N°181 avril 2015 DÉVELOPPEMENT ET ADMINISTRATION SUR SYSTÈMES OPEN SOURCE

GNU LINUX MAGAZINE / FRANCE

CODE / DOCUMENTATION
Testez une façon originale de documenter votre code avec apidoc p.60

LANGAGE / POSTSCRIPT
Plongez dans un langage exotique p.66

KUNAI / ACTU
Découvrez une nouvelle solution de service discovery
p.06

ALGO / DONNÉES
Huffman ou LZW ?

MAÎTRISEZ LES ALGORITHMES DE COMPRESSION !
p.26

PYTHON / AUTHENTIFICATION
Identifiez vos utilisateurs à l'aide des réseaux sociaux grâce à Authomatic p.48

ANDROID / CORDOVA
Retrouvez Cordova et le développement d'applications multiplateformes p.54

SYSADMIN / NETBSD
Mettez en place votre SAN/NAS avec RAIDframe p.36

L 19275 - 181 - F: 7,90 € - RD

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

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
Chef des rédactions : Denis Bodor
Rédacteur en chef : Tristan Colombo
Réalisation graphique : Kathrin Scali & Carine Greppat
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 €

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

Membre April www.april.org

SUIVEZ-NOUS SUR :

https://www.facebook.com/editionsdiamond

@gnulinuxmag

LES ABONNEMENTS ET LES ANCIENS NUMÉROS SONT DISPONIBLES !

En version papier et PDF :
boutique.ed-diamond.com

Codes sources sur https://github.com/glmf

ÉDITORIAL

De jour en jour la quantité de données que nous collectons augmente. Ces données il faut les stocker, ce qui occupe forcément un espace physique non négligeable. Plus la quantité de données croît, plus il faut d'espace pour les stocker, c'est logique. Mais que se passe-t-il si l'espace de stockage ne croît pas suffisamment vite ? Il me vient à l'esprit une image couramment employée pour montrer l'expansion de l'Univers et qui peut s'appliquer ici.

La question de départ est la suivante : pourquoi ne fait-il pas jour tout le temps puisque les étoiles emplissent l'Univers de lumière ? Ceci s'explique par l'expansion de l'Univers : si nous prenons un robinet qui coule à débit constant pour remplir une baignoire, mais que cette baignoire grossit sans cesse, l'eau ne pourra jamais la remplir et déborder. L'eau représente la lumière et la baignoire l'Univers : l'Univers étant en expansion, la lumière émise ne pourra jamais le remplir. Dans le cas des données, on peut se dire que l'on se retrouve dans la même situation : la quantité de données acquise croît, mais dans le même temps les capacités de stockage croissent encore plus vite... (sinon Google aurait de gros problèmes avec Gmail). Mais à force de stocker tout et n'importe quoi, de ne plus faire le ménage dans nos données et d'en collecter toujours plus de manière automatisée, ne va-t-il pas arriver un moment où notre capacité de stockage nous fera défaut ? Nos smartphones ont des capteurs de plus en plus performants et, sans modification des paramètres comme le font la plupart des gens, il faut être en possession d'un volume de mémoire assez conséquent pour conserver des « photos ». Heureusement que ces dernières utilisent un format compressé, un format qui nécessite un traitement avant de pouvoir être archivé, de manière à occuper le moins de place possible. Une fois compressées, il faut passer par une étape de décompression, les données n'étant pas lisibles en l'état. Ce mois-ci vous pourrez expérimenter différents algorithmes de compression pour en disséquer le fonctionnement et adapter éventuellement l'un d'entre eux aux données que vous manipulez et ne plus dépendre exclusivement de l'expansion de l'espace de stockage.

J'espère que vous aurez noté que, pour la première fois, cet édito aura revêtu la forme d'un calligramme ! Un poisson aurait sans doute été de mise en ce mois d'avril, mais un poisson compressé n'aurait pas été du meilleur effet en première page...

Tristan Colombo

AGENDA

PHP Unconference Europe 2015

Date : les 9 et 10 mai 2015

Lieu : Majorque - Espagne

Site officiel : http://www.phpuceu.org/

Une *unconference* ou, littéralement, une « non-conférence » reste une conférence malgré le préfixe privatif, mais une conférence dans laquelle on privilégie les discussions sous forme d'échanges informels entre participants plutôt que de suivre un programme rigide et préétabli. Il n'y a donc pas de programme pour ce type d'événement : le premier jour les participants décrivent leurs centres d'intérêt (en lien avec PHP, bien sûr...) et leurs suggestions de sessions puis, les deux jours sont organisés en fonction des réponses.

Rappel : ScilabTEC 2015

Date : les 21 et 22 mai 2015

Lieu : Paris - France (en anglais)

Site officiel : http://www.scilabtec.com/

ScilabTEC est une conférence internationale des utilisateurs de Scilab, le logiciel de calcul numérique, et de Xcos, l'éditeur graphique de modèles de systèmes dynamiques hybrides. Le programme a été publié (http://www.scilabtec.com/index.php/program) et couvre des domaines d'application très variés : astronomie, robotique, médecine, etc.

SOMMAIRE

GNU/LINUX MAGAZINE FRANCE
N° 181

DjangoCon europe 2015

Date : du 31 mai au 5 juin 2015

Lieu : Cardiff - Royaume-Uni

Site officiel : http://2015.djangocon.eu/

L'événement annuel des djangonautes européens (utilisateurs du framework Web Django sous Python) aura lieu à Cardiff. Les six jours de conférences seront répartis de la manière suivante :

- un premier jour de présentations et tutoriels destinés plus spécifiquement aux débutants ;
- les trois jours suivants sont dédiés aux présentations qui pourront prendre l'un des trois formats suivant : présentation longue, présentation courte ou présentation éclair.
- les deux derniers jours seront l'occasion de *sprints*, tutoriels et « *clinics* » (demandez de l'aide sur l'un de vos codes à des développeurs Django expérimentés).

International PHP Conference

Date : les 9 et 10 juin 2015

Deadline : 18 mars 2015

Lieu : Berlin - Allemagne (en anglais et quelques rares présentations en allemand)

Site officiel : https://phpconference.com/2015se/en

Les sujets abordés lors de cette conférence couvriront tous les domaines du développement Web, que ce soit les APIs des services Web, les méthodes Agiles, le Cloud, JavaScript ou vraiment du PHP « pur et dur » (frameworks, performances, sécurité, tests, etc.). Si vous vous intéressez justement aux tests unitaires, il faut noter la présence de Sebastian Bergman, le créateur de PHPUnit, qui interviendra plusieurs fois. ■

KUNAI, LE SERVICE DISCOVERY SIMPLEMENT

par **Jean Gabès** *[Auteur de Kunai et de Shinken, CTO Shinken Solutions]*

Nos systèmes et architectures sont de plus en plus complexes et distribués, mais le plus gros changement concerne leur cycle de vie qui a fortement évolué dernièrement. L'avènement des VM (et des conteneurs) jetables est arrivé, et avec lui son lot de nouveaux défis.

À une époque pas si lointaine, un serveur était installé (à la main) pour durer des années. Grâce aux outils de gestion de configuration et de déploiements, nous avons à notre portée des architectures où des nœuds peuvent avoir une durée de vie inférieure à l'heure afin d'absorber un pic de charge. Si pris unitairement la gestion d'un nouveau nœud n'est plus un souci, se pose encore celui du comportement vis-à-vis des autres membres du cluster. Et ceci avec en tout premier lieu une question simple : comment faire pour que les autres soient au courant de sa présence, et ce le plus rapidement possible dès qu'il est prêt ?

C'est à cette question que répondent les outils de *service discovery*, dont nous allons voir un représentant dans cet article : **Kunai**.

