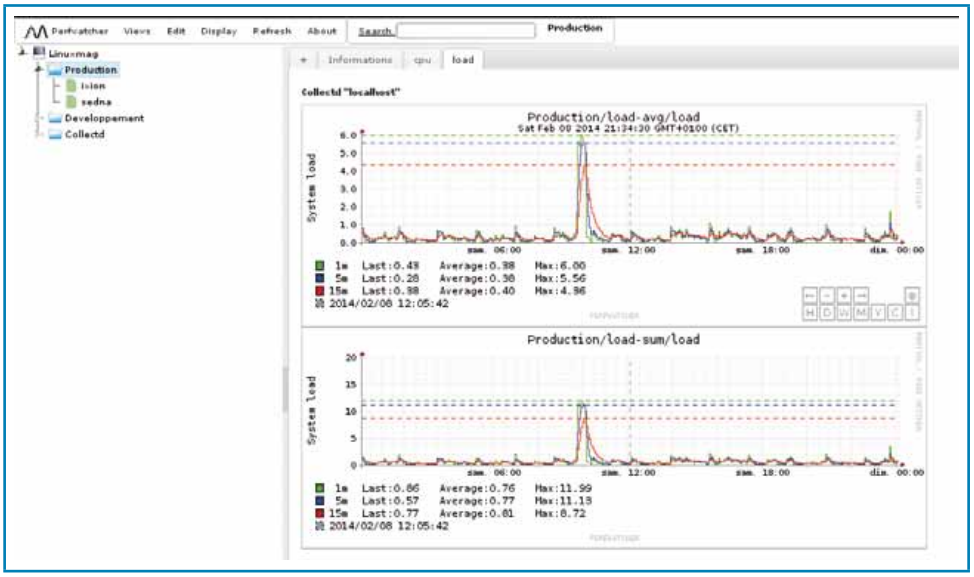


Fig. 10 : Agrégateur sur load : somme et moyenne

Un autre objectif du clipboard est de prendre facilement note de graphes qui vous intéressent, afin de les mettre dans un onglet. Comme nous allons le voir ci-dessous, lors de l'édition d'un onglet, vous pourrez facilement copier le contenu du clipboard.

6 Sélections et onglets personnalisables

6.1. Gestion des onglets

Lorsque vous affichez un serveur ou une sélection (nous verrons la notion de sélection plus loin), voire un « folder », vous pouvez créer des onglets. Cliquez sur le petit **+** et un petit formulaire vous demande le nom de l'onglet, ainsi que sa durée de vie (infinie, 7 jours ou 1 jour).

Il existe deux sortes d'onglets. Ceux correspondant aux plugins (ou agrégateurs dans une vue « folder ») sont statiques. Ils sont créés automatiquement et correspondent aux plugins de Collectd pour lesquels vous avez des fichiers RRD. Ces onglets ne peuvent pas être supprimés.

L'autre sorte d'onglets sont ceux que vous pouvez créer. Vous pouvez les déplacer pour changer l'ordre.

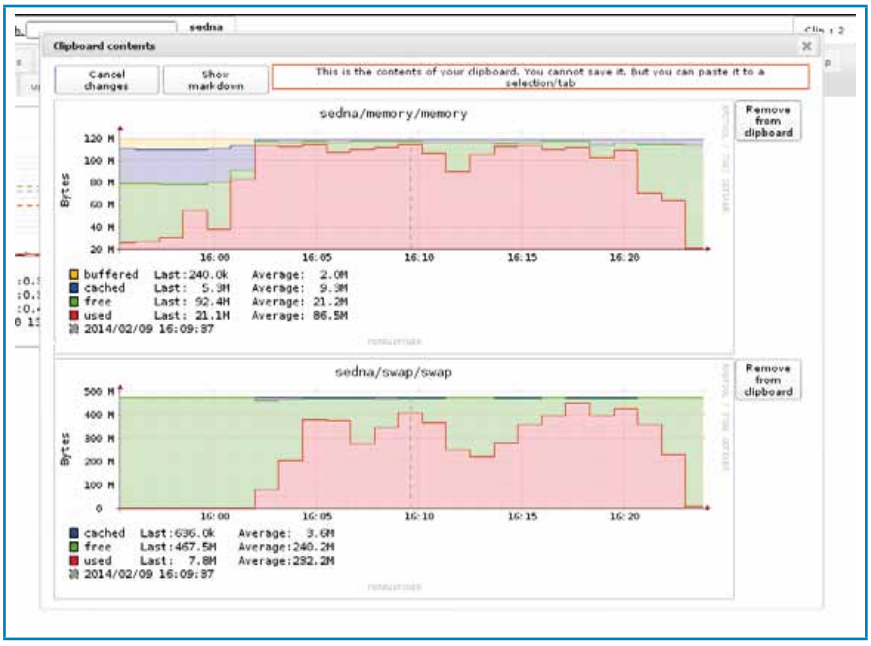


Fig. 12 : Le clipboard avec 2 graphes

En cliquant sur la petite croix sur le côté droit de l'onglet, vous le fermez après avoir validé une demande de confirmation.

Pour différencier les deux sortes d'onglets, vous notez la présence ou l'absence de la petite croix de fermeture.

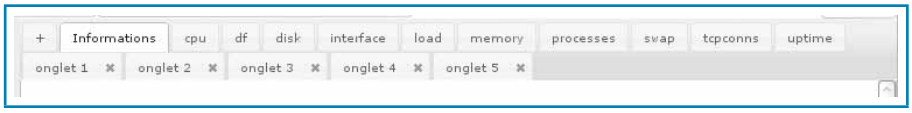


Fig. 13 : Onglets statiques et 5 onglets personnalisés

6.2. Personnalisation d'un onglet : syntaxe Markdown

Lorsque vous créez un nouvel onglet, seul un bouton ***Edit*** apparaît. Vous devez cliquer dessus pour éditer le contenu de votre onglet. Un formulaire avec un champ d'édition apparaît alors. C'est là que vous allez éditer le contenu de votre onglet.

La syntaxe du champ d'édition est une syntaxe dite Markdown. Cette syntaxe est expliquée, entre autres, sur Wikipédia (http://fr.wikipedia.org/wiki/Markdown). Aussi, nous ne rentrerons pas plus en avant dans le sujet. Si vous ne connaissez pas ce langage, sachez simplement que les habitués des wikis ne devraient pas être dépaysés.

PerfWatcher ajoute une extension à cette syntaxe pour pouvoir représenter ses graphes. La syntaxe est la suivante :

```
rrdgraph(<source>, <hostname>, <plugin>, <plugin_instance>, <type>, <type_instance>).
```

Les champs **<plugin_instance>** et **<type_instance>** peuvent être vides évidemment. Le champ **<source>** est « localhost » si vous n'avez pas configuré de source spécifique. Le reste coule de source (sans mauvais jeu de mots).

Vous n'êtes pas obligé de retenir cette syntaxe. En effet, vous pouvez sélectionner une liste de graphes dans le clipboard comme nous l'avons vu, puis au choix, soit cliquer sur le bouton ***Show markdown*** dans le clipboard et effectuer ensuite un copier-coller classique, soit cliquer sur ***Paste clipboard***. Un bouton ***Show clipboard*** vous permet de voir rapidement le contenu du clipboard avec cette syntaxe et vous aide également au copier-coller.

Un peu de syntaxe Markdown est proposée en bas du champ d'édition pour vous aider au début.

6.3. Sélections et pages de rapports

Une sélection est un bien grand mot pour désigner ce qui n'est pas un serveur et encore moins un conteneur. Il s'agit d'une entrée dans l'arborescence de gauche mais, lorsqu'on clique dessus, il n'y a rien. Cela pourrait ressembler à un serveur qui n'existe pas puisqu'il n'y a pas de fichiers RRD, donc pas de plugins, donc pas d'onglets existants. Mais une sélection n'est pas un serveur inexistant. Une sélection n'interroge pas Collectd pour savoir si un serveur du même nom existe. Une sélection ne renseigne pas sur une charge CPU et mémoire inexistante toutes les

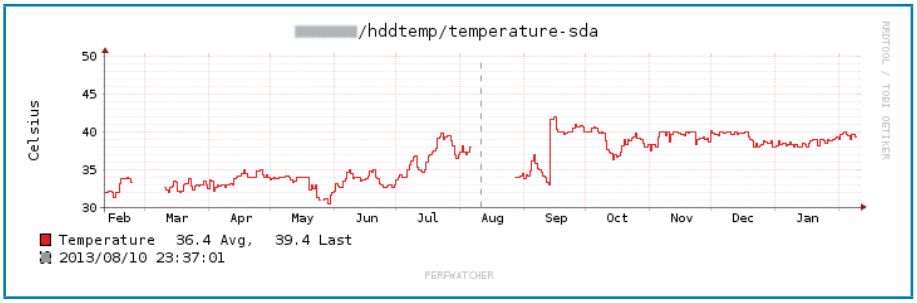


Fig. 14 : Courbe de température

minutes. Et surtout, une sélection a sa propre icône rouge et noire.

Une sélection peut donc être définie comme un ensemble d'onglets.

Les sélections peuvent donc servir à présenter des rapports d'état de serveurs ou, par l'intermédiaire des agrégateurs, de groupes de serveurs. Ou plus simplement rassembler plusieurs graphes issus de divers plugins ou de plusieurs serveurs ayant une relation quelconque.

Conclusion

Avez-vous remarqué ? Sur la capture d'écran indiquant le coupable de l'utilisation gourmande de la mémoire, en début d'article ? Pas le coupable, mais un peu plus bas... Là... Oui, *nrpe*. À quoi cela vous fait-il penser ? À Nagios ? Oui, gagné ! Nagios et Collectd cohabitent. En réalité, ils n'ont pas le même rôle. Nagios s'occupe d'alerter en cas de problème, alors que Collectd historise. Cela peut vous étonner si vous pensez aux journaux de Nagios qui permettent d'avoir la liste des alertes. Ou si vous pensez aux données de performance de Nagios. Cependant, Nagios n'est pas fait pour l'historisation, au même titre que Collectd n'est pas un remplaçant de Nagios (bien que pouvant envoyer des alertes). Vous pouvez donc faire cohabiter les deux, chacun ayant son rôle. Si vous pensiez encore que, au niveau de l'agent, cela fait deux fois plus de sondes, sachez que Collectd fournit un module Nagios. Ainsi, vous n'avez pas besoin de sonde Nagios. Seule celle de Collectd suffit. Dans notre exemple, elle n'était pas installée. Peut-être cela fera-t-il l'objet d'un prochain article ?

Note

Les captures d'écran ont été réalisées quelques jours avant la sortie de PerfWatcher-2.0 avec la version du moment disponible sur GitHub. Les modifications effectuées sur PerfWatcher entre ces captures et la sortie de la version 2.0 n'ont aucun impact sur celles-ci.

Et une petite dernière pour la route... Clic droit sur le graphe, ***Save image as ...***. Voici la courbe de température de notre disque dur sur un an, à défaut de celle de la carte mère (nous avons trop de trous dedans), qui donne un indice sur la tendance de la température extérieure. Comme quoi vous pouvez « grapher » tout ce que vous voulez, du moment que vous avez une sonde ! Et envoyer les images à vos amis ou vos clients (Fig. 14). ■

COLLECTD ET PERFWATCHER : INSTALLATION

par Yves Mettier [Auteur de C en action (2ième édition) paru chez ENI
Auteur du guide de survie Langage C paru chez Pearson]

yum/apt-get install collectd perfwatcher. Eh non, PerfWatcher nécessite des plugins spécifiques de Collectd. Cet article décrit une installation commentée du couple Collectd-PerfWatcher.

1 Collectd : l'agent

Nous allons maintenant installer Collectd. Commençons par le plus facile, un agent. Bien qu'il soit conseillé d'utiliser Collectd-pw sur l'agent, rien n'empêche de prendre un *package* de Collectd de votre distribution favorite. Vous y gagnez en simplicité puisqu'il n'y aura pas d'étape de recompilation, mais y perdrez sur certains modules, comme celui permettant d'afficher le *top process* ou le pourcentage mémoire utilisé. Aussi, lorsque nous aurons vu l'étape suivante, la compilation et l'installation de Collectd-pw pour le serveur, vous vous empresserez de faire de même pour l'agent. En attendant, nous pensons que les dernières versions 4 de Collectd (non testées), ainsi que les versions 5 (testées), sont compatibles avec le serveur Collectd-pw que nous configurerons plus loin.

Lorsque Collectd est installé, vous devez éditer le fichier de configuration. Il s'agit généralement de **/etc/collectd.conf** ou **/etc/collectd/collectd.conf**. Son contenu est long, mais facile à comprendre. La première section contient la configuration du démon (le dièse indique un commentaire) :

```
#Hostname "localhost"
#FQDNLookup true
#BaseDir "/var/lib/collectd"
#PluginDir "/usr/lib/collectd"
#TypesDB "/usr/share/collectd/types.db" "/etc/collectd/my_types.db"
#Interval 10
#Timeout 2
#ReadThreads 5
```

Toutes ces directives peuvent être commentées. Attardons-nous sur certaines d'entre elles. **Hostname** porte bien son nom mais par défaut, il s'agit du nom du serveur. Si vous déployez le même fichier de configuration sur chaque serveur de votre parc, il vaut mieux laisser Collectd trouver tout seul son nom. **Interval** indique la période de collecte. Normalement, une minute (**Interval 60**) devrait convenir. Notez cependant que ce choix est définitif pour chaque serveur. Nous y reviendrons dans le prochain article au sujet de la configuration avancée de Collectd. Enfin, **ReadThreads** indique non pas le nombre de modules (*plugins*) qui pourront être lancés, mais le nombre de threads qui exécuteront tous les modules. En effet, lorsqu'un module de collecte doit être exécuté, il sera lancé sur le prochain thread disponible. Si vous avez autant de threads que de modules, il sera lancé tout de suite. Si vous n'avez que 2 threads, la file d'attente sera longue avant l'exécution du module qui attendra son tour. Il s'agit donc de trouver le juste milieu entre la taille de la file d'attente et le nombre de threads que vous souhaitez permettre de lancer. Pour un petit nombre de modules, 5 est tout à fait correct.

Vient ensuite la section des plugins.

```
#LoadPlugin logfile
LoadPlugin syslog
<Plugin logfile>
    LogLevel "info"
    File STDOUT
    Timestamp true
    PrintSeverity false
</Plugin>
<Plugin syslog>
    LogLevel info
</Plugin>
[...]
```

Chaque module doit être chargé pour être pris en compte. Ainsi, dans l'extrait ci-dessus, nous avons deux modules de journalisation. L'un est activé (**syslog**) et l'autre non (**logfile**). Puis, vient la configuration de chaque module. La section de configuration de **logfile** est ignorée puisque le module n'est pas chargé. Celle de **syslog** sera par contre prise en compte.

Ce module est important, car en cas de problème de fonctionnement, vous aurez de nombreux indices dans les journaux. Pour des tests ou une petite entreprise, préférez **syslog**, dont la rotation et l'archivage sont déjà prévus par le système. Si vous optez pour **logfile**, n'oubliez pas de programmer quelques lignes pour **logrotate**, sinon c'est l'explosion assurée de votre système de fichiers !

Les autres modules se suivent, mais deux attirent notre attention. Nous les mettons ci-dessous à la suite l'un de l'autre.

```
LoadPlugin unixsock
<Plugin unixsock>
    SocketFile "/var/run/collectd-unixsock"
    SocketGroup "collectd"
    SocketPerms "0660"
    DeleteSocket false
</Plugin>
```

```
LoadPlugin network
<Plugin network>
#    # client setup:

    Server "192.168.2.14" "25826"

#    Server "ff18::efc0:4a42" "25826"
#    <Server "239.192.74.66" "25826">
#         SecurityLevel Encrypt
#         Username "user"
#         Password "secret"
#         Interface "eth0"
#     </Server>
    TimeToLive "128"

#    # server setup:
#    Listen "ff18::efc0:4a42" "25826"
#    <Listen "239.192.74.66" "25826">
#        SecurityLevel Sign
#        AuthFile "/etc/collectd/passwd"
#        Interface "eth0"
#    </Listen>
#    MaxPacketSize 1024

    MaxPacketSize 65535

#    proxy setup (client and server as above):
    Forward false

    # statistics about the network plugin itself
    ReportStats false

    #    "garbage collection"
    CacheFlush 1800
</Plugin>
```

Le module **unixsock** est très important au début, car c'est lui qui vous permettra de communiquer avec votre agent et vous assurer qu'il fonctionne comme vous voulez. Nous allons y revenir juste après la configuration réseau.