1 Une installation simple

Étant un projet récent, Kunai **[1]** n'est pas encore disponible nativement dans vos distributions linux, même si des dépôts pour votre distribution devraient être disponibles à l'heure où vous lisez ces lignes. Écrit en Python par l'auteur de Shinken (votre serviteur), son installation est des plus simples en utilisant la commande `pip` :

```
$root@debian: apt-get install python-requests python-openssl
python-setuptools python-pip python-leveldb python-jinja2
$root@debian: pip install kunai==0.9
```

Les chemins d'installation sont relativement classiques :

- `/etc/kunai` : la configuration ;
- `/usr/bin/kunai` : commande cli ;
- `/var/lib/kunai` : données diverses ;
- `/var/log/kunai` : les logs.

Vous pouvez alors démarrer votre daemon kunai :

```
$root@debian: /etc/init.d/kunai start
```

2 Intérêts des outils de service discovery

2.1 Les effets du cloud et de l'élasticité

Les outils de *service discovery* ne sont pas nouveaux, mais leur intérêt a fortement augmenté ces derniers temps. Leur but principal est de permettre à des nœuds de publier à d'autres les services qu'ils proposent. Par exemple, dans le cas d'un site de e-commerce, il y a en général un ou deux répartiteurs de charge (par exemple avec un daemon **HAProxy**) et une multitude de serveurs Web qui vont répondre aux requêtes.

Vu que les consultations de ce genre de site ne sont pas réparties équitablement au fil de la journée, il est très tentant de ne lancer la majorité des serveurs Web que lorsque c'est nécessaire (par exemple en début de soirée) sur des plateformes du type *EC2*, et de les éteindre dès que la charge est finie. L'économie ainsi réalisée est très intéressante.

2.2 Une forte réactivité aux changements

Seulement il faut alors que le serveur de répartition soit au courant dès qu'un nouveau serveur Web est prêt, et qu'il en tienne compte (qu'il le rajoute dans sa configuration et la recharge). Les outils de gestion de configuration peuvent techniquement le faire, mais leur latence de prise en compte des changements est grande. C'est la raison pour laquelle les outils de *service discovery* sont utiles.

Leur force est d'être très réactifs face aux changements de l'environnement des clusters applicatifs. Là où il faudrait plusieurs minutes pour qu'un outil de gestion de configuration réagisse, il ne faut qu'une poignée de secondes pour un outil de *service discovery*. Cependant, leur spectre d'usage est bien plus limité, et ne vous attendez pas à provisionner un serveur à partir de ces outils ou à gérer des dépendances complexes. À chaque problématique ses outils :)

2.3 Un fonctionnement coopératif et totalement décentralisé

Kunai est un outil de *service discovery*, et à ce titre son premier objectif est qu'un nœud (machine physique, virtuelle ou conteneur) puisse facilement se faire connaître auprès des autres. Contrairement à d'autres outils (comme **etcd [2]**), Kunai est totalement décentralisé. Tous les nœuds sont au même niveau de responsabilité et de connaissance. D'un point de vue connectivité réseaux, les nœuds d'un même cluster Kunai doivent être capables d'échanger entre eux.

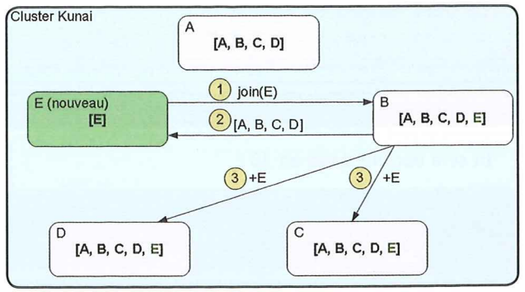


Fig. 1 : Arrivée d'un nœud dans un cluster gossip.

Un nœud a constamment en mémoire la liste complète des autres nœuds ainsi que leur rôle (sous forme de *tags*). Cette liste peut facilement être requêtée par des applications tierces tournant sur le serveur de l'agent Kunai via divers protocoles que nous verrons par la suite.

La gestion du cluster formé par les daemons Kunai se fait via un algorithme de type *gossip* **[3]**. Un article complet lui sera consacré prochainement. Pour faire simple, un nouveau nœud doit se présenter auprès d'un membre existant du cluster. Ce dernier lui fournit la liste complète des autres nœuds, et propage la nouvelle du nouveau membre aux autres, et ceci de proche en proche. Les figures 1 et 2 illustrent ce fonctionnement.

3 Exemple avec trois nœuds Kunai

3.1 Première configuration et entrée dans le cluster Kunai

Dans notre exemple, nous allons partir sur 3 nœuds Kunai : un serveur HAProxy et deux serveurs http. Une fois l'installation effectuée sur nos trois serveurs, vous pouvez démarrer vos daemons Kunai. Comme vu précédemment, un nœud seul n'est guère utile, il a besoin de rejoindre un cluster pour être utile.

Nos serveurs seront :

- `srv-haproxy` : 192.168.0.1 ;
- `srv-web1` : 192.168.0.2 ;
- `srv-web2` : 192.168.0.3.

Nous allons configurer les nœuds avec des tags afin qu'ils connaissent leur rôle et chargent par la suite les bons *checks* et les bons services associés à ces tags.

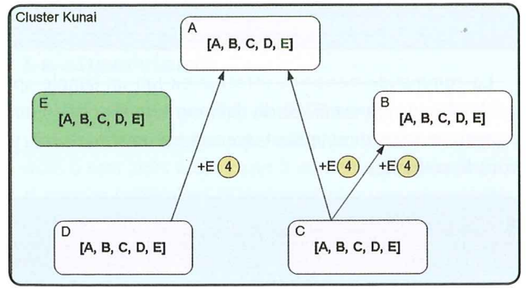


Fig. 2 : Propagation de l'arrivée d'un nouveau nœud.

La configuration des nœuds se fait dans leur fichier **/etc/kunai/local.json**. Celui du nœud **haproxy** va ressembler à :

```
{
    "port"           : 6768,
    "datacenters"    : ["dc1"],
    "data"           : "/var/lib/kunai/data",
    "log"            : "/var/log/kunai",
    "seeds"          : [],
    "tags"           : ["haproxy", "linux"],

    "encryption_key" : "",
    "master_key_priv": "",
    "master_key_pub" : "mfkey.pub"

}
```

Pour l'instant, nous n'activons pas le chiffrement entre les nœuds, nous le ferons dans un prochain article qui sera dédié à des fonctionnalités plus avancées. Nous avons configuré les tags **haproxy** et **linux**. Ce dernier arrive avec des *checks* prédéfinis. Pour **haproxy**, il sera utile par la suite pour définir des *checks*, mais également un service associé.

Dans le cas de nos serveurs http, les tags seront :

```
{
   [...]
   "tags"           : ["http", "linux"],
   [...]
}
```

Lorsqu'un nœud démarre, il peut avoir dans son fichier de configuration une liste d'autres nœuds du cluster avec lesquels il va échanger pour rejoindre le cluster. Le paramètre en question est **seeds**. N'importe quel nœud fera l'affaire et ne sera utilisé qu'au premier lancement du daemon.