Nous configurons également le module **network** pour transmettre les données à un serveur central. Nous avons laissé de nombreuses lignes commentées pour vous montrer les nombreuses possibilités. Parmi celles qui sont actives, **Server** indique à quel serveur envoyer les données, **192.168.2.14** dans notre cas. Il est possible (voire conseillé) d'en indiquer deux (deux directives **Server** successives) pour redonder les données et pallier la perte d'un serveur central. Encore faut-il avoir les moyens de se le permettre. La directive **MaxPacketSize** peut être laissée à **1024** si vous ne faites rien d'extraordinaire et qu'il ne s'agit pas de Collectd-pw. Avec Collectd-pw par contre, passez ce paramètre à **65535**, car la liste des processus et de leurs informations ne tiendraient pas dans 1024 octets. Vous pouvez configurer votre Collectd comme un proxy avec **Forward**, mais ce n'est pas l'objet ici. Nous y reviendrons. Enfin, nous gardons **ReportStats** pour le serveur central. Il s'agit de statistiques internes au fonctionnement du module Collectd et celles d'un agent n'ont que peu d'intérêt.

Note

Nous mettons une adresse IP, car en cas d'échec intempestif de la résolution des noms, il ne faudrait pas que l'agent soit bloqué. Mais pour garder la souplesse d'un nom ou d'un alias configurable à souhait dans le DNS, nous vous conseillons de configurer une adresse IP virtuelle sur votre serveur. Si vous voulez changer de serveur, basculez l'adresse IP virtuelle sur un autre et l'agent n'y verra que du feu. Vous pouvez également préférer un alias DNS, car l'IP virtuelle vous oblige à avoir le serveur sur lequel basculer sur le même VLAN. Faites vos choix !

Pourquoi stocker les données sur un serveur central ? Collectd, serveur central comme simple agent, est un démon capable de stocker ses données dans des fichiers RRD (ou autres), comme nous l'avons vu en introduction. Si vous souhaitez que chaque agent stocke ses fichiers en local, c'est possible. Cependant, cela pose quelques problèmes à l'usage, comme comment faire pour savoir ce qui s'est passé si votre serveur s'arrête et ne redémarre plus ? Aussi, il est préférable de transmettre toutes les informations à un serveur central, serveur sur lequel vous pourrez concentrer les efforts de sécurisation des données.

Lancez ou relancez votre agent :

```
$ ps -ef | grep collectd
root 5482 1 0 17:15 ? 00:00:00 /usr/sbin/collectdmon
-P /var/run/collectdmon.pid -- -C /etc/collectd/
collectd.conf
root 5484 5482 1 17:15 ? 00:00:09 collectd -C /etc/
collectd/collectd.conf -f
```

Vous voyez un processus **collectdmon**, père d'un processus **collectd**. Le premier lance le second et se charge de le relancer s'il venait à disparaître. Par extension, une méthode pour relancer Collectd consiste à s'assurer que **collectdmon** tourne et à tuer son fils **collectd**. Il sera relancé automatiquement en prenant en compte une nouvelle configuration éventuelle. Si vous le faites trop souvent, le processus père le détectera et arrêtera de relancer son fils.

Note

Faites attention ! En cas de surcharge ou de dysfonctionnement sur votre serveur qui fonctionne à l'identique, vous pourriez par exemple perdre Collectd, le fils. Comme il serait automatiquement relancé, vous ne verriez pas forcément le problème, même s'il devenait cyclique. Aussi, vérifiez bien de temps en temps le fonctionnement du serveur.

Revenons à la socket Unix, configurée ici dans **/var/run/collectd-unixsock**. Avec un outil comme **socat**, vous pouvez écrire facilement dans la socket et y lire la réponse. Le plus simple est la directive **LISTVAL** qui indique la liste des noms des valeurs collectées par votre agent.

```
$ echo listval | sudo socat - /var/run/collectd-unixsock |
grep memory
1391810754.180 sedna/memory/memory-buffered
1391810754.180 sedna/memory/memory-cached
1391810754.180 sedna/memory/memory-free
1391810754.180 sedna/memory/memory-used
```

La liste étant longue, nous avons filtré sur les valeurs concernant la mémoire. Ces valeurs ont été collectées la dernière fois à 1391810754,180 secondes après l'an 1 de l'ère Unix (le 1er janvier 1970). De combien de mémoire disponible disposait-on sur ce serveur *sedna* ? Interrogeons la socket...

```
$ echo 'getval artemis/memory/memory-free' | sudo socat -
/var/run/collectd-unixsock
1 Value found
value=1.236173e+08
```

Nous avons donc encore 1,236173 x 108 octets libres. Pour ceux qui voudraient des Mo, faites votre division ! Voici pourquoi cette socket est très utile au début, elle indique que le démon

Collectd fonctionne bien. Mais par la suite, grâce aux fichiers RRD, vous saurez par un autre moyen que votre agent fonctionne (ou pas) et la socket perdra de son utilité. Elle pourra alors être désactivée. Et deux opérations pour obtenir des valeurs qu'il faudra encore convertir en unités plus courantes, cela en est trop pour nous. Vite, des graphes ! Installons le serveur central et PerfWatcher !

Juste avant de l'installer, notez encore, à la fin du fichier de configuration, la section des filtres. Il est en effet possible d'ignorer ou de récrire certaines valeurs avant leur stockage ou leur envoi par le réseau à l'aide de règles. Il se peut que vous en ayez l'usage pour normaliser par exemple certaines valeurs liées au matériel. Mais pour le moment, ignorez-les.

2 Collectd-pw : sur le serveur

2.1 Prérequis

Le serveur central ne peut pas se contenter d'un simple Collectd pour fonctionner avec PerfWatcher. Vous devez impérativement récupérer des patchs de Collectd-pw, ou mieux, utiliser une archive déjà patchée, ce que nous allons faire. Vous l'obtiendrez depuis le site de téléchargement de PerfWatcher et elle porte le nom **collectd-<version officielle>.<date du patch>.tar.gz**. Décompactez votre archive, allez dans le répertoire ainsi créé, puis lancez le bon vieux **configure**, **make**, **make install**. Stop ! Il manque des prérequis :

- **rrdtool** version ≥ 1.4 (pour utiliser **rrdcached**). Normalement, il est inclus dans votre distribution. Assurez-vous que vous avez également installé les fichiers de développement (souvent **-dev** ou **-devel** selon votre distribution) ;
- **json-c** version ≥ 0.10. Elle s'obtient sur https://github.com/json-c/json-c/wiki. Si vous la recompilez, voyez ci-dessous un détail d'installation ;
- **libmicrohttpd**. Les versions avant 0.9.22 n'ont pas été testées, donc prenez au moins cette version. Voyez ci-dessous un détail d'installation également.

La compilation et l'installation de **json-c** et **libmicrohttpd** sont simples avec **./configure; make; make install**. Pour json-c 0.10, il vous manquera un fichier. Installez-le à la main :

```
$ ./configure; make; make install
$ cp json_object_iterator.h /usr/include/json
```

Note

Ce bug est corrigé dans la version 0.11, mais nous n'avons pas encore eu l'occasion de la tester.

Pour **json-c** et **libmicrohttpd**, sur RHEL et SUSE 64 bits, il se peut que vous ayez également à copier des bibliothèques dans **/usr/lib64** si elles vous manquent :

```
$ cp /usr/lib/libjson.* /usr/lib64/
$ cp /usr/lib/libmicrohttpd.so* /usr/lib64/
```

2.2 Installation

Ouf, les prérequis sont installés. Lancez votre **configure; make; make install** :

```
$ ./configure --enable-top --enable-sysconfig \
        --enable-jsonrpc --enable-notify_file \
        --enable-basic_aggregator --enable-write_top \
        --enable-rrdcached \
        --prefix=/opt/collectd/<version>
$ make -j
$ make install
```

Cette ligne **configure** indique à elle seule pourquoi Collectd ne suffit pas. Mais vous voilà arrivé à l'installer !

Les modules **top** et **sysconfig** sont pour l'agent, alors que les autres, **jsonrpc**, **notify_file**, **basic_aggregator** et **write_top** sont pour le serveur. Les deux premiers, pour l'agent, ne nécessitent pas de configuration spécifique. Les autres, ceux du serveur, doivent être configurés avec attention.

2.3 Configuration avancée

Collectd-pw pouvant jouer à la fois le rôle d'agent et de serveur central, vous pourriez être amené à tout mettre dans le même démon. Nous vous le déconseillons fortement ! Vous pouvez utiliser le même binaire si vous le souhaitez, mais faites bien tourner un **collectdmon**/**collectd** pour le central avec son propre fichier de configuration (que nous appellerons ultérieurement **collectd-master.conf**) et un autre pour l'agent avec son fichier de configuration d'agent, *a priori* **/etc/collectd/collectd.conf**. En effet, en cas de dysfonctionnement, ce découpage permettra de savoir si c'est l'agent ou le serveur qui pose problème. De plus, l'agent et le serveur n'ont pas l'utilité de charger les mêmes modules. Nous allons donc partir du principe que vous savez configurer un agent et nous attaquer à la configuration du serveur. Vous noterez que nous stockerons les données dans les sous-répertoires de **/var/lib/collectd**. Libre à vous de choisir un autre répertoire. Assurez-vous seulement d'avoir assez d'espace disque.

2.4 Utilisateur, groupe et permissions

Autant l'agent, qui collecte des données de performance dont certaines ne sont accessibles qu'à *root*, doit tourner avec des privilèges élevés, autant le serveur Collectd-pw n'a besoin d'aucun privilège spécifique. Aussi, nous vous conseillons de créer un utilisateur **collectd**, membre du groupe **collectd** que vous créerez également. Vous ajouterez l'utilisateur **apache** ou **www-data** (en fonction de qui fait tourner votre serveur web) au groupe **collectd**.

2.5 Configuration de base

Dans un premier temps, copiez un fichier de configuration générique de Collectd, par exemple **/etc/collectd/collectd.conf** si vous ne l'avez pas modifié. La copie pourra s'appeler **/etc/collectd/collectd-master.conf**.

Éditez ce fichier. Le début contiendra ces directives non commentées :

```
Hostname "collectd-master"
#FQDNLookup true
BaseDir "/home/collectd"
PIDFile "/var/run/collectd-master.pid"
PluginDir "/opt/collectd/<version>/lib/collectd"
TypesDB "/opt/collectd/<version>/share/collectd/types.db"
```

Complétez votre collection d'anciens numéros !

boutique.ed-diamond.com

numerique.ed-diamond.com

```
TypesDB "/opt/collectd/<version>/share/collectd/types-perfwatcher.db"
TypesDB "/etc/collectd/custom_types.db"

#Interval 60

Timeout 2
ReadThreads 20
WriteThreads 10

WriteQueueLengthLimitHigh 100000
WriteQueueLengthLimitLow 90000

LoadPlugin syslog

<Plugin syslog>
    LogLevel info
</Plugin>

LoadPlugin network
<Plugin network>
    Listen "0.0.0.0" "25826"
    MaxPacketSize 65535
    ReportStats true
</Plugin>
```

Vous notez d'emblée que le nom du serveur est forcé à « collectd-master ». En effet, le nom du serveur étant pris par l'agent Collectd, vous devez en indiquer un autre. Ensuite, vous trouvez des fichiers de types supplémentaires. Ceux de **/etc/collectd/custom_types.db** sont les vôtres si vous en avez besoin, à vous de créer le fichier et de vous inspirer des autres pour le remplir. Évitez de toucher à ceux livrés à l'installation. Nous avons choisi d'exécuter **20** threads de lecture et **10** en écriture. Cela est probablement démesuré et nous les réduirions si nous souffrions de problèmes de performance sur le serveur.

Ensuite, viennent deux nouvelles directives, **WriteQueueLengthLimitHigh** et **WriteQueueLengthLimitLow**. Elles ont pour objectif de protéger votre serveur. Lorsque vous avez trop de données à écrire, cela crée un engorgement qui peut freiner sérieusement le serveur, voire l'arrêter. Cela a créé de sérieux problèmes par le passé, qui semblent maintenant corrigés. Néanmoins, il a été choisi de pouvoir protéger le nombre d'écritures et de perdre des paquets s'il y en avait trop, afin de ne pas tout perdre à vouloir tout garder. Le fonctionnement est que lorsque le nombre d'écritures est inférieur à la limite basse, on ne perd pas de paquets à écrire. À partir de la limite basse, on en perd selon une formule qui veut que plus on s'approche de la limite haute, plus on risque d'en perdre. Si la limite haute venait à être atteinte, on perdrait alors 100% des paquets, ce qui aurait pour effet immédiat de faire diminuer le nombre de paquets à écrire grâce aux écritures en cours. Cette fonctionnalité étant très récente, le paramétrage est à votre discrétion, mais prévoyez large pour ne pas perdre trop de paquets ! Ainsi, les **100000** indiqués ci-dessus peuvent sembler énormes (et non protecteurs) pour un parc de 5 serveurs, et être trop faibles pour une grosse entreprise.

Vous devez choisir un type de journal, **syslog** ou **logfile**. Nous avons vu ce point auparavant.

La configuration réseau est celle d'un serveur et non d'un agent. En effet, vous indiquez **Listen** et non **Server**. Ici, nous voulons aussi faire apparaître les statistiques du module réseau. Cela peut être utile pour connaître le fonctionnement de notre serveur au fil du temps.

2.6 Module rrdcached

Il s'agit probablement du module le plus important. Mais auparavant, vous devez lancer **rrdcached**, le démon du projet **rrdtool** qui met en cache les RRD avant de les écrire. Vous devez auparavant configurer ses options de lancement (par exemple dans **/etc/init.d/rrdcached** si ce fichier existe). Nous vous proposons les options suivantes :

- **-m 660** : permissions, pour que l'utilisateur et le groupe puisse y accéder, mais pas les autres. Attention : l'utilisateur **apache** devra pouvoir y accéder, donc mieux vaut créer un groupe dédié ;
- **-l unix:/var/run/rrdcached-collectd.sock** : nom de la socket Unix par laquelle les diverses applications (Collectd, Apache/PerfWatcher) vont communiquer avec **rrdcached** ;
- **-b /var/lib/collectd/rrd** : répertoire racine de vos fichiers RRD. N'oubliez pas de créer ce répertoire avec les droits suffisants pour l'utilisateur **collectd** ;
- **-w 300** : écrire les données en cache toutes les 5 minutes ;
- **-z 10** : permettre un délai aléatoire maximum de 10 secondes pour atténuer les pics de charge ;
- **-f 3600** : partir à la chasse aux données « mortes » toutes les heures ;
- **-p /var/run/rrdcached-collectd.pid** : fichier PID.

Vous veillerez à ce que l'utilisateur qui lance **rrdcached** soit l'utilisateur **collectd** et non **root** (car nous n'avons pas besoin que ce soit **root**). Nous vous conseillons d'avoir un script de démarrage dédié pour cela, que vous construirez à partir du script de démarrage standard.

Lorsque **rrdcached** tourne, il attend des données sur sa socket et de temps en temps, les écrit sur le disque. Configurons le module **rrdcached** de Collectd-pw pour cela :

```
LoadPlugin rrdcached
<Plugin "rrdcached">
    DaemonAddress "unix:/var/run/rrdcached-collectd.sock"
    DataDir "/var/lib/collectd/rrd"
    CreateFiles true
    CreateFilesAsync true
</Plugin>
```

Nous précisons la socket et indiquons où seront stockés les fichiers RRD. Il faut savoir que **rrdcached** n'est pas capable de créer un fichier RRD. Si ce fichier est absent, **rrdcached** remontera une erreur. Aussi, nous indiquons au module qu'il peut créer les fichiers avec **CreateFiles**. Enfin, pendant longtemps dans l'histoire de Collectd, la création des fichiers RRD était un goulot d'étranglement. Aucun fichier RRD ne pouvait être mis à jour pendant la création de l'un d'eux. Lorsque vous installiez une nouvelle machine, voire une dizaine de machines à la chaîne et que vous les lanciez en même temps la première fois, le serveur Collectd recevait une rafale de données impliquant la création d'autant de fichiers RRD. Cela bloquait le serveur et les données à écrire s'empilaient (d'où le mécanisme de protection que nous avons vu ci-dessus). Cela est maintenant de l'histoire ancienne, car avec **CreateFilesAsync**, les fichiers RRD sont créés de manière asynchrone et ne bloquent plus les écritures.

Pour en terminer avec cette anecdote, si vous voulez voir comment se comportait Collectd autrefois, vous pouvez déclarer un seul thread en écriture (mais 10 ou 20 en lecture), ne mettre aucun mécanisme de limitation des écritures (**WriteQueueLengthLimitHigh** et **WriteQueueLengthLimitLow** à **0** et enfin, **CreateFilesAsync** à **false**). Sur un parc de 5 serveurs, cela devrait passer. Chez votre meilleur gros client, c'est la catastrophe.

Si vous êtes pressé, vous pouvez maintenant lancer Collectd-pw avec ce fichier de configuration et attendre quelques minutes, voire quelques secondes. Les fichiers RRD devraient apparaître dans **/var/lib/collectd/rrd**. Sinon, cherchez ce qui l'empêche, généralement des problèmes de droits ou de répertoire manquant.

Note

Faites attention ! Nous avons choisi le module **rrdcached** alors qu'il existe un module **rrdtool** plus simple à l'emploi, ne nécessitant pas le démon **rrdcached**. Nous pensons qu'utiliser **rrdcached** permet de sortir ce cache de la mécanique de Collectd, ce qui a pour intérêt majeur de pouvoir relancer Collectd sans perdre le contenu de ce cache. Aussi, si vous faites le même choix que nous, nous vous donnons ce conseil, en majuscules vu son importance : NE RELANCEZ JAMAIS RRDCACHED ! SI VOUS DEVEZ LE RELANCER, RÉFLECHISSEZ AVANT. ET AU MOMENT DE LE RELANCER, RÉFLECHISSEZ ENCORE. En effet, ce démon accumule les preuves de stabilité, et vous pouvez lui confier sans problème une heure de données. Son arrêt (puis relance) vous ferait perdre ainsi jusqu'à une heure de données sur l'ensemble des serveurs gérés.

2.7 Module notify_file

Le module **notify_file** sert à stocker sur disque toutes les notifications que Collectd-pw reçoit. Initialement conçu pour **top process**, il a montré sa limite, car le disque est trop pénalisé lorsqu'il faut écrire un tout petit fichier chaque minute pour chaque serveur lorsque vous avez trop de serveurs (comptons le nombre de fichiers au bout de quelques jours pour un parc avec 500 agents : 500 x 60 x 24 x 10 jours = 7,2 millions de fichiers !). Mais, légèrement modifié, il a également montré son utilité pour stocker les notifications issues du module **sysconfig**. Aussi, voici comment le paramétrer :

```
LoadPlugin notify_file
<Plugin notify_file>
    DataDir "/var/lib/collectd/notifications"
    Plugin "sysconfig"
    # InvertPluginList false
    LinkLast "sysconfig"
</Plugin>
```

Il s'agit de stocker les données dans un répertoire que vous devez créer (et faire appartenir à l'utilisateur **collectd**) : **/var/lib/collectd/notifications**. La directive **Plugin** permet d'indiquer que nous ne voulons que les notifications issues du module **sysconfig**. N'indiquez surtout pas **top** ici, sous peine d'observer la création de millions de fichiers comme indiqué ci-dessus. Notez que le module **sysconfig** envoie une notification à chaque démarrage de Collectd-pw et ensuite, une notification toutes les 24 heures. Pour 500 serveurs sur 10 jours, cela vous fait donc 5000 fichiers seulement (plus précisément le double ou le triple, car **sysconfig** envoie 2 ou 3 notifications). Enfin, comme ces fichiers sont stockés dans des arborescences construites sur le *timestamp* de la notification, il est difficile de parcourir la liste des serveurs pour y chercher le dernier fichier de chacun. Aussi, nous demandons à Collectd-pw de créer un lien **last** pour ces fichiers, facile à trouver et pointant systématiquement vers la dernière notification. Pour cela, nous utilisons la directive **LinkLast** et indiquons le module concerné.

Avec le module **sysconfig** et cette arborescence, vous pouvez facilement obtenir la version de **collectd** sur l'ensemble de vos agents.

```
$ cd /var/lib/collectd/notifications
$ for i in *; do echo "$i : "$(zcat $i/sysconfig/last/collectd_version_info-last.gz); done
[...]
sedna : Package=collectd Version=5.3.0.20130516 Complation date=May 24 2013 19:43:53
ixion : Package=collectd Version=5.4.0.20140219 Complation date=Feb 19 2014 19:31:36
[...]
```

2.8 Module write_top

Le module **write_top** a pour objectif de stocker les données **top process** sur le disque. Successeur de **notify_file**, il en a corrigé les inconvénients en gardant en cache ces données et en les regroupant lors de leur écriture sur disque. Le paramétrage est le suivant :

```
LoadPlugin write_top
<Plugin write_top>
    DataDir "/var/lib/collectd/top"
    FlushWhenOlderThanMin 30
    FlushWhenBiggerThanK 500
</Plugin>
```

Vous reconnaissez comment indiquer le nom du répertoire (n'oubliez pas de le créer au préalable !). Vous paramétrez également quand écrire sur disque les données en cache. Lorsque le cache d'un serveur est trop vieux ou trop grand, il faut l'écrire. En effet, il ne faut pas prendre le risque de perdre ce cache et il faut donc l'écrire régulièrement (directive **FlushWhenOlderThanMin**). Mais s'il est trop gros, il ne faut pas non plus faire gonfler la mémoire, mieux vaut l'écrire à tout moment, passé un certain seuil de taille (directive **FlushWhenBiggerThanK**).

2.9 Module jsonrpc

Nous continuons avec le module **jsonrpc**, le fleuron discret du projet PerfWatcher. En effet, ce module permet d'interroger Collectd-pw directement par une interface Json-RPC et d'effectuer des requêtes comme **LISTVAL** ou **GETVAL**. Mais d'autres requêtes ont été implémentées spécifiquement pour

PerfWatcher pour qu'il puisse disposer des informations les plus récentes directement depuis Collectd-pw et avec la charge la plus réduite pour le système.

L'origine de ce module vient d'une tentative de disposer de ces informations avec un mécanisme inverse. Collectd-pw disposait alors d'un module (maintenant abandonné) qui consistait à écrire en base toutes les donnés dont il disposait. La base (MySQL) n'a pas supporté ce trop grand nombre de requêtes, en écriture de la part de Collectd-pw et en lecture de la part de PerfWatcher, avec du ménage de temps en temps. Le stockage étant assuré par ailleurs (les fichiers RRD), il a été décidé de repenser complètement cette interface entre Collectd-pw et PerfWatcher et de l'implémenter via un serveur web, embarqué dans Collectd-pw (d'où le prérequis **libmicrohttpd**), permettant de passer facilement les proxies grâce au HTTP, et de se conformer à la norme JSON-RPC pour simplifier les traitements JavaScript qui s'ensuivent au niveau de PerfWatcher.

Par la suite, avec l'arrivée du module **write_top**, il est devenu plus difficile de consulter les fichiers **top process** du fait du nouveau format de fichier. Et il fallait également pouvoir demander de vider le cache pour un serveur. Le module **jsonrpc** étant facilement extensible, il a été décidé qu'il servirait également d'interface pour consulter les fichiers **top process**. Cela implique donc que si vous souhaitiez séparer certains modules de Collectd-pw sur plusieurs instances (comme nous le verrons plus loin), il est indispensable de conserver **jsonrpc** et **write_top** ensemble. Enfin, avec la version 2.0 de PerfWatcher qui peut être installée sur un autre serveur que celui sur lequel tourne Collectd, il faut pouvoir l'interroger sur les fichiers RRD et même créer des graphes à la volée. Il a été choisi une nouvelle fois d'étendre **jsonrpc** pour ce travail. Il doit donc également connaître l'emplacement des fichiers RRD et savoir se servir de **rrdtool** et **rrdcached**.

```
LoadPlugin jsonrpc
<Plugin jsonrpc>
    Port "8080"
    MaxClients 10
# Note : write_top and jsonrpc work together.
# This is why we need to define this path twice.
    DataDir "/var/lib/collectd/rrd"
# If you prefer to use rrdcached, uncomment and update the next line
    RRDCachedDaemonAddress "/var/run/rrdcached-collectd.sock"
    RRDToolPath "/usr/bin/rrdtool"
    TopPsDataDir "/var/lib/collectd/top"
</Plugin>
```

Nous indiquons ici le port du serveur. Il est possible de le protéger par une authentification basique, mais PerfWatcher ne sait pas encore montrer patte blanche. Ensuite, nous limitons le nombre de clients sur serveur web à 10 connexions en parallèle. Ce paramétrage est à évaluer en fonction du nombre de vos utilisateurs qui pourraient interroger PerfWatcher/Collectd-pw au même moment. Enfin, nous indiquons à nouveau là où se trouvent les fichiers RRD et **top process**, ainsi que le nécessaire pour **rrdtool** et **rrdcached**. Cela semble redondant, à raison. Il s'agit d'une limitation du module de lecture du fichier de configuration de Collectd. Cela permet, à l'inverse, de pouvoir faire fonctionner un seul de ces deux modules, **jsonrpc** et **write_top**, si jamais vous en aviez une bonne raison (mais nous n'en avons pas encore trouvé de bonne).

Note

Les nouvelles options de Collectd-pw pour PerfWatcher-2.0 apparaissent dans la version 5.4.0-20131129. Mais un bug se manifestant sur certaines plateformes (dont Debian 7) n'est corrigé qu'à partir de la version 5.4.0-20140219. Vous devez donc installer cette version au minimum. Et bien évidemment la dernière si possible, si une nouvelle version est sortie entre-temps.

2.10 Module basic_aggregator

Ce module est l'autre fleuron de Collectd-pw et PerfWatcher au point d'en être une fonctionnalité remarquable de ce dernier. En effet, il permet d'agréger des données. Conçu au même moment que le module **aggregator** intégré à Collectd, il a permis de constater qu'il y avait plusieurs visions de l'agrégation de données. Celui de Collectd s'oriente vers l'agrégation des données d'un module pour un serveur, par exemple tous les CPU pour calculer un **cpu_total**.

Basic_aggregator adopte une vision plus globale, en agrégeant les données de plusieurs serveurs. De plus, cela peut se faire par type, sans avoir à indiquer l'instance. Ainsi, il est possible d'agréger toutes les interfaces réseau de tous les serveurs d'une liste sans avoir à préciser que pour tel serveur, il s'agit d'**eth0** et pour tel autre, **eth2**. Cependant, **basic_aggregator** a été codé un peu trop rapidement dans l'objectif de ne pas perdre trop d'énergie à le réaliser, pour qu'il s'efface par la suite, derrière **aggregator**, celui de Collectd. Ne répondant finalement pas à la même demande, il doit rester, mais ne semble pas intéresser les auteurs de Collectd pour y être intégré. Peut-être est-il temps d'étudier les besoins, maintenant que nous avons deux solutions à deux demandes distinctes, et qu'un nouvel agrégateur contenterait tout le monde ? Voici le paramétrage :

```
LoadPlugin basic_aggregator
<Plugin basic_aggregator>
    Aggregators_config_file "/opt/perfwatcher/current/etc/aggregator-localhost.conf"
</Plugin>
```

Le fichier indiqué ici est créé par le script **bin/aggregator** de PerfWatcher. Il s'appelle plus exactement **$aggregator_config_dir/aggregator-<source Collectd>.conf** où **$aggregator_config_dir** et **<source Collectd>** sont définis dans le fichier de configuration de PerfWatcher. Nous y reviendrons.

Ce paramétrage simpliste a deux explications. La première est que le véritable paramétrage s'effectue depuis

PerfWatcher. La seconde est un besoin de pouvoir modifier la configuration à tout moment sans avoir à arrêter et redémarrer Collectd. Ce mécanisme de rechargement de la configuration n'étant pas possible dans Collectd, il a dû être intégré au module en attendant que les développeurs de Collectd proposent une solution. Notez encore ici une différence avec **aggregator**, dont un changement dans la configuration nécessite de relancer Collectd.

Enfin, notez que l'agrégation des CPU est intégrée à Collectd-pw sans que vous n'ayez rien d'autre à faire que de charger le module **cpu**.

2.11 Les autres modules

Nous avons laissé de côté les modules **sysconfig** et **top**, qui sont deux modules d'agent. Ils ne nécessitent aucun paramétrage. Vous devez juste penser à les charger dans votre fichier **/etc/collectd/collectd.conf** de votre agent. À partir de maintenant, avec ces deux modules et surtout **top**, vous ne devriez plus vous passer de Collectd-pw, aussi bien sur le serveur où il est indispensable, que sur les agents pour ces fonctionnalités que cela ajoute à Collectd.

Vous pouvez maintenant arrêter et relancer le serveur et vous assurer que vos agents tournent bien avec les modules **top** et **sysconfig** chargés. Cela se voit dans les fichiers des sous-répertoires de **/var/lib/collectd** qui doivent se créer (parfois au bout de 30 minutes si vous avez paramétré ainsi le module **write_top**). Laissez tourner. À partir de maintenant, ce n'est pas parce que PerfWatcher n'est pas encore installé que vous perdez du temps. À l'inverse, si vous installez PerfWatcher trop vite, vous risquez d'être déçu avec trop peu de données à visualiser (en particulier une timeline qui risque d'être vide).

2.12 La maintenance des répertoires notification et top

Autant les fichiers RRD sont créés une bonne fois pour toutes, autant les fichiers **sysconfig** (notifications) et **top** s'accumulent. Vous allez donc devoir créer un script de maintenance en fonction de vos besoins en archivage. Le principe sera de supprimer tous les fichiers ayant passé un âge limite, par exemple avec la commande **find /var/lib/collectd/top -type f -mtime +30 -delete** pour supprimer ceux de plus de 30 jours. Vous devrez aussi supprimer les répertoires vides qui vont s'accumuler, peut-être avec un **find /var/lib/collectd/top -type d -exec rmdir {} \;** en redirigeant les nombreuses erreurs vers le trou noir **/dev/null**. Ces scripts de maintenance sont indispensables, mais n'existent pas encore pour Collectd-pw, ni pour PerfWatcher, que nous allons voir ci-dessous. Pour le moment, chacun crée les siens en fonction de ses besoins et de ses contraintes.

3 PerfWatcher

L'installation de PerfWatcher, à l'instar de Collectd-pw, nécessite des prérequis. Vous aurez besoin de :

- Apache, PHP, MySQL (client et serveur) ;
- PHP en ligne de commandes (**php-cli** ou **php5-cli** en général sur votre distribution) ;
- Les modules Curl, MySQL et Pear pour PHP (**php(5)-curl**, **php(5)-mysql** et **php(5)-pear**).

Vous installerez ensuite le module MySQL de Pear :

```
pear install MDB2_Driver_mysql
```

Cela devrait, par résolution de dépendances, vous installer également MDB2. Ce module a été choisi en remplacement des fonctions natives MySQL qui sont devenues obsolètes en PHP. Il existe maintenant plusieurs implémentations, mais il en fallait une qui dispose également de connecteurs vers d'autres bases comme Postgres, voire Oracle, Sybase, MSSQL... C'est MDB2 qui a été choisi, même si cela pourrait prêter à discussion.

Vous pouvez maintenant décompacter l'archive de PerfWatcher dans **/opt/perfwatcher/<version>**. Nous vous invitons à créer un lien vers ce répertoire :

```
$ ln -s /opt/perfwatcher/<version> /opt/perfwatcher/current
```

Cela vous évitera de reconfigurer à chaque fois le module **basic_aggregator** de Collectd-pw et Apache. La configuration du serveur web Apache est assez commune (et inutile si vous avez installé PerfWatcher directement dans le répertoire **htdocs**).

```
<IfModule mod_alias.c>
    Alias /perfwatcher /opt/perfwatcher/current
</IfModule>
<Directory /opt/perfwatcher/current>
    AuthType Basic
    AuthUserFile /etc/.../htpasswd
    AuthGroupFile /etc/.../htgroups
    AuthName "Perfwatcher"
    require group perfwatcher
</Directory>
```

La configuration de PerfWatcher nécessite dans un premier temps une base de données. Tout y est décrit dans le fichier **install/README** que nous interprétons ici :

```
$ mysql -u root -p
mysql>CREATE DATABASE perfwatcher;
mysql> GRANT ALL PRIVILEGES ON perfwatcher.* TO
'perfwatcher'@'localhost' IDENTIFIED BY 'changeme';
mysql> exit; Bye
$ cat create.sql | mysql -u perfwatcher -h localhost -p perfwatcher
```

Vous devez ensuite créer un fichier de configuration **etc/config.php**. Pour cela, recopiez le fichier **etc/config.sample.php** et modifiez-le pour qu'il corresponde à votre installation. Il est normalement bien commenté et, si vous installez un seul Collectd-pw et PerfWatcher sur le même serveur (que vous nommerez en tant que source « localhost »), il ne nécessite pas d'explications si vous avez réussi à installer Collectd-pw.