Listons les membres Kunai vus par le nœud **haproxy** pour l'instant :

```
$root@srv-haproxy: kunai members
srv-haproxy alive srv-haproxy:6768  haproxy,linux
```

La commande **members** effectue en fait un simple appel HTTP sur le port **6768** du daemon **kunai** local. Vous pouvez le faire directement avec la commande **curl** si vous le souhaitez :

```
$root@srv-haproxy: curl http://srv-haproxy:6768/agent/members
{"...": {"uuid": "...", "addr": "srv-haproxy", "state": "alive",
"services": {}, "incarnation": 1, "port": 6768, "tags":
["haproxy", "linux"], "name": "srv-haproxy"}, [...]}
```

Pour l'instant, le nœud **srv-haproxy** ne se voit que lui-même. Nous allons donc rajouter les autres nœuds. Pour cela, nous allons utiliser la commande **join** de kunai. Elle prend en paramètre l'adresse d'un autre membre du cluster. Nous aurions tout aussi bien pu lancer la commande depuis l'un ou l'autre des deux autres serveurs :

```
$root@srv-haproxy: kunai join srv-web1
Joining srv-web1 : OK
$root@srv-haproxy: kunai join srv-web2
Joining srv-web2 : OK
```

Une fois ceci fait, vous pouvez demander la liste des nœuds connus par votre agent local :

```
$root@srv-haproxy: kunai members
srv-haproxy alive srv-haproxy:6768  haproxy,linux
srv-web1    alive srv-web1:6768     http,linux
srv-web2    alive srv-web2:6768     http,linux
```

Si nous lançons la commande depuis un autre nœud nous aurons le même résultat.

```
$root@srv-web1: kunai members
srv-haproxy alive srv-haproxy:6768  haproxy,linux
srv-web1    alive srv-web1:6768     http,linux
srv-web2    alive srv-web2:6768     http,linux
```

3.2 Détection d'un nœud tombé

Tous les nœuds d'un cluster kunai se *ping* entre eux très régulièrement et aléatoirement, en UDP sur leur port **6768**. Un nœud qui ne répond plus sera identifié très rapidement (quelques secondes environ) et l'information sera transmise à tous les autres nœuds. Ici, nous allons tuer le processus sur le nœud **srv-web1**.

Avec le paramétrage par défaut, il sera identifié comme **suspect** en 6 secondes :

```
$root@srv-haproxy: kunai members
srv-haproxy alive    srv-haproxy:6768  haproxy,linux
srv-web1    suspect  srv-web1:6768     http,linux
srv-web2    alive    srv-web2:6768     http,linux
```

Et sera déclaré **dead** en 30 :

```
$root@srv-haproxy: kunai members
srv-haproxy alive srv-haproxy:6768   haproxy,linux
srv-web1    dead  srv-web1:6768      http,linux
srv-web2    alive srv-web2:6768      http,linux
```


L'information sera la même au sein du cluster en quelques secondes seulement.

Si nous redémarrons le nœud, il revient en état **alive** au sein du cluster quasi immédiatement :

```
$root@srv-haproxy: kunai members
srv-haproxy  alive  srv-haproxy:6768  haproxy,linux
srv-web1     alive  srv-web1:6768     http,linux
srv-web2     alive  srv-web2:6768     http,linux
```

3.3 Définition du service http et son check associé

Pour l'instant, nos nœuds sont dans le même cluster et peuvent savoir rapidement quand un membre est disponible. Il est désormais temps de leur donner de vrais rôles et d'être un peu plus utiles qu'un simple **ping** entre les nœuds.

Nous allons commencer par nos serveurs Web en leur définissant un service **http** ainsi qu'un check associé. Pour les checks, Kunai est notamment capable d'utiliser les sondes *nagios/shinken*. Nous allons utiliser ici la sonde **check_http** disponible dans le paquet **nagios-plugins**.

```
$root@srv-web1: apt-get install nagios-plugins
$root@srv-web2: apt-get install nagios-plugins
```

Une petite recherche après, nous avons le chemin vers la sonde :

```
$root@srv-web1: updatedb && locate check_http
/usr/lib/nagios/plugins/check_http
```

Son utilisation est très simple, ici pour superviser la disponibilité du serveur Web local :

```
$root@srv-web1: /usr/lib/nagios/plugins/check_http -H localhost
HTTP OK: HTTP/1.1 200 OK - 18393 bytes in 0.002 second response time
```

Créons notre service **http** dans un fichier **/etc/kunai/http.json** sur nos serveurs Web :

```
{
    "service": {
        "tags": ["master"],
        "port": 80,
        "apply_on": "http",
        "check": {
            "script": "/usr/lib/nagios/plugins/check_http -H localhost",
            "interval": "10s",
            "handlers": ["default"]
        }
    }
}
```

Nous n'avons pas besoin de donner un nom au service, c'est le nom du fichier qui sera pris en compte (en tenant compte des sous-répertoires à partir du répertoire **/etc/kunai** bien évidement afin de ne pas avoir de doublon possible).

Ici, le service sera activé pour tous les nœuds qui ont un tag *http*, et le *check* de vérification sera lancé toutes les 10 secondes. Libre à vous de descendre cette valeur si vous souhaitez une supervision encore plus rapide.

Les paramètres **tags** et **port** seront utiles dans le cadre de la découverte de services par d'autres applications. Le paramètre **handlers** du *check* est quant à lui utilisé pour savoir quelle action effectuer lors du retour d'un *check*. Le *handler* **default** est défini dans */etc/kunai/default.json* et consiste en un envoi de mail lorsqu'un *check* passe en erreur.

Une relance du daemon faite, vous pouvez alors vérifier que le service et son *check* sont bien pris en compte via la commande **state** de kunai :

```
$root@srv-web1: kunai state
Services:
   http   OK  - HTTP OK: HTTP/1.1 200 OK - 18376 bytes in 0.024
second response time
Checks:
   service:http  OK  - HTTP OK: HTTP/1.1 200 OK - 18376 bytes in
0.024 second response time
```

Ici, l'état du service sera bien évidemment égal à celui de son *check* associé. Il est tout à fait possible de définir un *check* en dehors d'un service, et il alertera par mail un administrateur lorsqu'il passera dans un état anormal sans que l'information ne soit transférée aux autres nœuds (par exemple pour un simple *check* de mémoire).

La supervision

Il est tout à fait possible de détourner Kunai pour l'utiliser comme un outil de supervision totalement distribué, mais il sera alors à comparer à un Sensu plutôt qu'à un Nagios ou un Shinken. Si vous souhaitez une supervision avancée (gestion des périodes de temps, de la corrélation et autres escalades de notifications), vous devrez le lier à un véritable outil de supervision comme Shinken justement.

Pour l'instant, nous définissons le fichier de configuration **http.json** à la main sur nos trois serveurs. Il est possible de mettre en place une synchronisation automatique des fichiers de configuration entre les nœuds (autre que le fichier **local.json** qui reste local à la machine), mais ceci sera vu dans un prochain article qui analysera les capacités avancées de Kunai.

3.4 Définition du service haproxy

La définition du service **haproxy** pour notre nœud **srv-haproxy** sera très similaire à celle des nœuds http. Ici encore, nous limiterons à un simple *check* HTTP pour voir si le service **haproxy** est disponible. De nombreuses sondes du monde *nagios/shinken* sont disponibles pour effectuer une supervision plus approfondie, il est laissé au lecteur le soin de voir la méthode qui lui convient le mieux.

Créons notre service **haproxy** dans le fichier **/etc/kunai/haproxy.json** sur notre serveur **srv-haproxy** :

```
{
    "service": {
        "tags": ["master"],
        "port": 80,
        "apply_on": "haproxy",
        "check": {
            "script": "/usr/lib/nagios/plugins/check_http -H localhost",
            "interval": "10s",
            "handlers": ["default"]
        }
    }
}
```

Une fois encore vous pouvez vérifier que le service et son *check* sont bien pris en compte :

```
$root@srv-haproxy: kunai state
Services:
   haproxy  OK  - HTTP OK: HTTP/1.1 200 OK - 18371 bytes in
0.009 second response time
Checks:
   service:haproxy  OK  - HTTP OK: HTTP/1.1 200 OK - 18371 bytes
in 0.009 second response time
```

3.5 Création automatique du fichier haproxy.cfg

Nous avons nos services **http** et notre service **haproxy**, mais notre réel objectif est que lorsqu'un nouvel hôte http arrive, il soit inséré directement dans la configuration du daemon **haproxy** et de recharger ce dernier.

C'est justement le rôle des **generators**. Ces objets sont très simples. Ils prennent un fichier *template* (défini par rapport au répertoire **/etc/kunai/templates**) et génèrent un fichier final grâce aux informations disponibles dans l'agent kunai. Une fois ceci effectué, ils lancent une commande, généralement une demande de rechargement de daemon.