Enfin, il faut programmer l'exécution automatique et périodique de deux scripts dans la **crontab** :

- **peuplator** qui, comme son nom l'indique en bon franglais, *peuplera* des arborescences de serveurs en fonction de certains critères ;
- **aggregator** qui, comme son nom ne l'indique pas exactement, même en bon anglais, créera le(s) fichier(s) de configuration de l'agrégateur.

Voici ce que nous vous proposons d'indiquer dans un fichier **/etc/cron.d/perfwatcher** :

```
* * * * * apache (date; /opt/perfwatcher/current/bin/peuplator; date)
>> /opt/perfwatcher/logs/peuplator.log 2>&1
* * * * * apache (date; /opt/perfwatcher/current/bin/aggregator;
date) >> /opt/perfwatcher/logs/aggregator.log 2&1
```

Cela vous obligera d'une part à créer un répertoire **/opt/perfwatcher/logs** appartenant au compte **apache** (ou **www-data**, cela dépend de votre système) ; d'autre part, vous devrez créer un fichier pour la rotation des journaux, par exemple **/etc/logrotate.d/perfwatcher** :

```
/opt/perfwatcher/logs/*.log {
    daily
    rotate 7
    missingok
    compress
}
```

L'absence de fichier de configuration de l'agrégateur n'est pas inquiétante en soi, tant que vous n'avez pas configuré d'agrégateur dans PerfWatcher. Par contre, lorsque vous en aurez au moins un, attendez une minute que le script se lance, puis vérifiez sa présence. Vous en profiterez pour voir son contenu.

Mais au fait, maintenant que c'est installé, après ce rude labeur, avez-vous été sur votre nouveau PerfWatcher, sur http://votreserveur/perfwatcher ? Vous pouvez renommer la racine, ajouter des répertoires, des serveurs...

4 Maintenance

Une application bien conçue sur un serveur bien entretenu ne devrait pas avoir de maintenance. Cependant, au moins au début, vérifiez tous vos journaux, aussi bien au niveau de leur contenu que de leur rotation/suppression. Il ne faudrait pas qu'un **collectd.log** grossisse au point de remplir votre système de fichiers.

Vous regarderez également, dans PerfWatcher puisque c'est une de ses fonctions, l'usage de vos disques et de vos systèmes de fichiers. Une pente montante est normale au début pour le système de fichiers qui héberge **/var/lib/collectd**. Cependant, elle devrait se stabiliser.

Par ailleurs, il serait bon d'avoir une surveillance sur la présence de tous les agents. Le plus simple est de la constater par l'objectif à atteindre : la mise à jour des fichiers RRD et/ou la présence de fichiers **top process**. Nous préférons les fichiers RRD, car eux seuls garantissent que l'agent tourne. L'absence du module **top** peut s'expliquer simplement par le fait que l'agent est un agent Collectd et non Collectd-pw. Ou même plus simplement, que le module **top** n'est pas chargé (absent du fichier de configuration). Le principe est alors que, pour chaque serveur, on repère le fichier le plus récent (ou de manière plus simple, on cherche un fichier mis à jour il y a moins d'un jour avec **find**). Lorsque le résultat est satisfaisant, l'agent est considéré comme fonctionnel. Sinon, il y a un problème à régler, qui peut être un serveur éteint, planté, fonctionnel mais avec un problème réseau, ou avec tout simplement un dysfonctionnement sur Collectd(-pw). À vous de surveiller cette liste de serveurs, par exemple tous les matins en vous brossant les dents.

Pensez également à vérifier les fichiers RRD du module **tcpconns** qui se créent lorsqu'un port est ouvert, et peuvent ne plus avoir d'utilité lorsque le port est fermé, si le numéro de port est aléatoire.

Pour déterminer la liste des fichiers RRD non mis à jour, vous pouvez utilisez une commande de ce genre : **find /var/lib/collectd/rrd -type f -mtime +30 -name "*.rrd" -delete**.

Si vous ne vous intéressez qu'à la liste des serveurs non mis à jour, vous pouvez vous en sortir avec deux requêtes JSON-RPC.

```
$ curl --data-urlencode '{"jsonrpc": "2.0", "method": "pw_get_dir_
hosts", "params": "", "id": 0}' http://localhost:8080
```

Cette première requête demande la liste des serveurs ayant des fichiers RRD. Vous devez l'analyser et la reformater (voire la filtrer selon des règles qui vous sont propres, telle une liste d'exclusion). Puis, vous fournissez la liste de ces serveurs dans une seconde requête JSON-RPC :

```
$ curl --data-urlencode '{"jsonrpc": "2.0", "method": "pw_get_
status", "params": { "timeout": 240, "server": ["serveur1",
"serveur2", ..., "serveurN" ] }, "id": 0}' http://localhost:8080
```

Le résultat indiquera pour chaque serveur un état. Lorsque celui-ci est **unknown**, cela signifie qu'il n'est pas dans le cache de Collectd. Autrement dit, c'est un serveur qui n'alimente plus Collectd, indiquant soit un problème, soit que vous pouvez supprimer les fichiers RRD correspondants.

Si les requêtes JSON-RPC vous intéressent, en l'absence de documentation, vous pouvez lire le code source qui donne en commentaire des explications : `src/jsonrpc_cb_*.c`.

Conclusion

Arrivés à la fin de cet article, nous avons normalement un serveur Collectd-pw et un PerfWatcher qui tournent, ainsi que des agents qui envoient leurs données à Collectd.

Note **La version des logiciels utilisés pour cet article**

Au moment où cet article est écrit, les dernières versions des logiciels utilisés sont :

- Collectd 5.4.1 ;

- Collectd-pw 5.4.0-20140219 ;

- PerfWatcher 2.0-20140221.

Collectd-pw est prêt pour communiquer avec PerfWatcher-2.0 depuis novembre 2013 (à un bug près, corrigé dans la version 5.4.0-20140219). Nous avons naturellement utilisé la dernière version disponible. Notez par ailleurs que notre serveur, *ixion*, disposait bien d'une version récente. Notre machine virtuelle, *sedna*, par contre, faisait tourner une version 5.3.0.20130516. Vous pouvez constater la différence sur le graphe mémoire des deux machines (voir l'article d'introduction).

Vous pouvez donc maintenant commencer à utiliser votre nouvel outil. Le prochain article aura pour objet l'utilisation avancée de Collectd et PerfWatcher, c'est-à-dire des problématiques que vous risquez de rencontrer avec des explications et des propositions de choix possibles. ■

Remerciements

L'auteur tient à remercier tous ceux sans qui Collectd, Collectd-pw et PerfWatcher ne seraient pas. Florian Foster et les nombreux contributeurs de Collectd. Cyril Feraudet pour avoir lancé le projet PerfWatcher qu'il maintient toujours, et pour avoir commencé à écrire des patchs pour Collectd, ce qui allait devenir Collectd-pw. En tant que contributeur de nombreux patchs de Collectd-pw, l'auteur remercie également ceux qui le soutiennent, ils se reconnaîtront.

Liens

Collectd : http://www.collectd.org

PerfWatcher/Collectd-pw : http://perfwatcher.org

Sur GitHub : https://github.com/perfwatcher

GRAPHING, LOGGING ET MONITORING 2.0 : VOS ALERTES AVEC SENSU

par Benoît Benedetti
[Administrateur Système Linux Université de Nice Sophia Antipolis]

De plus en plus décrié, nombre d'administrateurs système se détournent de Nagios, après de nombreuses années de bons et loyaux services. Dans ces colonnes, vous avez déjà pu apprendre à connaître Icinga, le fork de Nagios, et surtout Shinken, qui essayent de combler certaines lacunes de Nagios. Nous allons dans cet article regarder ce qui se passe du côté de Sensu, un projet qui a pris le parti de développer de zéro une solution de monitoring.

1 Présentation

Sensu **[SENSU]** est décrit comme un routeur d'alertes (*monitoring router*) : le serveur reçoit les messages des clients du réseau, et les connecte à des notifications. Vous définissez quels *checks* (scripts de contrôle) exécuter sur vos machines, et planifiez leur exécution depuis le serveur Sensu. Un check peut tester la disponibilité d'un service, ou récupérer des métriques. Les checks peuvent s'écrire dans le langage de votre choix (shell, Perl, Python, Ruby pour vos checks d'administration, CasperJS pour vos tests d'interface web, ou même R pour vos scripts d'analyse statistique) ; certains sont disponibles et vous pouvez ré-utiliser les scripts Nagios existants. Un check va retourner le résultat de son exécution au serveur. Suivant le message reçu du client (erreur, métriques), le serveur va router le message vers un *handler* (gestionnaire d'alertes, qui peut être lui aussi écrit dans le langage de votre choix) : envoi d'un e-mail, redémarrage d'un service, etc.

Sensu repose donc sur une architecture très simple, basée sur des messages au format JSON **[JSON]**. Sa configuration est elle aussi au format JSON, et peut être éclatée en plusieurs fichiers : elle peut donc être facilement gérée par un système de gestion de configuration type Puppet, Salt, ou encore CFEngine. Une autre particularité qui rend Sensu facilement scalable et utilisable pour gérer des infrastructures élastiques, est l'enregistrement automatique des clients auprès du serveur : pas besoin de modifier la configuration du serveur et d'effectuer un redémarrage du service à chaque ajout d'un nouvel hôte à monitorer.

Sensu est écrit en Ruby, basé sur EventMachine. Sa flexibilité repose sur un ensemble de briques logicielles (Fig. 1), qui peuvent facilement passer à l'échelle suivant la taille de votre infrastructure et le nombre de machines à monitorer. Les clients et le serveur utilisent un serveur AMQP pour communiquer, Sensu nécessitant RabbitMQ : un service client doit tourner sur toutes les machines que vous souhaitez monitorer, et va effectuer les checks, puis retourner leur résultat au serveur via RabbitMQ. Une base de données NoSQL Redis est utilisée par le serveur pour stocker les informations persistantes, comme les détails des différents clients et leur état. Un service d'API REST permet d'accéder à ces différentes données. Une *dashboard* (interface web) vous permet d'avoir une vue d'ensemble de l'état de votre infrastructure Sensu, et d'effectuer des actions, comme désactiver temporairement des alertes, tout ça via l'API (Fig. 1).

2 Installation

Nous allons installer Sensu 0.12, dernière version stable du projet. La documentation officielle est très claire et complète **[INSTALLATION]**. Nous allons la reprendre ici de manière plus concise, pour revenir sur les différentes briques qui constituent Sensu. Nous installerons dans cette partie Sensu en tant que client et serveur sur la même machine `serveur-sensu`. Nous commencerons donc par tester services client et serveur sur la même machine. Nous installerons, dans la dernière partie de cet article, une deuxième machine `client-sensu` en tant que client.

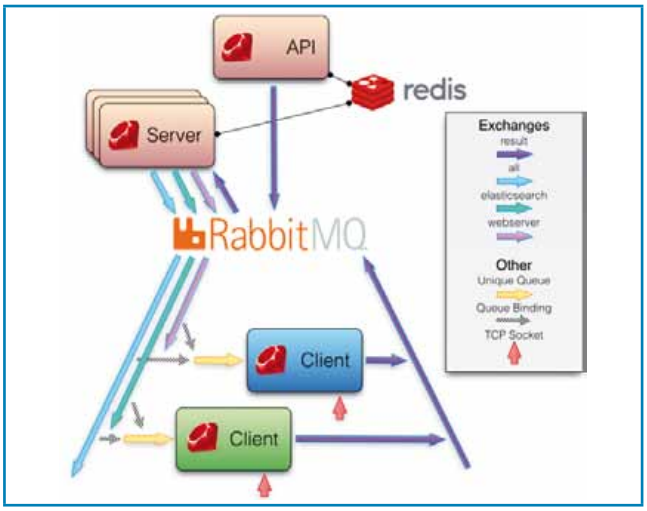


Fig. 1

2.1 Prérequis serveur

Les commandes suivantes sont à exécuter seulement sur une machine que vous souhaitez utiliser comme serveur Sensu. Il faut donc installer un serveur RabbitMQ, en activant les dépôts officiels du projet :

```
$ wget http://www.rabbitmq.com/rabbitmq-signing-key-public.asc -O - |
sudo apt-key add -
$ echo "deb http://www.rabbitmq.com/debian/ testing main" | sudo tee
-a /etc/apt/sources.list.d/rabbitmq.list
$ sudo aptitude clean
$ sudo aptitude update
$ sudo aptitude -y install rabbitmq-server
```

Par défaut, RabbitMQ écoute en clair sur le port 5672. Nous allons utiliser un script existant pour générer des certificats SSL et chiffrer les communications :

```
$ wget http://sensuapp.org/docs/0.12/tools/ssl_certs.tar -O /tmp/ssl_certs.tar
$ tar xf /tmp/ssl_certs.tar -C /tmp
$ cd /tmp/ssl_certs/
$ sudo ./ssl_certs.sh generate
$ sudo mkdir /etc/rabbitmq/ssl
$ sudo cp  sensu_ca/cacert.pem server/cert.pem server/key.pem /etc/rabbitmq/ssl/
```

Créez le fichier **/etc/rabbitmq/rabbitmq.config** de configuration suivant, pour que RabbitMQ écoute sur un nouveau port 5671, en SSL (n'oubliez pas le point final) :

```
[
    {rabbit, [
    {ssl_listeners, [5671]},
    {ssl_options, [{cacertfile,"/etc/rabbitmq/ssl/cacert.pem"},
                   {certfile,"/etc/rabbitmq/ssl/cert.pem"},
                   {keyfile,"/etc/rabbitmq/ssl/key.pem"},
                   {verify,verify_peer},
                   {fail_if_no_peer_cert,true}]}
  ]}
].
```

On crée un hôte virtuel RabbitMQ **/sensu**, dans lequel tous les échanges clients et serveur auront lieu :

```
$ sudo rabbitmqctl add_vhost /sensu
```

On crée un identifiant de connexion **sensu**, que clients et serveur utiliseront pour se connecter à l'hôte virtuel **/sensu** :

```
$ sudo rabbitmqctl add_user sensu mypass
$ sudo rabbitmqctl set_permissions -p /sensu sensu ".*" ".*" ".*"
```

On installe la console web de gestion :

```
$ sudo rabbitmq-plugins enable rabbitmq_management
```

La console de gestion possède un utilisateur **guest** par défaut. On donne les droits à cet utilisateur sur l'hôte virtuel :

```
$ sudo rabbitmqctl set_permissions -p /sensu guest ".*" ".*" ".*"
```

Ajoutez le service RabbitMQ au démarrage du système, puis redémarrez-le :

```
$ sudo update-rc.d rabbitmq-server defaults
$ sudo /etc/init.d/rabbitmq-server restart
```

Si RabbitMQ est correctement configuré, vous devriez pouvoir vous connecter à la console de gestion (utilisateur **guest**, mot de passe **guest**), en vous rendant à l'adresse http://ip.du.serveur.sensu:15672, via votre navigateur.

Comme prérequis serveur, il ne nous reste plus que Redis à installer. Installation beaucoup plus simple que RabbitMQ, surtout depuis que la Debian Wheezy possède par défaut la version 2.4 (Sensu nécessite Redis en version 2 minimum, ce qui demandait d'activer les dépôts *backports* sous Debian 6) :

```
$ sudo aptitude -y install redis-server
```

Redis devrait être démarré par défaut, ajoutez le service au démarrage du système :

```
$ sudo service redis-server status
redis-server is running
$ sudo update-rc.d redis-server defaults
update-rc.d: using dependency based boot sequencing
```

2.2 Installation de Sensu

L'installation de Sensu est grandement facilitée par sa mise à disposition sous forme de paquet Omnibus **[OMNIBUS]**, permettant d'inclure toutes les dépendances nécessaires à l'application (Ruby, Rubygems, etc.). C'est la procédure d'installation que nous allons suivre, recommandée par la documentation officielle, plutôt que l'utilisation depuis les sources. Commencez par ajouter le dépôt officiel de Sensu à votre système :

```
$ wget -q http://repos.sensuapp.org/apt/pubkey.gpg -O- | sudo apt-key add -
$ sudo echo "deb     http://repos.sensuapp.org/apt sensu main" | sudo tee
-a /etc/apt/sources.list.d/sensu.list
```

Mettez à jour votre système de paquets, puis installez l'unique paquet nécessaire :

```
$ sudo aptitude update
$ sudo aptitude -y install sensu
```

Sensu est livré avec une installation récente de Ruby, car beaucoup de checks disponibles par défaut sont écrits en Ruby :

```
$ /opt/sensu/embedded/bin/ruby -v
ruby 2.0.0p353 (2013-11-22 revision 43784)
[x86_64-linux]
```

Nous n'avons pas installé Ruby, nous allons profiter de ce Ruby embarqué, il faut donc le préciser dans la configuration de Sensu, dans le fichier **/etc/default/sensu** :

```
EMBEDDED_RUBY=true
```

On en profite aussi pour l'ajouter au **PATH**, nous aurons besoin d'exécuter des commandes Ruby plus tard :

```
$ export PATH=$PATH:/opt/sensu/embedded/bin
```

Copiez ensuite les certificats SSL générés précédemment pour RabbitMQ, les versions client cette fois-ci, dans un dossier dédié **/etc/sensu/ssl** :

```
$ sudo mkdir /etc/sensu/ssl
$ sudo cp /tmp/ssl_certs/client/key.pem /tmp/
ssl_certs/client/cert.pem /etc/sensu/ssl/
```

Activez au démarrage du système les différents services nécessaires à Sensu :

```
$ sudo update-rc.d sensu-server defaults
$ sudo update-rc.d sensu-api defaults
$ sudo update-rc.d sensu-dashboard defaults
$ sudo update-rc.d sensu-client defaults
```

Ces différents services lisent leur configuration depuis des fichiers JSON du dossier **/etc/sensu/conf.d**. On commence par le fichier **/etc/sensu/conf.d/rabbitmq.json**, dont ces services ont besoin pour communiquer avec le serveur RabbitMQ :

```
{
  "rabbitmq": {
    "ssl": {
      "cert_chain_file": "/etc/sensu/ssl/cert.pem",
      "private_key_file": "/etc/sensu/ssl/key.pem"
    },
    "host": "localhost",
    "port": 5671,
    "vhost": "/sensu",
    "user": "sensu",
    "password": "mypass"
  }
}
```

Le fichier équivalent **/etc/sensu/conf.d/redis.json** pour Redis :

```
{
  "redis": {
    "host": "localhost",
    "port": 6379
  }
}
```

Le fichier de configuration **/etc/sensu/conf.d/api.json** de l'API :

```
{
  "api": {
    "host": "localhost",
    "port": 4567,
    "user": "admin",
    "password": "secret"
  }
}
```

On définit un handler par défaut **/etc/sensu/conf.d/handler_default.json** :

```
{
  "handlers": {
    "default": {
      "type": "pipe",
      "command": "true"
    }
  }
}
```

Note

Ce handler ne fait rien, mais par défaut, les checks s'attendent à pouvoir utiliser un handler nommé **default**.

Et enfin, le fichier de configuration **/etc/sensu/conf.d/dashboard.json** de l'interface web :

```
{
  "dashboard": {
    "port": 8080,
    "user": "admin",
    "password": "secret"
  }
}
```

Sensu est installé et configuré, on peut démarrer les différents services :

```
$ sudo /etc/init.d/sensu-server start
$ sudo /etc/init.d/sensu-api start
$ sudo /etc/init.d/sensu-dashboard start
```

Chaque service a son fichier de logs dans le dossier **/var/log/sensu/**, à consulter pour valider son bon démarrage. Vous devriez pouvoir vous connecter à la dashboard de Sensu sur le port 8080 de votre serveur, identifiant **admin**, mot de passe **secret** (comme configurés dans le fichier **dashboard.json**). L'interface est très épurée et vide pour l'instant, car nous n'avons pas configuré de client, ni de checks. Pour le moment, rendez-vous dans le menu ***Info***, pour valider que Sensu fonctionne correctement et arrive à se connecter aux serveurs Redis et RabbitMQ (Fig. 2).

Fig. 2

2.3 Votre premier client

Nous n'avons pas démarré le service client, car nous n'avons pas encore défini sa configuration. On va définir sa configuration dans un fichier **/etc/sensu/conf.d/client.json** :

```
{
      "client": {
        "name": "serveur-sensu",
        "address": "192.168.0.11",
        "subscriptions": [ "base", "cron" ]
      }
}
```

Une section **client** prend un nom, une adresse et une liste de canaux d'abonnements (option **subscriptions**, qui prend un tableau Ruby comme

valeur). Le nom et l'adresse ne servent pas à se connecter à un client, mais juste à nommer ce client. Vous pouvez mettre ce que bon vous semble, mais restez logique et mettez les valeurs adéquates. Au démarrage, le service client va lire **client.json** et va seulement lire la section **rabbitmq** du fichier **config.json**, et se connecter à RabbitMQ en créant une queue, pour communiquer avec le serveur. De plus, pour chaque valeur du tableau de l'option **subscriptions**, le client va s'abonner à un *exchange* RabbitMQ du même nom (visible dans la console de gestion de RabbitMQ, onglet ***exchanges***, **base** et **cron** ont été créés) : lorsque le serveur enverra un message sur un exchange, tout client abonné le recevra. Ici, notre client s'est abonné aux exchanges **base** et **cron**. Démarrez le client :

```
$ sudo /etc/init.d/sensu-client start
```

Le client va se connecter à RabbitMQ. Nul besoin de redémarrer le serveur qui va découvrir ce nouveau client la prochaine fois qu'il va contacter le serveur RabbitMQ : finis les redémarrages intempestifs à la Nagios du serveur de monitoring à chaque ajout de nouveau client ! L'onglet ***Clients*** de la dashboard de Sensu devrait afficher notre client (Fig. 3). Si vous cliquez sur ce client, une popup apparaît (Fig. 4) pour supprimer le client ou simplement le désactiver des différents checks à exécuter. Mais encore faudrait-il que nous ayons des checks configurés...

Fig. 3

3 Checks

Dès qu'un client Sensu est découvert par le serveur, celui-ci vérifie la connectivité (*Keepalive*) du client à RabbitMQ. Si le service client ne répond plus pendant plus de 180 secondes, une alerte apparaîtra sur la dashboard. Sensu peut bien sûr effectuer des checks beaucoup plus avancés. Un check est une commande (le plus souvent un script) exécutée sur le client. Suivant l'exécution de cette commande, le client renvoie seulement un code retour au serveur : 0 pour une réussite, 1 pour un *warning*, 2 pour une erreur critique. Tout code de retour supérieur ou égal à 3 peut être utilisé pour un warning maison.

Les checks peuvent également être utilisés pour la collecte de métriques (charge serveur, etc.), puis envoyer au serveur cette mesure de données. Serveur qui va ensuite alimenter avec cette mesure un service de graphiques, par exemple, à l'aide d'un handler, comme nous le verrons plus loin.

Fig. 4

3.1 Les bases

Il est plus facile à maintenir d'écrire un fichier de configuration par check. Dans cet article, pour notre exemple et par simplicité, nous allons regrouper plusieurs checks d'exemple dans un seul et même fichier **/etc/sensu/conf.d/check_exemples.json** :

```
{
  "checks": {

    "ok1": {
     "handlers": ["default"],
     "command": "echo -n 'exécution de ok1 '",
     "subscribers": ["base"],
     "interval": 20
    },

    "ok2": {
     "command": "echo -n 'exécution de ok2 ' && exit 0",
     "subscribers": ["base"],
     "interval": 20
    },

    "warning1": {
     "command": "echo -n 'exécution de de warning1 ' && schmilblick --papy --mougeot",
     "subscribers": ["base"],
     "interval": 20
    },

    "warning2": {
     "command": "echo -n 'exécution de warning2 ' && exit 1",
     "subscribers": ["base"],
     "interval": 20
    },

    "alerte1": {
     "command": "echo -n 'exécution de alerte1 ' && ls /toto",
     "subscribers": ["base"],
     "interval": 20
    },

    "alerte2": {
     "command": "echo -n 'exécution de alerte2 ' && exit 2",
     "subscribers": ["base"],
     "interval": 20
    },

    "custom": {
     "command": "echo -n 'exécution de custom ' && exit 3",
     "subscribers": ["base"],
     "interval": 20
    },

    "metrique": {
     "type": "metric",
     "command": "echo servers.$(hostname -s).load 10 `date +%s`",
     "subscribers": ["base"],
```

```
    "interval": 20
  }

 }
}
```

On inclut un ou plusieurs check(s) dans une section **checks**. Pour chaque check, on indique un nom unique. Ici **ok1**, **ok2**, **warning1**, etc. Ensuite, l'option **command** précise la commande à exécuter sur le client. Dans le premier check, la commande est un simple **echo**, qui réussira donc toujours et enverra un code 0 vers le serveur Sensu, qui ne générera pas d'alerte. Idem pour le deuxième check **ok2**, qui cette fois sort explicitement avec un code 0. Le check **warning1** finit par une commande **schmilblick**, qui n'existe pas. La commande de ce check renverra un code 1 au serveur, qui générera un warning. Idem pour le check suivant **warning2**, avec un **exit** explicite. Le check **alerte1** finit par un **ls** sur un dossier **/toto**, qui n'existe pas. La commande de ce check renverra un code 2, et le serveur générera une alerte. Idem pour le check suivant **alerte2**, à l'aide d'un **exit** explicite. L'avant-dernier check **custom** est un warning maison créé à l'aide du code retour 3.

Tous ces checks sont des checks que je qualifierai à partir de maintenant de « classiques », qui servent à tester la disponibilité d'un service, la réussite d'une commande. C'est le type par défaut. Pour créer un check qui collecte des métriques, on utilise l'option **type** avec la valeur **metric**, comme pour le dernier check nommé **metrique**. Ce check fait un simple **echo** de la chaîne de caractères **servers.serveur-sensu.load**, suivie du nombre 10 (une métrique factice, à titre d'exemple) et d'un horodatage au format epoch.

On assigne un handler à un check avec le paramètre **handlers**, qui prend un tableau de noms de handlers, qui doivent exister. Pour le premier check **ok1**, je lui ai assigné le check **default** défini précédemment. En fait, cet assignement du check **default** est optionnel : lorsqu'aucun paramètre **handlers** n'est spécifié dans la configuration d'un check, il se voit automatiquement attribué ce check **default**. Tous les checks suivants ont donc le handler **default** assigné par défaut.

Pour continuer dans les options de définition d'un check, **subscribers** précise à quels clients s'applique le check. Ici, tous nos checks seront appliqués aux clients abonnés à l'exchange **base**, auquel est abonné notre seul et unique client s'exécutant sur **serveur-sensu**. Pour chaque check, un intervalle d'exécution très court de 20 secondes a été configuré, pour faciliter les tests. Dans la réalité, vous utiliserez plutôt un intervalle d'une ou plusieurs minutes pour certains checks.

Les checks configurés, redémarrez API et serveur :

```
$ sudo /etc/init.d/sensu-api restart
$ sudo /etc/init.d/sensu-server restart
```

Les checks configurés devraient apparaître dans la section ***Checks*** de la dashboard Sensu (Fig. 5). Les différents warnings et alertes sont visibles dans la section ***Current Events*** (Fig. 6).

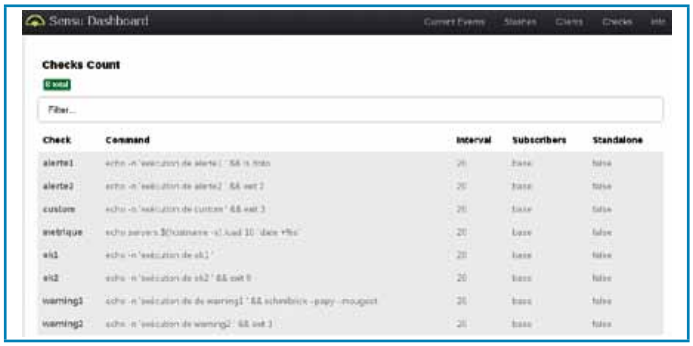


Fig. 5

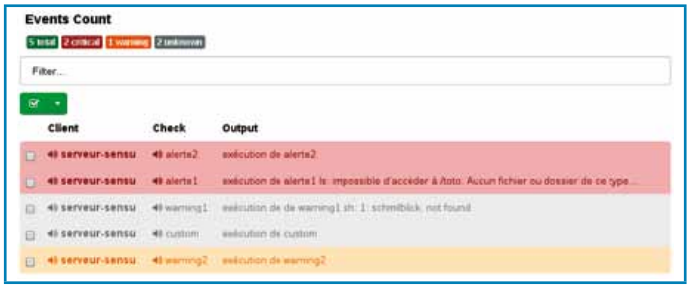


Fig. 6

3.2 Interactions dans la dashboard

À l'instar des clients sur la page ***Clients***, sur la page ***Events*** vous pouvez afficher en détails un événement en cliquant dessus, et le supprimer ou le désactiver (Fig. 7). Les interactions avec les événements sont très basiques depuis la dashboard. Nous verrons plus loin que c'est vraiment dans les handlers que réside toute la logique de notification.

Fig. 7

L'icône verte en haut à gauche vous permet de sélectionner tout ou partie des checks, et/ou d'effectuer des actions par lots sur ceux sélectionnés (Fig. 8). Les clients et checks désactivés apparaissent dans la section ***Stashes*** de la dashboard (Fig. 9) et peuvent être rétablis depuis cette page. Nous verrons plus loin comment gérer les stashes de manière plus dynamique, et qu'ils ne servent pas qu'à désactiver des hôtes ou checks.

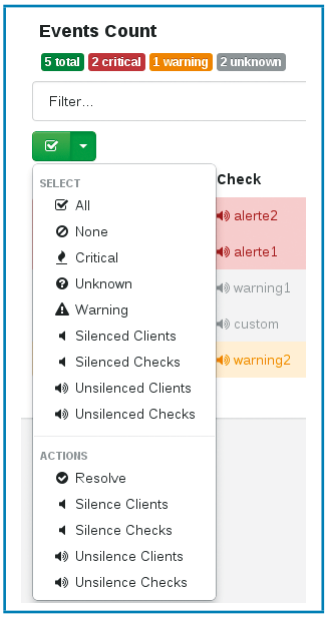


Fig. 8

3.3 Écrire vos checks

Les checks précédents sont quelque peu limités. Vous pourrez écrire vos propres checks dans le langage de scripts de votre choix, et passer ce script en paramètre de l'option **command** de la définition d'un check. Un framework a été créé par l'équipe de Sensu **[FRAMEWORK]** pour développer plus facilement vos checks. Il est écrit en Ruby, mais se veut accessible, même pour les personnes nouvelles au langage, et vous permet d'écrire rapidement des checks classiques ou de type **metric**.

3.4 Utiliser les scripts existants

Plusieurs checks sont disponibles sur un dépôt Git du projet **[PLUGINS]**, écrits à l'aide du framework. Checks pour tester les processus d'un système **[PLUGINS2]**, de type **metric** pour extraire des logs d'Apache au format Graphite, ou encore lire les données d'un serveur Graphite pour surveiller une métrique. Ils répondent à des besoins courants, vous évitent de ré-inventer la roue et peuvent servir d'exemples pour développer vos propres checks.