Nous allons ici utiliser le template **haproxy.cfg** défini dans **/etc/kunai/templates/haproxy.cfg**, qui va générer le fichier **/etc/haproxy/haproxy.cfg** et qui lancera ensuite le rechargement de haproxy. Nous définissons cet objet au sein du fichier **/etc/kunai/generators/haproxy.cfg** :

```
{
    "generator": {
        "apply_on": "haproxy",
        "template": "haproxy.cfg",
        "path":     "/etc/haproxy/haproxy.cfg",
        "command":  "/etc/init.d/haproxy reload"
    }
}
```

Afin de nous faciliter la tâche, nous allons récupérer le fichier de configuration de haproxy comme *template*.

```
$root@srv-haproxy: cp /etc/haproxy/haproxy.cfg /etc/kunai/
templates/haproxy.cfg
```

La configuration de haproxy est organisée en plusieurs blocs, avec notamment un bloc **global** et un bloc **defaults**. Nous allons rajouter un bloc **listen** qui nous permettra de définir un simple *load balancing* HTTP.

Afin qu'il redirige vers nos deux serveurs Web, nous avons besoin de générer un bloc tel que :

```
# Main load balancing
listen http_cluster *:80
        mode http
        balance roundrobin
        option httpclose
        option forwardfor

        # Real servers
       server srv-web1 srv-web1:80 check
       server srv-web2 srv-web2:80 check
```

La syntaxe du système de template de Kunai est celle de la librairie **jinja2 [4]**. Elle est très flexible et nous permet de facilement mettre en place des boucles et autres tests sur nos nœuds et nos services. Nous allons alors simplement rajouter le bloc ci-dessous à notre fichier *template* :


```
# Main load balancing
listen http_cluster *:80
        mode http
        balance roundrobin
        option httpclose
        option forwardfor

        # Real servers
    {% for (node, service) in ok_nodes(service='http') %}
       server {{node['name']}} {{node['addr']}}:{{service['port']}} check
    {% endfor %}
```

La fonction **ok_nodes** proposée par les **generators** de Kunai nous permet d'obtenir facilement les nœuds et leurs services s'ils sont vivants et que leur service est également en statut **OK**. Nous pouvons récupérer la définition de la propriété **port** de notre service **http** directement depuis notre objet service.

Nous pouvons vérifier sur la page de statistiques de haproxy que notre *load balancing* **http_cluster** est bien défini et qu'il est composé de nos deux nœuds **srv-web1** et **srv-web2** (voir figure 3).

http_cluster

	Queue			Session rate			
	Cur	Max	Limit	Cur	Max	Limit	Cur
Frontend				1	2	-	1
srv-web1	0	0	-	0	1		0
srv-web2	0	0	-	0	1		0
Backend	0	0		0	1		0

Fig. 3 : Prise en compte de la configuration par **haproxy**

Si nous arrêtons le service apache sur le serveur **srv-web2**, nous pouvons voir que le *check* va passer en état *critical* :

```
$root@srv-web2: /etc/init.d/apache2 stop
$root@srv-web2: kunai state
Services:
   http  CRITICAL  - connect to address localhost and port 80:
Connection refused
Checks:
   service:http  CRITICAL - connect to address localhost and
port 80: Connection refused
```

http_cluster

	Queue			Session rate			
	Cur	Max	Limit	Cur	Max	Limit	Cur
Frontend				1	2	-	1
srv-web1	0	0	-	0	1		0
Backend	0	0		0	1		0

Fig. 4 : Prise en compte par **haproxy** *d'un service http tombé.*

http_cluster

	Queue			Session rate			
	Cur	Max	Limit	Cur	Max	Limit	Cur
Frontend				1	2	-	1
srv-web1	0	0	-	0	1		0
srv-web2	0	0	-	0	1		0
Backend	0	0		0	1		0

Fig. 5 : Rajout d'un nouveau nœud http.

Et quasi instantanément notre configuration haproxy a été mise à jour sur le nœud **srv-haproxy** :

```
# Real servers
server srv-web1 srv-web1:80 check
```

Ce que confirme le daemon **haproxy** comme on peut le voir en figure 4.

Cependant, même si c'est pratique, détecter les nœuds tombés n'est pas la principale raison de l'utilisation d'un outil de *service discovery*. Ici, par exemple, HAProxy est déjà capable de détecter les serveurs qui ne répondent plus. Mais il devient très utile dans le cas de rajout d'un nouveau nœud http. Le simple démarrage d'un nœud le rajoutera automatiquement dans la configuration de notre *load balancer*.

Par exemple si l'on démarre un nouveau serveur **srv-web3**, clone d'un des deux premiers, il sera rajouté à notre cluster HTTP comme le montre la figure 5.

4 Accéder aux données d'inventaires

4.1 Requêter les informations par http

Les daemons Kunai écoutent par défaut en HTTP sur leur port **6768**. Pour vos tests et la lisibilité des données, il est conseillé d'utiliser l'outil **httpie** qui mettra en forme les données json disponibles via l'api des daemons. Pour l'installer, c'est très simple :

```
$root@srv-haproxy: apt-get install httpie
```

Avoir la liste des nœuds qui exposent le service **http** se résume alors à :

Fig. 6 : Interface de Kunai.

```
$root@srv-haproxy: http -b 192.168.0.1:6768/state/services/http
{
    [...]
    "apply_on": "haproxy",
    "failing": 0,
    "failing-members": [],
    "incarnation": 0,
    "members": ["srv-web1", "srv-web2"],
    "name": "http",
    "passing": 2,
    "passing-members": ["srv-web1", "srv-web2"],
    "port": 80,
    "state_id": 0,
    "tags": ["master"]
}
```

Dans la valeur **members** vous aurez la liste complète des nœuds exportant ce service. Ils seront répartis entre deux listes : **failing-members** avec ceux ayant un problème, et **passing-members** ceux qui sont en état **ok**.

4.2 Requêter par DNS

Si HTTP reste un protocole facile pour récupérer des données, il en existe un qui est déjà utilisé en très grande majorité par les applications et ce depuis un long moment afin d'identifier des nœuds distants : le DNS.

Les daemons Kunai implémentent un serveur DNS. La configuration se fait au sein du fichier **/etc/kunai/dns.json**. Il est possible de définir son port d'écoute et son domaine d'activation. Par défaut, ce dernier est **.kunai** :

```
{
    "dns":{
        "enabled"    : true,
        "port"       : 6766,
        "domain"     : ".kunai"
    }
}
```

Afin d'avoir la liste des nœuds qui exportent sans soucis le service dans le *datacenter* **dc1** (paramètre du fichier **local.json** de chaque nœud), il suffit alors de lancer une simple requête DNS :

```
$root@srv-haproxy: dig @localhost -p 6766 http.service.dc1.kunai
[...]
;; QUESTION SECTION:
;http.service.dc1.kunai.              IN      A

;; ANSWER SECTION:
http.service.dc1.kunai. 60      IN      A       192.168.0.2
http.service.dc1.kunai. 60      IN      A       192.168.0.3
```

L'utilisation de **dig** est cependant limitée à notre test. Il serait bien plus utile d'inclure les résultats de notre daemon directement dans les recherches système. Les daemons n'ayant pas la vocation à gérer les requêtes DNS de tout le système, nous n'allons rediriger vers eux que le domaine **.kunai** grâce à l'utilisation du programme **dnsmasq**. Ce dernier ne gérera les requêtes que pour le domaine précisé et passera la main aux serveurs DNS suivants pour le reste.

Son installation est fort simple :

```
$root@srv-haproxy: apt-get install dnsmasq
```

Sa configuration l'est tout autant. Elle se situe dans le fichier **/etc/dnsmasq.conf** :

```
server=/.kunai/127.0.0.1#6766
```

Ici nous souhaitons que les demandes pour le domaine **.kunai** soient redirigées vers l'agent Kunai écoutant sur le port **6766**.