Pour utiliser un de ces checks, commencez par récupérer un script dans le dossier **/etc/sensu/plugins**, comme par exemple le check **check_dir_count.rb** :

```
$ sudo  wget https://raw.github.com/sensu/sensu-community-plugins/master/
plugins/files/check_dir_count.rb -O /etc/sensu/plugins/check_dir_count.rb
```

Donnez-lui les droits d'exécution :

```
$ sudo chmod a+x /etc/sensu/plugins/check_dir_count.rb
```

Ce script Ruby teste le nombre de fichiers d'un répertoire. Avec les options **-w** et **-c**, on peut préciser des seuils d'alerte et de warning :

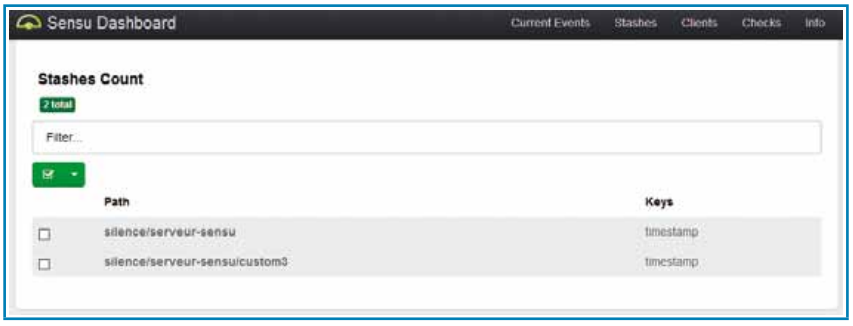


Fig. 9

```
$ /etc/sensu/plugins/check_dir_count.rb -w 10 -c 20 -d /etc/sensu/
DirCount OK: /etc/sensu/ has 8 files
$  /etc/sensu/plugins/check_dir_count.rb -w 5 -c 10 -d /etc/sensu/
DirCount WARNING: /etc/sensu/ has 7 files (threshold: 5)
$ /etc/sensu/plugins/check_dir_count.rb -w 1 -c 5 -d /etc/sensu/
DirCount CRITICAL: /etc/sensu/ has 7 files (threshold: 5)
```

On voit donc que ce script a une double utilité : retourner une métrique, et envoyer un warning ou une alerte en cas de problème. On peut donc créer le check **/etc/sensu/conf.d/check_dir_sensu.json** suivant, de type **metric** :

```
{
  "checks": {
    "check-dir-sensu": {
      "type": "metric",
      "command": "/etc/sensu/plugins/check_dir_count.rb -w 5 -c 10 -d /etc/sensu/",
      "subscribers": ["base"],
      "interval": 10
    }
  }
}
```

Redémarrez le serveur, qui va notifier au client de faire ce check, client qui va exécuter le **script check-dir-sensu** avec les paramètres fournis.

3.5 Substitution de commande

Vous pouvez utiliser des valeurs particulières à un client, dans la commande d'un check. Pour revenir à notre exemple **check-dir-sensu**, imaginons que nous voulions donner des seuils différents pour chaque machine monitorée. On peut créer des paramètres personnalisés du nom de notre choix dans la configuration d'un client (à condition de les nommer différemment d'un paramètre existant), par exemple dans notre fichier **conf.d/client.json** :

```
{
      "client": {
        "name": "serveur-sensu",
        "address": "192.168.0.11",
        "subscriptions": [ "base","cron"],
        "param-dir-sensu": {
                "warning": 8,
                "critical": 10
        }
      }
}
```

On a créé ici un paramètre **param-dir-sensu**, qui est un hash (on aurait très bien pu définir un nouveau paramètre classique **''option'':''valeur''**), qui contient deux sous-paramètres **warning** et **critical**. Redémarrez le client Sensu sur **serveur-sensu** pour qu'il prenne en compte sa nouvelle configuration.

Pour utiliser ces options dans la commande du check **/etc/sensu/conf.d/check_dir_sensu.json** précédent, il faut encadrer la valeur de ces options par trois double-points :

```
{
  "checks": {
    "check-dir-sensu": {
      "type": "metric",
      "command": "/etc/sensu/plugins/check_dir_count.rb -w :::param-dir-
sensu.warning::: -c :::param-dir-sensu.critical::: -d /etc/sensu/",
      "subscribers": ["base"],
      "interval": 10
    }
  }
}
```

Redémarrez le serveur, les alertes devraient apparaître dans la dashboard pour des seuils aussi bas.

3.6 Utiliser les checks Nagios

Si vous avez déjà utilisé Nagios, le script précédent doit vous être familier. Les développeurs de Sensu sont restés sur ce format de message des scripts de check à la Nagios, car c'est un format simple, facilement parsable, et pour justement pouvoir réutiliser les scripts Nagios existants. On peut donc installer un paquet contenant des scripts Nagios de base pour ré-utiliser un de ses checks :

```
$ sudo aptitude install -y nagios-plugins-basic
```

En particulier, il contient le binaire **/usr/lib/nagios/plugins/check_procs**, qui teste l'existence de processus. Par exemple, si on veut tester qu'un démon **cron** tourne :

```
$ /usr/lib/nagios/plugins/check_procs -c 1:1 -a '/usr/sbin/cron' -u
root
PROCS OK: 1 processus avec args '/usr/sbin/cron', UID = 0 (root)
```

Et si on coupe **cron**, il retourne en erreur :

```
$ sudo /etc/init.d/cron stop
$ /usr/lib/nagios/plugins/check_procs -c 1:1 -a '/usr/sbin/cron' -u root
PROCS CRITIQUE: 0 processus avec args '/usr/sbin/cron', UID = 0 (root)
```

Ne reste plus qu'à définir un nouveau check **conf.d/check_cron.json** :

```
{
    "checks": {
       "check-cron": {
         "command": "/usr/lib/nagios/plugins/check_procs -c 1:1 -a
'/usr/sbin/cron' -u root",
         "interval": 20,
         "subscribers": [ "cron" ]
       }
    }
}
```

J'ai attribué ce check à l'exchange **cron**, auquel notre client est déjà abonné via sa configuration **client.json**. Redémarrez Sensu, notre client va exécuter ce check et une erreur devrait apparaître dans la dashboard.

En plus d'être peu fonctionnels, l'exécution de nos checks n'a pour seule notification qu'un affichage dans la dashboard. C'est aux handlers que revient la charge d'effectuer des notifications plus avancées.

4 Handlers

Un handler entre en action à la réception d'un événement généré par un check : exécution d'un script, envoi d'un e-mail, etc. Il existe plusieurs types de handlers suivant l'action à exécuter : **tcp**, **udp**, **amqp**. Dans le cas d'un check classique, le serveur Sensu passe le résultat à un handler sur son entrée standard, seulement si le check s'est terminé par un code différent de 0. Pour un check de type **metric**, la mesure est logiquement toujours transférée par le serveur au handler associé.

4.1 Le type TCP

Nous avons déjà vu comment assigner des handlers dans la configuration d'un check, et que tous nos checks sont déjà assignés au handler par défaut **default**, qui ne fait rien. Nous allons créer un handler de type TCP, qui reçoit un événement d'un check de type **metric** pour alimenter un (hypothétique) serveur Graphite.

Créez le fichier de handler **conf.d/handler_graphite.json** suivant :

```
{
  "handlers": {
    "graphite": {
      "type": "tcp",
      "socket": {
        "host": "127.0.0.1",
        "port": 2003
      }
    }
  }
}
```

Notre handler est simple, il envoie tout message reçu vers un socket TCP local. Je l'ai appelé **graphite**, car on peut l'utiliser pour envoyer les métriques collectées vers un serveur Graphite. Pour cela, modifiez le check **metrique** du fichier **conf.d/check_exemples.json** pour lui assigner ce handler, à la place du handler **default** :

```
...
    "metrique": {
      "type": "metric",
      "handlers": ["graphite"],
      "command": "echo servers.$(hostname -s).load 10 `date +%s`",
...
```

Redémarrez Sensu, et utilisons **netcat** pour simuler un serveur Graphite écoutant sur le port local 2003, pour pouvoir visualiser les messages envoyés par le handler sur le socket :

```
$ sudo aptitude -y install netcat
$ while [ 1 ]; do nc -l -s 127.0.0.1 -p 2003; echo ; done
{"client":{"name":"serveur-sensu","address":"192.168.0.11","subscriptions":
["base","cron"],"param-dir-sensu":{"warning":5,"critical":10},"timestamp":1-
393950566},"check":{"type":"metric","command":"echo servers.$(hostname -s).load
10 `date +%s`","subscribers":["base"],"handlers":["graphite"],"interval":
20,"name":"metrique","issued":1393950576,"executed":1393950576,"output":"serve
rs.serveur-sensu.load 10 1393950576\n","status":0,"duration":0.04,"history":["0"
,"0","0","0","0","0","0","0","0","0","0","0","0","0","0","0","0","0","0","0"
]},"occurrences":1}
```

4.2 Mutator

Les messages sont envoyés au format JSON par défaut, format lisible et facilement parsable. Or, le message envoyé par le handler vers ce socket contient beaucoup d'informations. Trop par rapport au message généré par le check qui était déjà au format attendu par **Graphite**.

Pour manipuler le message avant son envoi par le handler, on utilise un **mutator** pour modifier son format. On indique le mutator à utiliser dans la configuration du handler. Ici, nous allons utiliser le mutator prédéfini **only_check_output**, qui renvoie au format brut au handler le message généré par le check comme reçu par le serveur, sans le formatage JSON :

```
{
  "handlers": {
    "graphite": {
      "type": "tcp",
      "socket": {
        "host": "127.0.0.1",
        "port": 2003
      },
      "mutator": "only_check_output"
    }
  }
}
```

Redémarrez Sensu, le format attendu devrait être envoyé au socket :

```
servers.serveur-sensu.load 10 1393928531
servers.serveur-sensu.load 10 1393928551
servers.serveur-sensu.load 10 1393928571
```

4.3 Le type Pipe

Le type de handler TCP est pratique pour l'envoi de métriques. Il existe également des handlers pour utiliser un socket UDP ou un exchange **AMQP** **[HANDLERS]**, en utilisant le type éponyme. Mais vous utiliserez plutôt un handler de type **pipe**. Avec **pipe**, le message reçu par le serveur est passé sur l'entrée d'une commande définie dans le handler. C'est typiquement le genre de handler pour traiter un message d'erreur renvoyé par un check, et exécuter une notification. Par exemple, pour envoyer un e-mail en cas d'erreur sur un check, on peut créer un handler **email conf.d/handler_email.json** :

```
{
      "handlers": {
        "email": {
          "type": "pipe",
          "command": "mail -s 'sensu alert' root@localhost",
          "mutator": "only_check_output"
        }
      }
}
```

Le message reçu par le serveur au format JSON est passé sur l'entrée standard de la commande **mail**, message qui sera

donc le corps de l'e-mail envoyé. Pour attribuer ce handler, par exemple au **check-dir-sensu** :

```
...
{
  "checks": {
    "check-dir-sensu": {
      "type": "metric",
      "handlers": ["email","graphite"],
...
```

Redémarrez Sensu, vous devriez recevoir les e-mails :

```
$ mail
Mail version 8.1.2 01/15/2001.  Type ? for help.
"/var/mail/user": 4 messages 4 new
>N  1 sensu@serveur-sensu       Tue Mar 04 17:39   16/510   sensu alert
 N  2 sensu@serveur-sensu       Tue Mar 04 17:39   16/510   sensu alert
 N  3 sensu@serveur-sensu       Tue Mar 04 17:39   16/510   sensu alert
 N  4 sensu@serveur-sensu       Tue Mar 04 17:39   16/510   sensu alert
&
Message 1:
From sensu@serveur-sensu Tue Mar 04 17:39:10 2014
Envelope-to: root@localhost
Delivery-date: Tue, 04 Mar 2014 17:39:10 +0100
To: root@localhost
Subject: sensu alert
From: Sensu Monitoring Framework <sensu@serveur-sensu>
Date: Tue, 04 Mar 2014 17:39:10 +0100

DirCount OK: /etc/sensu/ has 7 files
```

4.4 Sévérité

On a assigné au check le handler **graphite** au passage. Si vous redémarrez le serveur et modifiez un peu les seuils, vous vous apercevrez que les handlers **email** et **graphite** sont lancés par le serveur plus souvent que ce que l'on pourrait souhaiter, ce qui est normal par rapport au comportement par défaut énoncé précédemment : un check de type **metric** a toujours ses handlers exécutés par le serveur ; les handlers **email** et **graphite** sont donc toujours exécutés, que le check réussisse ou non.

En l'état, le handler **graphite** est donc toujours exécuté, ce qui nous convient. Mais nous voudrions que le handler **email** s'exécute uniquement si le check n'a pas réussi (code retour différent de ok/0). Il faut modifier le handler **email**, en utilisant le paramètre **severities** pour cela :

```
...
"command": "mail -s 'sensu alert' root@localhost",
          "severities": ["warning", "critical", "unknown"],
          "mutator": "only_check_output"
...
```

Redémarrez Sensu, vous ne devriez plus recevoir d'e-mails de type OK.

4.5 Mutator 2, le retour

Autre souci : le handler **graphite** ne retourne pas le bon format en cas de réussite de ce deuxième check :

```
$ while [ 1 ]; do nc -l -s 127.0.0.1 -p 2003; echo ;done
servers.serveur-sensu.load 10 1393951394
DirCount OK: /etc/sensu/ has 7 files
DirCount OK: /etc/sensu/ has 7 files
servers.serveur-sensu.load 10 1393951414
DirCount OK: /etc/sensu/ has 7 files
DirCount OK: /etc/sensu/ has 7 files
```

Nous avons utilisé le mutator par défaut **only_check_output** dans le handler **graphite**, mais nous pouvons écrire nos propres mutators. Notre mutator **graphite** va exécuter un script Ruby **mutators/graphite.rb**, qui va parser l'output du message s'il provient du check **check-dir-sensu**, et le mettre au bon format :

```
#!/usr/bin/env ruby

require 'rubygems'
require 'json'

#On récupére le message complet au format JSON
event = JSON.parse(STDIN.read, :symbolize_names => true)
# dans une variable output on ne récupère que le message au format brut
output =  event[:check][:output]

nom = event[:check][:name]
#Si le nom du check est check-dir-sensu
if nom == 'check-dir-sensu'
        #on parse la variable output pour récupérer le nombre de fichiers du dossier
       metric = output.split(' ')[4]
       date = event[:check][:executed]
       client = event[:client][:name]
        #et on forge une nouvelle chaîne de caractères adéquate output
       output = "servers.#{client}.dir  #{metric} #{date}"
end

puts output
```

On donne les droits d'exécution au script :

```
$ sudo chmod a+x /etc/sensu/mutators/graphite.rb
```

On peut ensuite définir ce mutator dans le fichier **conf.d/mutator_graphite.json** dans une section **mutators**, en précisant le script à exécuter avec le paramètre **command** :

```
{
  "mutators": {
    "graphite": {
      "command": "/etc/sensu/mutators/graphite.rb"
    }
  }
}
```

Il ne reste plus qu'à modifier la valeur de l'option **mutator** du fichier **conf.d/handler_graphite.json** du handler **graphite** :

```
{
  "handlers": {
    "graphite": {
...
      "mutator": "graphite" ,
...
```

Redémarrez Sensu, les métriques devraient être toutes au bon format dorénavant :

```
$  while [ 1 ]; do nc -l -s 127.0.0.1 -p 2003; echo ;done
servers.serveur-sensu.load 10 1393951620
servers.serveur-sensu.dir  7 1393951622
servers.serveur-sensu.dir  7 1393951632
servers.serveur-sensu.load 10 1393951640
servers.serveur-sensu.dir  7 1393951642
```

4.6 Handlers set

Vous pouvez aussi définir un ensemble de **handlers** (*handlers set*), dans une seule et même définition de handler. Imaginons que par défaut, vous souhaitiez attribuer plusieurs handlers de base pour tous les checks : l'envoi d'un e-mail et l'envoi de métriques vers graphite, grâce aux handlers précédents. Il vous suffirait de redéfinir un **handlers set**, c'est-à-dire un groupe de handlers, et d'attribuer ce groupe à vos checks. Nous avons déjà les deux handlers **email** et **graphite**. Et nous avons un handler nommé **default**, qui est attribué à tous les checks par défaut. Il nous suffit de redéfinir ce handler **default** comme un handlers set, dans le fichier **handler_default.json** :

```
{
       "handlers": {
         "default": {
          "type": "set",
          "handlers": [ "email", "graphite" ]
         }
       }
}
```

Note

Créer des checks qui font à la fois office de collecteur de métriques et déclencheur d'alertes n'est pas conseillé suivant la documentation de Sensu. Les manipulations précédentes ont plus un intérêt pédagogique. En pratique, un check de type métrique doit toujours réussir et sortir avec un code de retour 0, et il vaudra mieux avoir deux checks distincts qui tournent sur chaque client, pour éviter d'avoir des handlers et mutators superflus, dont le travail peut surcharger le serveur.

4.7 Écrire ses handlers

Tout comme pour les checks, vous pouvez écrire vos propres handlers. Le framework sensu-plugin **[FRAMEWORK]** vous facilitera à nouveau la tâche sous Ruby. Vous pourrez facilement accéder à toutes les valeurs dans le message du check reçu par le handler au format JSON, ainsi que d'autres valeurs définies par vos soins. À la différence des checks, les paramètres personnalisés dans un fichier de client ne sont pas disponibles comme option du paramètre **command** de la configuration d'un handler (mais restent accessibles dans le script appelé par le handler).