Nous pouvons alors ajouter le serveur **dnsmasq** à notre configuration système dans **/etc/resolv.conf** :

```
nameserver 127.0.0.1
nameserver 8.8.4.4
```

N'importe quelle application sur le système peut désormais échanger avec un nœud valide d'un cluster :

```
$root@srv-haproxy: curl http://http.service.dc1.kunai
It works!
```

Et ce depuis n'importe quel nœud Kunai vu qu'ils sont tous au même niveau de connaissance du cluster.

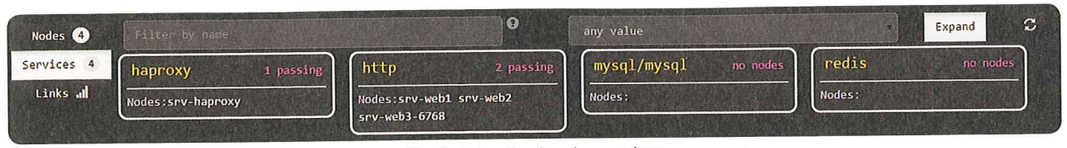


Fig. 8 : Visualisation des services.

5 Interface de visualisation

Les daemons Kunai incluent de base une interface de visualisation des nœuds, des services et des *checks*. Sa mise en place est extrêmement simple, car elle n'est pas interprétée côté serveur, mais uniquement côté client. Un simple copier/coller ou lien du répertoire **/var/lib/kunai/ui** vers un répertoire **/var/www/** fera l'affaire. Pour le test nous lancerons simplement un serveur HTTP depuis le répertoire de l'UI :

```
$root@srv-haproxy: cd /var/lib/kunai/ui
$root@srv-haproxy: python -m SimpleHTTPServer 8000
```

Vous pouvez dès lors accéder sur le port **8000** à l'interface de Kunai (voir figure 6). Nous pouvons y voir nos trois nœuds ainsi que le nœud **srv-web3** qui est en état **leave**, car il s'est éteint proprement et a donc pu prévenir de son départ.

Le principe de l'interface est d'être simple et réactive aux changements au sein du cluster d'agents. Lors de l'ouverture de la page, une requête est effectuée par défaut sur le daemon à la même adresse IP que le site. Pour modifier ceci, ou rajouter d'autres daemons utilisables pour récupérer les données (si le premier tombe par exemple), vous pouvez configurer la liste dans le fichier **/var/lib/kunai/ui.config.js**. Une fois connectée à un daemon, l'interface va récupérer l'ensemble des nœuds par http/json via l'API.

Ensuite, l'ensemble des données est mise à jour uniquement via une websocket ouverte vers le serveur Kunai. Lorsqu'un changement est propagé au sein du cluster Kunai par le nœud auquel l'interface est connectée, il sera également envoyé sur la websocket.

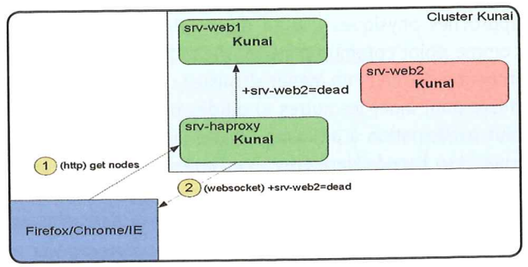


Fig. 7 : Architecture de l'interface de Kunai.

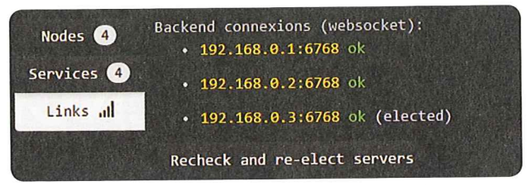


Fig. 9 : Visualisation des connexions websockets.

Ceci apporte une réactivité maximale à l'interface, tout changement au sein du cluster étant mis à jour instantanément sur l'interface. Une illustration de ce fonctionnement est disponible sur la figure 7.

Sur l'interface, vous pouvez voir des nœuds, leurs *checks*, mais également les services (figure 8). Il est notamment possible de filtrer les nœuds suivant leur tag en inscrivant **t:linux** dans le champ de recherche, ou bien ceux qui exportent un service en mettant cette fois-ci **s:http** par exemple.

Enfin, il est possible de vérifier l'état de la connexion websocket avec les différents daemons configurés. Lorsque la connexion active est perdue, une autre est automatiquement élue et l'interface se resynchronise. Ceci est visible sur la figure 9.

Conclusion

Nous n'avons fait qu'effleurer ce dont est capable Kunai ; nous avons déjà évoqué le chiffrement et la synchronisation automatique de la configuration et des sondes entre les nœuds. Mais Kunai a pour objectif d'être le meilleur ami de vos clusters applicatifs. Nous verrons lors de prochains articles comment exécuter de manière sécurisée une commande sur de multiples nœuds, laisser Kunai calculer pour vous un leader de cluster applicatif grâce à l'algorithme Raft [5] ou bien encore détecter et exposer facilement les services proposés par des conteneurs Docker. ■

Références

[1] Site du projet Kunai : http://kunai.io

[2] Site du projet Etcd : https://github.com/coreos/etcd

[3] Algorithmes de type gossip : http://en.wikipedia.org/wiki/Gossip_protocol

[4] Documentation de Jinja2 : http://http://jinja.pocoo.org/docs/dev

[5] Algorithme de consensus Raft : https://raftconsensus.github.io/

NOTRE 11 SEPTEMBRE : MÊMES EFFETS ET MÊMES CONSÉQUENCES

par **Tris Acatrinei** *[Consultante pour FAIR-Security]*

Suite aux attentats de Paris, le Gouvernement a souhaité mettre en place un certain nombre de mesures visant à éradiquer le terrorisme. Certains commentateurs ont parlé de 11 septembre français pour désigner les attaques du mois de janvier. Sur le moment, ça paraissait décalé. Mais en observant les raccourcis, le vocabulaire et les invectives dont ont usé nos dirigeants au mois de février 2015, on se rend compte que le parallèle était non seulement fondé, mais malheureusement prophétique.

1 Les 90% de Cazeneuve

Le 19 février 2015, avant de s'envoler vers les États-Unis, le ministre de l'Intérieur, Bernard Cazeneuve a fait la déclaration suivante au micro de BFM TV : « 90% des individus qui basculent dans le terrorisme le font par Internet. » Comme dirait Abe Burrow « la raison d'être des statistiques, c'est de donner raison ». Le problème étant que cette donnée chiffrée est basée sur un échantillon de 160 personnes. En effet, le centre de prévention contre les dérives sectaires liées à l'Islam, qui a rédigé un rapport **[1]**, intitulé « La métamorphose opérée chez le jeune par les nouveaux discours terroristes », souligne qu'Internet est effectivement un mode d'endoctrinement, mais précède le propos en expliquant très clairement pourquoi le Web est privilégié par rapport aux autres modes : facilité, déracinement, attitude de zapping et surtout, il se cantonne à poser Internet comme un mode de communication et non comme un mécanisme de basculement : « 98% du discours de l'islam radical utilise Internet, qui apparaît comme un moyen de communication qui permet de dépasser les contraintes de temps et d'espace, qui correspond aux pratiques de mise en réseau. ». Notons que le rapport date du mois de novembre 2014.

Par ailleurs, le document dresse une typologie des profils des personnes et on se rend compte que les différents réseaux sectaires utilisent les réseaux sociaux non seulement pour recruter, mais d'abord et surtout pour profiler les jeunes susceptibles de basculer. Ce ne sont pas les jeunes qui cherchent, ce sont les gourous qui entreprennent des travaux d'approche pour séduire les jeunes. Voici ce que dit le rapport dans la section consacrée aux approches physiques : « Au niveau des filles, elles ont comme point commun d'avoir affiché sur leur profil leur intention d'exercer un métier altruiste (« Je veux faire infirmière pour aider les autres ») ou des photos attestant de leur participation à un camp humanitaire (« Moi au Burkina Faso l'année dernière… »). Toutes ont été abordées sur une valorisation de leur engagement pour un monde plus juste, comme s'il existait des sortes de « chercheurs de tête » ou de mots-clés qui permettaient de les « repérer ». Cette différence n'est pas qu'anecdotique : une majeure

partie des jeunes étaient parfaitement intégrés, viennent de familles appartenant aux classes moyennes françaises, avec une histoire familiale linéaire et sans drame. Certains étaient scolarisés dans de grandes écoles comme Sciences Po ou des filières sportives de haut niveau. Une minorité (5%) est fichée comme délinquant et 40% de l'échantillon avait connu des épisodes de dépression. Au lieu de chercher des jeunes qui peuvent potentiellement être déjà repérés comme étant « à problèmes » du fait d'actes de délinquance, d'histoires familiales complexes, de difficultés scolaires, etc., on cherche à recruter des jeunes qui sont, en apparence, très bien insérés, sans difficultés apparentes, mais fragiles, car il sera plus difficile de les détecter. Si on voulait caricaturer, on pourrait faire le parallèle avec les trafiquants de stupéfiants qui utilisent des enfants pour faire les livraisons, car « naturellement » moins suspects aux yeux des forces de l'ordre. On retrouve également ce type d'approche chez les prédateurs sexuels, qui repèrent les personnes les plus fragiles psychologiquement, car plus faciles à manipuler.

Le rapport explique de façon très simple tout le processus d'endoctrinement des jeunes, les typologies de profils, le basculement et souligne bien que le Web n'est qu'un vecteur, un facilitateur et l'intègre dans un corpus de propagande.

D'une étude de 91 pages, écrite à plusieurs mains, intégrant des analyses théologiques, historiques, sociales, et culturelles, le ministre de l'Intérieur se fait l'exégète de l'anti-Internet, utilisant les mêmes raccourcis que certains extrémistes.

La répulsion éprouvée par Bernard Cazeneuve à l'encontre d'Internet n'est pas neuve, on connaissait les velléités du ministre de l'Intérieur à faire un procès en sorcellerie systématique à tout ce qui provient du réseau, comme si les terroristes attaquaient à coups d'octets les personnes et non avec des armes bien réelles.

2 La promenade dans la Silicon Valley

Après une halte à Washington D.C., notre ministre de l'Intérieur continue son petit voyage, direction la Silicon Valley pour discuter avec Facebook, Google, Apple et Twitter **[2]**. But de l'opération ? Obtenir de ces entreprises qu'elles retirent rapidement tous les contenus susceptibles d'inciter au terrorisme. Il les a félicités pour « l'effort par-

ticulier » qu'elles ont fourni, mais les appelle à redoubler d'efforts pour bloquer les messages extrémistes.

Première remarque : il semblerait que ce ne soit pas les grands patrons des différentes entreprises qui aient reçu le ministre, mais des équipes de juristes malgré la grande communication qui a été faite autour de cette visite.

Deuxième remarque : il semble très curieux que le ministre de l'Intérieur français demande à des entreprises de services américains le comportement qu'elles doivent adopter et très cavalier de les féliciter, comme si un lien de subordination invisible liait Bernard Cazeneuve et les sociétés de la Silicon Valley. Les remercier pour leur réactivité est normal et apprécié, les féliciter semble déplacé.

Troisième remarque : le ministre parle d'autorégulation. Or, l'autorégulation se caractérise par une régulation spontanée, humaine et sans pression d'une entité, selon des normes et des critères définis à l'avance et acceptés par une communauté. Typiquement, c'est le cas du canal IRC GCU : les règles sont annoncées sur le site du groupe et toute personne qui se connecte pour la première fois sait qu'elle doit montrer une capture d'écran, sous peine de bannissement. Ces règles ne sont pas imposées par Freenode, elles sont le résultat d'une expérience sur du long terme, de membres actifs.

Bernard Cazeneuve a invité les quatre géants à Paris, au mois d'avril pour « arrêter un code de bonne conduite ». Sur le plan du droit, on ne peut être que dépité(e) de voir un ministre de l'Intérieur déléguer des opérations de police à des acteurs privés. En effet, non seulement les sociétés invitées n'ont clairement pas attendu Bernard Cazeneuve pour travailler et faire le ménage sur leurs réseaux, mais surtout, bloquer et supprimer des contenus, en tenant compte de la volonté d'une force extérieure, sur le plan diplomatique, cela s'appelle de l'ingérence et sur le plan administratif, de la délégation. Problème : en droit français, les missions de police ne se délèguent pas. On objectera que cette visite et l'invitation qui s'en est suivie tiennent plus de la coopération, mais à partir du moment où un ministre dicte des règles à une entreprise privée, on dépasse le stade de la coopération.

Le problème de cette attitude particulièrement maladroite et grossière est qu'elle ne règle pas le problème de fond, mais se contente d'un joli habillage, qui ne sera pas suivi d'effets concrets. Facebook, Apple, Twitter et Google sont des sociétés américaines, régies par le droit américain et même si elles font des efforts pour adapter leurs conditions générales d'utilisation aux États dans lesquels elles sont implantées, dans la majeure partie des cas, ce sont les règles américaines qui s'imposent. Prenons l'exemple de la politique concernant les contenus estampillés pornographiques chez Facebook : des photos représentants des peintures ou des seins ou des sexes dénudés sont immédiatement ou presque supprimées alors que les contenus racistes, nécessitant une maîtrise approfondie de la langue pour en détecter les subtilités, continuent à prospérer.

3 Un nouvel axe du Bien et du Mal ?

Le 23 février 2015 s'est tenu le traditionnel dîner du Conseil Représentatif des Institutions juives de France (CRIF), en présence de François Hollande. Durant son discours, il a annoncé que les propos de haine, incitant au racisme, à l'antisémitisme et à l'homophobie sortiraient du cadre de délit de presse pour relever du droit pénal exclusivement, que les effectifs de la plateforme PHAROS seraient renforcés (à ce jour, ils sont une quinzaine), et la mise en place rapide du PNR. Mais il a surtout dit cette phrase : « si vraiment les grands groupes d'internet ne veulent pas être complices du mal, ils doivent participer à la régulation » **[3]**.

Le problème d'un terme comme « mal » est qu'il est difficile à définir. Il ne relève pas du droit, il ne relève pas de l'éthique, il relève d'une morale qui est intrinsèquement liée à celui qui s'exprime. En utilisant ce type de vocabulaire, l'autorité du Chef de l'État est mise à mal, car il procède aux mêmes subterfuges que Georges W. Bush : utiliser des notions de morale pour justifier des actions dont on sait pertinemment qu'elles sortent du cadre légal. L'utilisation du verbe « devoir » renforce cette absence de légitimité juridique.

Que cela soit dit une fois pour toutes : Facebook, Google, Apple, Twitter, Microsoft et les autres ne sont pas nos amis, ce ne sont pas nos ennemis, ce ne sont pas nos collaborateurs, ce ne sont pas des complices. Ce sont des entreprises de droit privé, relevant du droit américain, dont l'objectif est d'être rentable. Le seul vecteur d'action qu'elles peuvent avoir est : telle action est-elle susceptible de me faire perdre ou de me faire gagner des clients ?

Même devant le droit américain, certaines entreprises refusent de céder, parce qu'elles savent que cela leur serait néfaste commercialement, comme Microsoft et son bras de fer avec la justice américaine concernant les informations d'utilisateurs, stockées en Irlande.

Poursuivant son discours, le chef de l'État a sobrement énoncé : « nous fixerons un cahier des charges clair et précis avec ces géants d'internet et je vous assure que nous contrôlerons son application ». Là encore, on se demande par quels moyens un État pourrait contrôler des entreprises privées étrangères. On pourrait penser à des sanctions économiques en France mais, en période de crise économique, avec 10% de la population au chômage, je ne suis pas persuadée que le Chef de l'État souhaite voir s'aggraver la situation.