Comme les checks, vous trouverez sur le dépôt communautaire du projet **[PLUGINS]** un dossier contenant des exemples de handlers **[HANDLERS2]**, aussi bien pour gérer des métriques que des notifications. Par exemple, vous trouverez un handler d'envoi d'e-mails **[MAILER]** plus avancé que celui vu précédemment, qui prend comme configuration un fichier de valeurs personnalisées **[MAILER2]**. Il utilise également un paramètre **notification**, comme valeur du sujet de l'e-mail, que l'on définit dans la configuration d'un check pour avoir un message de description plus lisible en guise d'objet du mail. Ce champ n'est pas une valeur officielle Sensu de configuration d'un check, mais un paramètre personnalisé que des utilisateurs et développeurs de handlers Sensu ont pris comme convention pour un tel usage :

```
{
     "checks": {
        "check-cron": {
          "notification" : "Check de vérification du processus cron",
...
```

5 Pour aller plus loin

5.1 Ajouter un client supplémentaire

Pour l'instant, nous n'avons utilisé qu'une machine **serveur-sensu** en guise de client et serveur Sensu. Nous allons ajouter une nouvelle machine à monitorer par le serveur. Nous l'appellerons de manière très originale **client-sensu**, et installerons la partie cliente de Sensu. Pour cela, reportez-vous à la section d'installation en début d'article. Redis et RabbitMQ sont inutiles sur un client. Contentez-vous d'installer Sensu suivant la section 2.2. N'oubliez pas de récupérer depuis le serveur les certificats utiles à la connexion à RabbitMQ, et d'éditer le fichier **/etc/default/sensu** Ruby embarqué. Le fichier de **/etc/sensu/conf.d/rabbitmq.json** de configuration de la connexion à RabbitMQ est quasi identique :

```
{
  "rabbitmq": {
    "ssl": {
      "cert_chain_file": "/etc/sensu/ssl/cert.pem",
      "private_key_file": "/etc/sensu/ssl/key.pem"
    },
    "host": "192.168.0.11",
    "port": 5671,
    "vhost": "/sensu",
    "user": "sensu",
    "password": "mypass"
  }
}
```

Ainsi que le fichier client **/etc/sensu/conf.d/client.json** :

```
{
    "client": {
        "name": "client-sensu",
        "address": "192.168.0.12",
       "subscriptions": ["base"],
          "param-dir-sensu": {
              "warning": 3,
              "critical": 5
          }
     }
}
```

Les services **sensu-server**, **sensu-api** et **sensu-dashboard** sont bien sûr inutiles sur notre client. Activez au démarrage et démarrez seulement le service client :

```
$ sudo update-rc.d sensu-client defaults
$ sudo /etc/init.d/sensu-client start
```

Vous devriez voir apparaître des alertes similaires à celles renvoyées par le service client s'exécutant sur le serveur Sensu (Fig. 10). Notre client s'est abonné uniquement à l'exchange **base**. Notre nouveau client n'exécute pas le check **check-cron**, car il n'est pas abonné à l'exchange **cron**.

Sur la figure 10, on peut voir que le check **check-dir-sensu** a un souci sur notre nouveau client et affiche un warning : il n'arrive pas à s'exécuter, car il ne trouve pas le script **check_dir_count.rb**. Le client a bien récupéré les ordres et différents checks à exécuter depuis le serveur, via RabbitMQ. Seulement, le serveur ne lui transfère pas le script, il doit être physiquement présent sur chacun des clients l'exécutant. Qu'à cela ne tienne, il suffit d'installer le script sur le nouveau client, comme nous l'avons fait lors de la création initiale du check sur le serveur :

```
$ sudo wget https://raw.github.com/sensu/sensu-community-plugins/master/plugins/files/check_dir_count.rb -O /etc/sensu/plugins/check_dir_count.rb
$ sudo chmod a+x /etc/sensu/plugins/check_dir_count.rb
```

5.2 Checks autonomes

Nous avons vu la capacité de découverte automatique des clients par Sensu, qui permet d'éviter les redémarrages de serveur. Les checks autonomes vont vous

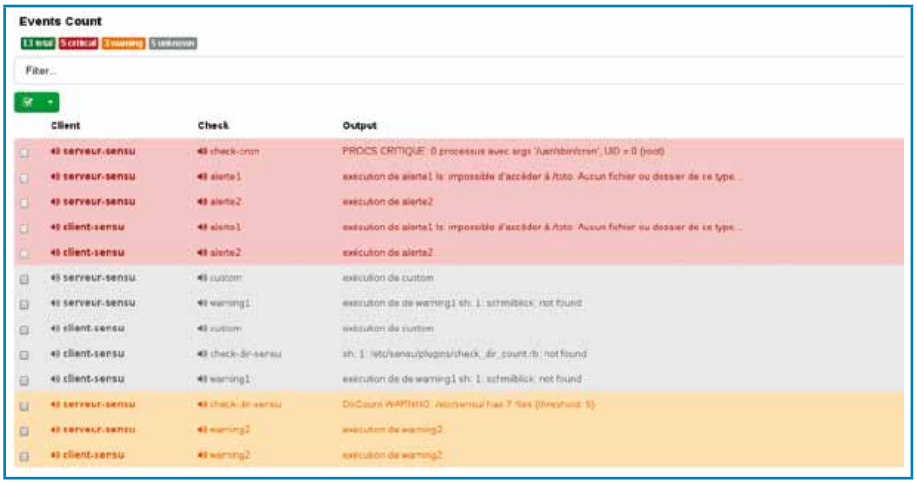


Fig. 10

permettre de faire exécuter des checks par des clients, en les configurant et les exécutant depuis les clients, tout en envoyant le résultat au serveur. Ainsi, vous éviterez au serveur d'avoir de nombreux checks à gérer et faire exécuter par les clients. Mais surtout, vous pourrez ajouter des nouveaux checks sur les clients, sans changer la configuration du serveur et avoir à le redémarrer.

Par exemple, pour faire exécuter par **client-sensu** un check similaire au **check-dir-sensu** précédent, mais de manière *standalone*, on crée le fichier **conf.d/check_dir_sensu_standalone.json** *sur* **client-sensu** :

```
{
  "checks": {
    "check-dir-sensu-standalone": {
      "type": "metric",
      "handlers": ["default"],
      "command": "/etc/sensu/plugins/check_dir_count.rb -w :::param-dir-sensu.warning::: -c :::param-dir-sensu.critical::: -d /etc/sensu/",
      "interval": 10,
      "standalone": true
    }
  }
}
```

On a défini un check nommé **check-dir-sensu-standalone**, et on indique qu'il est autonome par le paramètre **standalone** à **true**. On ré-utilise ici le script **check_dir_count.rb** copié précédemment. Pour un check autonome, le paramètre **subscribers** est inutile, vu qu'un check standalone est obligatoirement exécuté par le client sur lequel il est défini.

Le paramètre **handlers** est par contre toujours pris en compte par le serveur (ici, je l'ai explicitement indiqué, même si c'était inutile car on utilise le handler par défaut **default**) : le processus d'exécution du check reste le même, il envoie le retour de la commande au serveur qui enclenche le(s) handler(s) associé(s). Handler, associé dans la configuration sur le client, et qui doit toujours être défini sur le serveur. On redémarre le service client Sensu seulement sur **client-sensu**, et notre serveur va recevoir correctement les messages de ce check autonome (Fig. 11).

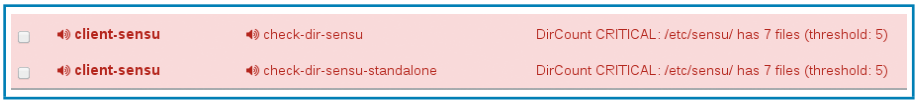


Fig. 11

PROFESSIONNELS DES TICE, COLLECTIVITÉS, ÉCOLES D'INGÉNIEURS, UNIVERSITÉS, R & D, ENSEIGNANTS, ...

GNU LINUX MAGAZINE / FRANCE

VOUS PROPOSE 2 NOUVEAUX SERVICES !

1 VOUS SOUHAITEZ LIRE GNU/LINUX MAGAZINE EN VERSION PDF ?

PDF

LINUX MAGAZINE / FRANCE 11 NOS

VOICI LES ABONNEMENTS PDF COLLECTIFS !

Ce service vous permet d'abonner votre structure (écoles, collectivités, entreprises, etc.) à l'édition PDF de nos magazines afin d'en profiter dès leur parution chez les marchands de journaux.

Sans DRM, téléchargez simplement, lisez et annotez vos eBooks sur votre PC, smartphone ou liseuse électronique.

2 VOUS SOUHAITEZ RETROUVER ET CONSULTER LES ARTICLES DE GLMF ?

BASE DOCUMENTAIRE LINUX MAGAZINE / FRANCE + HORS-SÉRIES

VOICI LA BASE DOCUMENTAIRE !

L'accès à la base documentaire en ligne de GNU/Linux Magazine et de ses Guides vous permettra d'effectuer des recherches dans la majorité des articles parus, qui seront disponibles 6 mois après leur parution en magazine. Vous pourrez ainsi effectuer des recherches sur les articles indexés, copier les codes, etc.

La consultation s'effectue sur notre nouveau service **connect.ed-diamond.com** qui est en place depuis janvier 2014 (n'hésitez pas à le visiter !)...

connect.ed-diamond.com

N'hésitez pas à consulter notre offre sur **boutique.ed-diamond.com**,

« Base Documentaire TOTALE » à 399 € HT/an

(5 connexions comprises) pour profiter de la base documentaire de l'ensemble des parutions des Éditions Diamond !

BESOIN DE RENSEIGNEMENTS SUPPLÉMENTAIRES OU D'UN DEVIS SUR MESURE ?

N'hésitez pas à envoyer un e-mail à **aboprof@ed-diamond.com** ou à téléphoner au **+33 (0)3 67 10 00 27**

5.3 Socket client et modèle Push

Au démarrage, le service client de Sensu démarre également un socket UDP et TCP sur le port 3030. Cela ouvre la voie à de nombreuses interactions depuis un client, comme par exemple la possibilité d'utiliser des checks en mode Push. Ainsi, on peut utiliser **netcat** pour envoyer un événement au format JSON avec la syntaxe adéquate (Fig. 12) :

```
$ echo '{
 "handlers": ["email"],
 "name": "email-push-test",
 "output": "Ceci est un test",
 "status": 2
}' | nc -w1 127.0.0.1 3030
```

Ce socket offre beaucoup d'opportunités, même si pour l'instant un modèle de sécurité doit être mis en place pour le protéger.

Fig. 12

5.4 Contrôle des handlers

Nous avons déjà vu les stashes lors de la désactivation temporaire de clients et/ou checks depuis la dashboard. Pour une désactivation continue sur une plage horaire donnée, vous pouvez désactiver le ou les handler(s) assigné(s) à un check, standalone ou non, avec le paramètre **subdue**, dans la configuration du check :

```
{
  "checks": {
    "check-dir-sensu": {
   ...
        "interval": 10,
        "subdue" : {
        "begin": "1AM UTC",
        "end": "12PM UTC",
        "days": ["saturday", "sunday"]
         },
         "occurrences": 10,
...
```

Ici, j'ai désactivé l'exécution des handlers assignés au check **check-dir-sensu**, toute la journée des samedis et dimanches. J'ai aussi utilisé le paramètre **occurrences** : pour les plages horaires en dehors de **subdue**, les handlers ne seront exécutés qu'après 10 retours en erreur du check.

La définition des checks propose aussi la section **dependencies**, pour gérer les dépendances entre checks : si un ou plusieurs checks listés par ce paramètre sortent en erreur, les handlers du check courant ne seront pas exécutés. Cela permet d'éviter de recevoir des tonnes d'e-mails, alertes, pour des problèmes qui sont souvent liés.

5.5 API

Sensu possède également une API très simple pour interagir avec les données persistantes stockées dans la base Redis **[API]**.

Pour récupérer la liste des événements en cours, un simple **curl** suffit :

```
$ sudo aptitude install -y curl
$ curl  -u admin:secret http://localhost:4567/events  #sortie non affichée car trop longue
```

Ou les événements d'un client en particulier :

```
$ curl  -u admin:secret http://localhost:4567/events/client-sensu   #sortie trop longue
```

Ou liés à un check en particulier :

```
$ curl  -u admin:secret http://localhost:4567/events/client-sensu/warning1
{"output":"exécution de de warning1 sh: 1: schmilblick: not found\n","status":127,"issued":1393953635,"handlers":["default"],"flapping":false,"occurrences":26,"client":"client-sensu","check":"warning1"}
$ curl -u admin:secret  http://localhost:4567/events/client-sensu/check-dir-sensu-standalone
{"output":"DirCount CRITICAL: /etc/sensu/ has 7 files (threshold: 2)\n","status":2,"issued":1393954408,"handlers":["default"],"flapping":false,"occurrences":4,"client":"client-sensu","check":"check-dir-sensu-standalone"}
```

5.5.1 Stashes

Nous avons vu les Stashes précédemment, que l'on peut créer depuis la dashboard. Un stash doit avoir un format particulier pour être pris en compte par les handlers. Si vous avez créé un stash depuis la dashboard, on peut utiliser l'API pour étudier son format :

```
$ curl -u admin:secret  http://localhost:4567/stashes
[{"path":"silence/client-sensu","content":{"timestamp":1393955295},"expire":-1}]
```

Le **path** correspond au chemin pour récupérer le contenu d'un stash en particulier :

```
$ curl  -u admin:secret http://localhost:4567/stashes/silence/client-sensu
{"timestamp":1393955295}
```

Vous pouvez créer vos propres stashes, qui sont en fait des fichiers de données JSON utilisables par vos handlers :

```
$ curl -u admin:secret -XPOST  http://localhost:4567/stashes/data -d '{"var1":"toto","var2":30}'
{"path":"data"}
$ curl  -u admin:secret http://localhost:4567/stashes/data
{"var1":"toto","var2":30}
```

Cette fonctionnalité est encore récente et la documentation officielle reste laconique sur le sujet.

5.6 Interface en ligne de commandes

L'API de Sensu est tellement simple que l'on pourrait se servir uniquement de Curl pour interagir avec elle en ligne de commandes. Un utilisateur de Sensu a néanmoins développé une gemme Ruby **sensu-cli**, pour nous simplifier le travail en mode CLI :

```
$ sudo /opt/sensu/embedded/bin/gem install sensu-cli
$ mdkir ~/.sensu
```

Créez le fichier **~/.sensu/settings.rb** pour spécifier les paramètres de connexion à l'API :

```
host  "127.0.0.1"
port  "4567"
ssl   false
user "admin"
password "secret"
```

Vous pouvez ensuite utiliser l'outil, par exemple pour lister les clients :

```
$ sensu client list
-------
name:  serveur-sensu
address:  192.168.0.11
subscriptions:  ["base", "cron"]
param-dir-sensu:  {"warning"=>9, "critical"=>10}
timestamp:  1393955662
-------
name:  client-sensu
address:  192.168.0.12
subscriptions:  ["base"]
param-dir-sensu:  {"warning"=>30, "critical"=>50}
timestamp:  1393955662
2 total items
```

Ou travailler avec les stashes :

```
$ sensu stash
** Stash Commands **
sensu-cli stash list (OPTIONS)
sensu-cli stash show STASHPATH
sensu-cli stash delete STASHPATH
sensu-cli stash create PATH
$ sensu stash list
-------
path:  silence/client-sensu
content:  {"timestamp"=>1393955295}
expire:  -1
-------
path:  data
content:  {"var1"=>"toto", "var2"=>30}
expire:  -1
2 total items
$ sensu stash delete silence/client-sensu
The item was successfully deleted.
```

Il existe de nombreuses sous-commandes, visibles avec l'option **--help**, ou encore sur le dépôt officiel de la gemme **[SENSU-CLI]**.

Conclusion

Comparé à Nagios/Icinga, Sensu se veut plus simple à mettre en place pour monitorer de nombreuses machines, mais est encore loin derrière pour tout ce qui est escalade de notifications et politique de notification avancée. Shinken rassemble le meilleur des deux mondes, mais il n'a pas la reconnaissance qu'il mérite, malgré le travail de son auteur Jean Gabès. Est-ce que cela est dû à son héritage Nagios, le fait que ce soit un projet développé par un français, ou les deux ? Sensu, de son côté, a très rapidement gagné en popularité. Pour beaucoup parce que c'est un projet américano-ruby et surtout, parce qu'il a été adopté par beaucoup d'administrateurs système qui voulaient se démarquer complètement de Nagios. Cet engouement va accélérer son développement et lui ajouter des fonctionnalités pour combler son retard.