Ces entreprises ne sont pas des services de police, ce ne sont pas des fonctionnaires, elles n'ont pas à suivre les positions et les souhaits d'un intervenant extérieur, peu importe la justification. Si ces mots avaient été employés envers un autre État, toute la scène internationale aurait hurlé à l'ingérence. Or, même envers le Pakistan, il n'y a jamais eu de termes aussi violents.

Sous couvert de lutter contre le terrorisme, nos politiques donnent dans l'habillage médiatique, la superficialité et le sensationnalisme. À ce jour, aucune mesure concrète concernant le trafic des armes, la politique de la ville, la réinsertion des délinquants, le réaménagement des établissements pénitentiaires, la réforme de la justice et la formation des magistrats, l'augmentation des effectifs de police sur le terrain, n'a été annoncée ni même esquissée. Les terroristes ont gagné. ■

Références

[1] D. Bouzar, C. Caupenne et S. Valsan avec l'aide de l'équipe du CPDSI, des familles et des partenaires, *« La métamorphose opérée chez le jeune par les nouveaux discours terroristes »*, novembre 2014 : http://www.bouzar-expertises.fr/images/docs/METAMORPHOSE.pdf

[2] Voir l'article ici : http://www.lesechos.fr/politique-societe/societe/0204173849803-cazeneuve-dans-la-silicon-valley-contre-lembrigadement-terroriste-sur-internet-1095247.php

[3] Voir les éléments de discours du chef de l'État ici : http://www.francetvinfo.fr/monde/terrorisme-djihadistes/diner-du-crif-hollande-veut-renforcer-l-arsenal-repressif_832611.html

LE PREMIER ORDINATEUR À CIRCUIT INTÉGRÉ EST ALLÉ SUR LA LUNE

par ***Pierre-Alexandre Voye*** *[Caméliste heureux]*

À l'orée des années 1960, l'informatique est encore une technologie très chère, difficile d'accès et exigeant des moyens que seules de grosses structures peuvent s'offrir. Le besoin de miniaturiser n'était pas forcément impérieux, la taille justifiant les marges de sociétés comme IBM. Le programme d'exploration lunaire Apollo va changer la donne en exigeant une forte miniaturisation et en finançant de fait la R&D sur les circuits intégrés, lançant l'industrie informatique vers une puissance de calcul sans cesse décuplée.

En 1960, le transistor n'est pas aussi mature que l'encombrant tube à vide. Mais l'invention de la photolithogravure, utilisée par Fairchild et Texas Instrument dès 1959, va permettre de rendre le transistor complètement fiable, et surtout de pouvoir imaginer en graver plusieurs côte à côte.

1 Des circuits intégrés dans un ordinateur, le programme Apollo

Le 4 septembre 1957, l'URSS lance en orbite un satellite de 80 kg, Spoutnik.

Pour James M. Gavin, le directeur de la recherche et du développement de l'armée, cet événement constituait un « Pearl Harbor technologique ». Les États-Unis étaient humiliés et l'Armée peu rassurée : la fusée R7 Semiorka était de fait un potentiel missile balistique. Il convenait de se lancer dans la course dans l'espace et rattraper le retard. Après avoir rechigné à s'adjoindre les services de Werner Von Braun et ses ingénieurs, auteurs de la funeste V2, ceux-ci firent fonctionner une fusée en moins de trois semaines, coiffant au poteau les ingénieurs américains qui y planchaient depuis plusieurs années : Von Braun et son équipe avaient déjà essuyé tous les problèmes d'ajustage pendant la guerre.

Le 25 mai 1961, suite à l'envoi du premier homme dans l'espace par l'URSS, John F. Kennedy définit l'objectif suprême : poser des hommes sur la Lune avant la fin de la décennie. Il réitère son discours, à destination de l'opinion publique, le 12 septembre 1962, au Rice Stadium, dans son fameux discours *« We choose to go to the moon »*.

1.1 L'ordinateur qui a posé un Homme sur la Lune

De fait, le programme Apollo était déjà démarré suite au discours du congrès en 1961. C'est ainsi que dès 1961, la NASA a commandé au MIT une étude d'un ordinateur généraliste capable d'être embarqué dans un véhicule spatial - comprendre raisonnable en taille et suffisamment puissant - l'Apollo Guidance Computer (AGC). Le débat n'était pas encore tranché : un ordinateur ? Pour quoi faire ? Avec quelle interface utilisateur à l'époque des cartes perforées ?

Malheureusement, la NASA ne fut pas très précise dans l'expression des besoins et des utilisations potentielles de l'AGC, ce qui mena l'équipe chargée de son développement à revoir 18 fois la taille de la mémoire, de sorte que le logiciel puisse y être stocké.

La conception hardware, dirigée par Charles Stark Draper, a été dessinée par Eldon C. Hall, J.H. Laning Jr., Albert Hopkins, Ramon Alonso, Hugh Blair-Smith. La conception de l'appareillage électronique dédié au vol est due à Herb Thaler **[1]**. La figure 1 montre l'AGC dans son caisson moulé.

1.2 Hardware

L'AGC est un ordinateur 16 bits, avec 1 bit de parité et 1 bit de signe. Comme souvent, encore dans les années 1960, c'est une machine à complément à 1. Il est exclusivement construit à partir d'assemblages de portes logiques NON-OU à 3 entrées, matérialisées par des puces Fairchild F321527 comprenant chacune deux portes (voir figure 2).

La mémoire à tore de ferrite **[2]** est accessible tous les 12 cycles de 1 Mhz, le cycle comprenant la lecture et la réécriture, la lecture étant destructive.

L'AGC comportait un nombre non négligeable de registres :

Nom du registre	Rôle
`A`	Registre accumulateur
`Z`	Program Counter : adresse de la prochaine instruction à exécuter
`Q`	Registre contenant le reste d'une division
`LP`	Stocke la partie haute du nombre calculé avec l'instruction de multiplication `MP`
`S`	Registre d'adresse 12 bits
`Fbank`	Registre 4 bits pour sélectionner la banque de 1ko de mémoire morte (soit 16ko adressable)
`EBank`	Registre 3 bits pour sélectionner la banque de 256 octets de mémoire vive
`SBank`	Extension d'1 bit à FBank
`SQ`	Program counter – instruction courante
`G`	Buffer 16 bits pour le transfert de données en mémoire
`X`	Registre d'opérande pour l'addition 1
`Y`	Registre d'opérande pour l'addition 2
`U`	Sortie de l'additionneur
`B`	Registre buffer d'instruction, ventilé entre `SQ` et `S` pour l'index donné dans l'instruction
`C`	Complément à 1 du registre `B`
`In`	Registre d'entrée 16 bits
`Out`	Registre de sortie 16 bits

La machine contenait 12 instructions de base, plus quelques instructions supplémentaires sans index dans chaque version embarquée sur les 17 missions Apollo. Le format d'instruction était de 3 bits pour l'instruction et 12 bits pour *l'index* mémoire fourni en paramètre. Voici les principales instructions, auxquelles s'ajoutent 29 autres instructions étendues :

Instruction	Description
`TC` (*transfer control*)	Branchement inconditionnel
`CCS` (*count, compare, and skip*)	Branchement conditionnel
`Index`	Ajoute la valeur contenue dans *index* à `Z`
`Resume`	Retourne à l'adresse d'exécution avant l'interruption
`XCH` (*exchange*)	Échange le contenu de la mémoire et du registre `A`
`CS` (*clear and subtract*)	Charge avec complément à 1 la valeur mémoire à *index*
`TS` (*transfer to storage*)	Sauve le registre `A` à l'adresse *index*
`AD` (*Add*)	Ajoute la valeur *index* à `A`
`MASK`	ET logique sur `A`, sauvegardé dans `A`
`MP` (*Multiply*)	Multiplie `A` par la valeur à *index* et sauve le résultat dans `A` et `LP`
`DV` (*Divide*)	Divise `A` par la valeur à *index* et sauve le résultat dans `A` et `Q`
`SU` (*subtract*)	Soustrait la valeur *index* à `A`

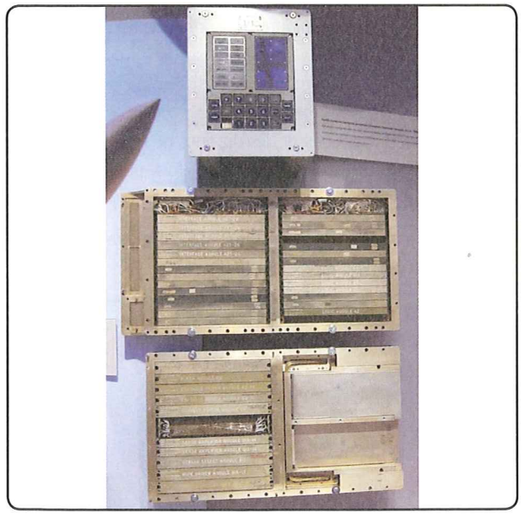


Fig. 1 : L'AGC dans son caisson moulé.