Si vous êtes nouveau dans le monde du monitoring, lancez-vous dans Sensu ! Pour les anciens, utilisateurs de Nagios, vous pourrez toujours installer en test une infrastructure de supervision, ré-utiliser vos checks Nagios, pour faire une migration progressive et en douceur. Dans tous les cas, des alternatives à Nagios, Zabbix, etc., avec une philosophie différente, existent, et j'espère que cet article vous a démontré que Sensu en est une viable. ■

Références

[SENSU] http://sensuapp.org

[JSON] http://sensuapp.org/docs/0.12/events

[INSTALLATION] http://sensuapp.org/docs/0.12/guide

[OMNIBUS] https://github.com/opscode/omnibus-ruby

[FRAMEWORK] https://github.com/sensu/sensu-plugin

[PLUGINS] https://github.com/sensu/sensu-community-plugins

[PLUGINS2] http://sensuapp.org/docs/0.12/adding_a_check

[HANDLERS] http://sensuapp.org/docs/0.12/handlers

[HANDLERS2] https://github.com/sensu/sensu-community-plugins/tree/master/handlers

[MAILER] https://github.com/sensu/sensu-community-plugins/blob/master/handlers/notification/mailer.rb

[MAILER2] https://github.com/sensu/sensu-community-plugins/blob/master/handlers/notification/mailer.json

[API] http://sensuapp.org/docs/0.12/api

[SENSU-CLI] https://github.com/agent462/sensu-cli

YAML ET PYTHON

par Tristan Colombo

On a toujours besoin d'utiliser une structure, un format particulier, pour représenter des données dans un fichier. Cela permet d'accéder plus rapidement aux informations sans avoir à passer des heures en développement. Seulement, certains formats sont plus simples que d'autres...

Tout le monde connaît le XML et son écriture particulièrement verbeuse basée sur des balises ouvrantes et fermantes. Peu de gens utilisent correctement ce langage qui nécessite normalement un fichier décrivant la structure de données. Ce fichier, appelé DTD, pour *Document Type Definition*, soit Définition de Type de Document, est en quelque sorte la grammaire qui permettra de valider un document : soit il respecte la DTD et il peut être traité, soit il est inutile d'aller plus loin et on peut déclencher une erreur.

Le YAML (prononcer yamel) est un XML allégé dont l'écriture sera plus lisible par un être humain. Attention, je ne dis pas qu'il faut enterrer le XML ! Seulement, dans certains cas, il est inutile de s'encombrer d'une telle lourdeur et l'on peut alors se tourner vers des formats plus légers tels que le YAML.

Dans cet article, nous verrons quelle est la syntaxe du YAML qui nous permettra de créer des fichiers lisibles et utilisables depuis un programme Python.

1 YAML

Pour faire simple, le YAML est une sorte de dictionnaire (au sens Python), mais sous forme de fichier. La syntaxe s'approche également énormément du JSON... Et c'est un peu normal, puisque nous sommes ici en présence d'une représentation par sérialisation des données.

Voici les éléments indispensables permettant de créer un document YAML :

- **---** : séparateur de documents. Permet de créer plusieurs documents à l'intérieur d'un seul fichier.
- **#** : commentaire en ligne. Tous les caractères qui suivent un **#** ne seront pas évalués.

```
--- # Premier document
...
--- # Deuxième document
...
```

- **-** : délimiteur d'item de liste. Une ligne commençant par un tiret correspond à un item dans une liste.

```
--- # Document
- Premier item
- Deuxième item
```

- **[]** : définition d'une liste en ligne. Permet de définir une liste sur une seule ligne.

```
--- # Document
[Premier item, Deuxième item]
```

- **:** : définition d'un tableau associatif (dictionnaire). On peut associer des valeurs à des clés. C'est la syntaxe qui va permettre d'accéder le plus simplement aux éléments du fichier YAML.

```
--- # Document
 cle_1: valeur_1
 cle_2: valeur_2
```

- **{}** : définition d'un tableau associatif en ligne.

```
--- # Document
 {cle_1: valeur_1, cle_2: valeur_2}
```

- **chaîne de caractères** : une chaîne de caractères. Pas de syntaxe particulière pour indiquer une donnée de type chaîne de caractères sur une ligne.

```
--- # Document
Ceci est un texte qui sera lu
```

- **|** : une chaîne de caractères multiligne. Cette syntaxe permet d'indiquer une donnée sur plusieurs lignes.

```
--- # Document
 nom: Tux
 adresse: |
    666 route Emacs
    29812 GNUCity
```

- **>** : chaîne de caractères multiligne avec remplacement du retour à la ligne par un espace.

```
--- # Document
  texte: >
    Ceci est une
    même phrase sur
    plusieurs lignes
```

- **&** et ***** : permettent de faire des références à l'intérieur d'un même document. Le caractère **&** suivi d'un label indique une ancre à laquelle on pourra faire référence en utilisant le même label, précédé cette fois par le caractère *****. Attention, ceci fonctionne à l'intérieur d'un même document (entre deux lignes contenant **---**).

```
--- # Document
  - config: &conf01
    val_1: 0
    val_2: 100

  - config: &conf02
    val_1: 50
    val_2: 75

  - etape_1: *conf01 # fait référence à la première configuration
  - etape_2: *conf02 # fait référence à la deuxième configuration

--- # Document 2
  - etape_2_1: *conf01 # ERREUR ! La référence n'existe que dans
                       # le document précédent
```

Pour une référence complète de la syntaxe YAML, vous pourrez consulter les spécifications de la version 1.2, qui sont hébergées sur le site officiel du projet **[1]**.

Passons maintenant à l'aspect pratique.

2 Traitement en Python

En Python, comme toujours, lorsque quelque chose n'est pas traité nativement il faut installer un nouveau module. Pour YAML, ce module se nomme **PyYAML [2]** et il pourra être installé :

- soit à l'aide de **pip** :

```
$ sudo pip install pyyaml
```

- soit à l'aide du gestionnaire de paquetages intégré à votre distribution. Sur une distribution basée sur Debian, les paquetages seront **python-yaml** ou **python3-yaml** suivant que vous souhaitiez travailler en Python 2 ou Python 3 :

```
$ sudo aptitude install python3-yaml
```

Nous allons commencer par voir comment sont représentées les données YAML en Python. Pour cela, nous allons utiliser le shell Python interactif.

2.1 Du YAML dans Python

Le module PyYAML (qui se charge sous le nom **yaml**), nous propose une fonction de chargement de données au format YAML (**load**) et une fonction de transformation d'un dictionnaire au format YAML (**dump**).

```
>>> import yaml
>>> yaml.load("""
...   nom: Tux
...   adresse: |
...     666 route Emacs
...     29812 GNUCity
... """)
{'nom': 'Tux', 'adresse': '666 route Emacs\n29812 GNUCity\n'}
```

Ici, nous avons utilisé une chaîne de caractères multiligne contenant des données au format YAML. La fonction **load** a lu les informations et les a structurées sous forme de dictionnaire. On peut voir que l'utilisation du caractère **|** a bien conservé les retours à la ligne. En utilisant la fonction **dump** sur le dictionnaire résultant, nous retrouvons fort logiquement la syntaxe YAML de départ :

```
>>> yaml.dump({'nom': 'Tux', 'adresse': '666 route Emacs\n29812 GNUCity\n'})
"{adresse: '666 route Emacs\n\n    29812 GNUCity\n\n    ', nom: Tux}\n"
```

Bien sûr, la syntaxe utilisée ici est moins lisible, mais il s'agit bien exactement de la même chose. Vous pouvez enregistrer la ligne résultante dans un fichier pour la lire ensuite en tant que fichier YAML.

2.2 Lecture d'un fichier YAML

Les fichiers YAML sont lus sous forme de flux de caractères. Le lien vers le fichier qui sera ouvert en lecture ou en écriture est un lien classique qui utilise la fonction **open** et qui contiendra le même traitement des erreurs. L'écriture des données ne pose aucun problème :

```
>>> stream = open('doc.yaml', 'w')
>>> stream.write(yaml.dump({'nom': 'Tux', 'adresse': '666 route
Emacs\n29812 GNUCity\n'}))
64
>>> stream.close()
```

Pour la lecture, nous allons utiliser une nouvelle fonction : **load**. Celle-ci prend un flux en entrée et en extrait les informations au format YAML pour constituer une structure utilisable en Python (ici, il s'agit d'un dictionnaire) :

```
>>> stream = open('doc.yaml', 'r')
>>> dico = yaml.load(stream)
>>> print(dico)
{'nom': 'Tux', 'adresse': '666 route Emacs\n29812 GNUCity\n'}
>>> print(dico['nom'])
Tux
>>> stream.close
```

Si vous souhaitez utiliser plusieurs documents à l'intérieur d'un même fichier, il faudra utiliser la fonction **load_all**. Pour illustrer son fonctionnement, considérons que nous ayons le fichier YAML suivant (**doc.yaml**) :

```
--- # Configuration
  min: 0
  max: 100

--- # Experience 1
  - start: 5
  - speed: 20
  - laps: 10

--- # Experience 2
  - start: 12
  - speed: 60
  - laps: 4
```

Dans ce fichier, il y a trois documents qui sont séparés par les caractères **---**. Le programme Python permettant de lire ces données est :

```
01: import yaml
02:
03: with open('doc.yaml', 'r') as f:
04:     stream = yaml.load_all(f)
05:     for doc in stream:
06:         print(doc)
```

À chaque itération de la boucle **for** de la ligne 5, la variable **doc** contient un nouveau document qu'il convient ensuite de traiter en fonction des données attendues. Dans cet exemple, nous nous contentons d'afficher les données lues :

```
{'max': 100, 'min': 0}
[{'start': 5}, {'speed': 20}, {'laps': 10}]
[{'start': 12}, {'speed': 60}, {'laps': 4}]
```

Conclusion

Le format YAML est assez léger pour être utilisé très simplement et en même temps, il permet une structuration des données suffisamment fine pour en faire autre chose qu'une simple lecture de données de configuration. Plus complexe que du INI ou du CSV, mais plus simple et moins verbeux que du XML, le YAML est le format parfait pour les données structurées qui restent lisibles par un être humain. Par contre, vous devrez vérifier vous-même que le fichier respecte la grammaire attendue : on ne peut pas tout avoir... ■

Références

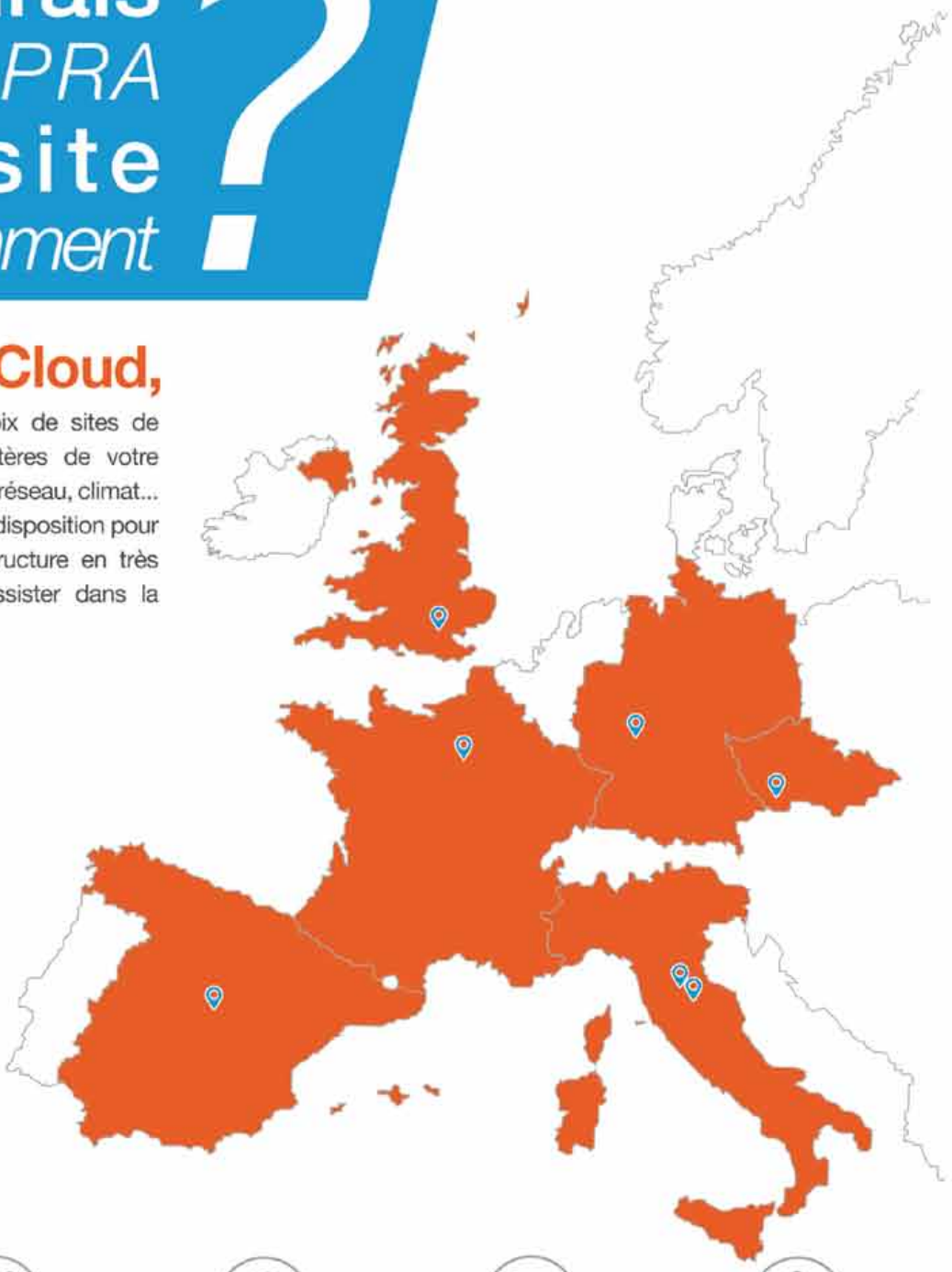
[1] Spécifications de YAML 1.2 : http://www.yaml.org/spec/1.2/spec.html

[2] Site officiel de PyYAML : http://pyyaml.org/wiki/PyYAML

Je voudrais *bâtir un PRA* **multi-site** *Je fais comment* ?

Avec Aruba Cloud,

vous disposez d'un large choix de sites de secours, en fonction des critères de votre stratégie de sécurité: proximité, réseau, climat... Nos équipes sont aussi à votre disposition pour vous aider à bâtir une infrastructure en très haute disponibilité et vous assister dans la définition de votre stratégie.



- 3 hyperviseurs
- 6 datacenters en Europe
- APIs et connecteurs
- 70+ templates
- Contrôle des coûts

Nous avons choisi Aruba Cloud car nous bénéficions d'un haut niveau de performance, à des coûts contrôlés et surtout car ils sont à dimension humaine, comme nous. Xavier Dufour - Directeur R&D - ITMP

Contactez-nous! 0810 710 300 www.arubacloud.fr

aruba CLOUD

MY COUNTRY. MY CLOUD.*

Cloud Public | Cloud Privé | Cloud Hybride | Cloud Storage | Infogérance

BESOIN DE **SUPPORT** POUR VOS LOGICIELS LIBRES ?

Découvrez

OSSA

OPEN SOURCE SOFTWARE ASSURANCE

EN MAI, VENEZ PARTICIPER À NOTRE SÉMINAIRE GRATUIT CONSACRÉ À L'OSSA !

15 MAI 2014 9H00-12H00 à LINAGORA Puteaux

Information et inscription sur www.linagora.com

L'OSSA est l'offre exclusive de support et de maintenance des logiciels libres de LINAGORA depuis plus de 10 ans pour des grandes organisations : la Direction générale des Finances publiques (DGFIP), la Direction générale des douanes et droits indirects (DGDDI), l'Insee, Orange, SNCF, la Banque Postale, l'INSERM, Chorégie, la Banque de France, le Ministère de la Défense, l'Assemblée Nationale, etc.

L'OSSA est disponible au travers du 08000linux.com ou au N°Vert 0 8000 LINUX

***Linagora est titulaire du marché de l'UGAP du support sur Logiciels Libres.**

LINAGORA