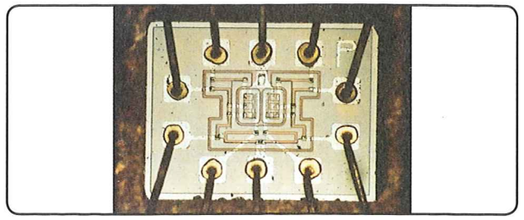


Fig. 2 : Le circuit intégré de base de l'AGC.

1.3 Une mémoire très limitée

La mémoire à tore de ferrite coûtait très cher et sa densité était limitée. Le coût étant un problème secondaire dans le faramineux programme Apollo, la taille et le poids étaient par contre cruciaux. La mémoire vive étant un composant spécial, elle devait être limitée.

La NASA dont le projet évoluait sans cesse, émettait régulièrement des nouvelles demandes quant aux services offerts par l'ordinateur. La mémoire logicielle nécessaire pour pouvoir disposer de tous les sous-programmes gonflait sans cesse.

L'accès à la mémoire était assez compliqué pour le programmeur : l'adresse d'index mémoire était de 12 bits, 10 bits servant à adresser le contenu d'une « banque » de 1024 mots de 16 bits (soit 2 Kio), et 2 pour définir la banque utilisée. La banque `0` correspondait à la mémoire vive, les banques `1` et `2` étaient de la mémoire morte toujours disponible. Quant à la banque `3`, elle faisait référence au registre `Fbank` qui permettait ainsi d'accéder à 16 384 mots de 16 bits, soit 32kio.

La construction de la mémoire morte utilisait une technique difficilement imaginable aujourd'hui : pour atteindre une densité maximale, on codait une série de bits sur un fil que l'on faisait passer ou non dans un tore de ferrite disséminé le long du trajet du fil. Si le fil passait dans le tore, on codait un `1`, sinon on codait un `0`. En faisant passer des dizaines de fils dans le même tore, on utilisait ainsi un tore jusqu'à 64 fois, pour obtenir une densité énorme pour l'époque !

Toute la mémoire morte dévolue au logiciel était ainsi « tissée » comme on peut le voir sur la figure 3.

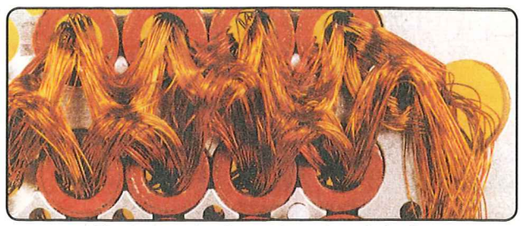


Fig. 3 : La mémoire morte en tore de ferrite.

1.4 Périphériques de mesures et d'interaction

Loin des classiques ordinateurs en mode batch ou *time sharing* de l'époque, l'AGC était connecté à quelques instruments de mesure dont il relevait à intervalle régulier les informations.

Cet ensemble de périphériques comprenait :

- Gyroscopes ;
- Accéléromètres ;
- Radars ;
- *Diskey* (voir plus loin) ;
- Contrôle des moteurs fusée ;
- Bouton d'annulation (*Abort Button*).

1.5 Interface utilisateur : un des premiers shell

Le *Diskey* (*Display Keyboard*) était un clavier spécial orienté ligne de commandes. Chaque programme, ou chaque commande était composé d'un verbe et d'un nom, chacun représenté par un code que les astronautes connaissaient sur le bout des doigts. L'interface proposait 50 programmes, 80 verbes et 90 noms.

Par exemple, lorsque l'astronaute voulait savoir dans combien de temps le moteur allait redémarrer, il tapait : `Verb 06` (*Display Decimal Data*), `Noun 33` (*Time of Ignition*) et appuyait sur la touche `<Enter>`. Vous pouvez jouer avec celui-ci sur un émulateur disponible en logiciel libre **[3]**.

Des coordonnées temporelles, numériques, dans diverses unités étaient affichées sur le *Diskey* en fonction des commandes entrées par l'utilisateur (voir figure 4).

L'ensemble de ces composants, une fois dûment testé (avec le logiciel tissé dans sa mémoire en tore de ferrite), était moulé dans de la résine et donc impossible à modifier, ce, dix mois avant le départ de la mission.

En phase de développement logiciel, un mini-ordinateur IBM servait de simulation de mémoire morte, voire de simulation tout court, pour les premières phases de développement.

1.6 Un logiciel au cordeau

Bien que le logiciel écrit pour le réseau SAGE (voir *GNU/Linux Magazine* de février 2015) était tout à fait stratégique, les logiciels de navigation des capsules Apollo et du module lunaire devaient être irréprochables : des vies et le prestige national étaient en jeu.

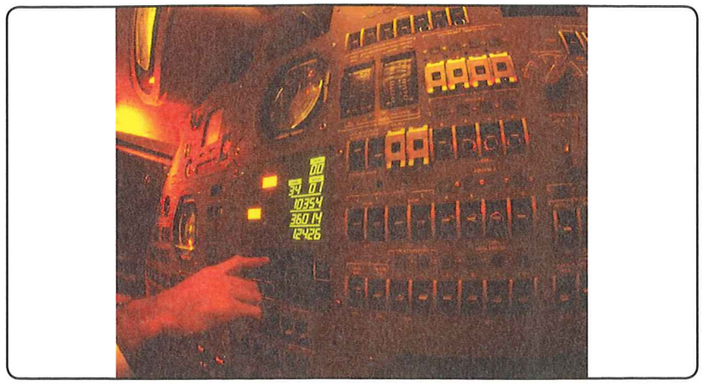


Fig. 4 : Le Diskey en cours d'utilisation.
À gauche du clavier, les touches Verb et Noun, à droite Enter et Reset.

Cela dit, de par la culture essentiellement tournée vers l'électronique, le logiciel était considéré comme quantité négligeable. La NASA mit peu de temps à découvrir son importance, et l'impérieux besoin de rigoureuses méthodologies de développement, se mit en place cahin-caha.

De plus, l'approche était inédite dans l'industrie informatique : les ressources de calculs étant rares, elles étaient toujours partagées entre de nombreux utilisateurs, et les contraintes temps réel peu nombreuses. Ici, l'ordinateur devait contrôler en temps réel un certain nombre de paramètres, mais qui plus est pouvoir réagir en temps réel aux commandes de l'astronaute qui utilisait la machine en tant qu'utilisateur unique, via le *Diskey*.

De par la taille limitée de sa mémoire morte et de sa mémoire vive, et le fait qu'une précision de 15 bits signés (on doit enlever le bit de parité) soit insuffisante, la seule façon de faire « tenir » tout le logiciel était d'utiliser une machine virtuelle. Ainsi, l'*Apollo Guidance Software* (AGS) était souvent écrit dans un assembleur virtuel, 32 bits. Ce langage virtuel similaire au *p-code* était à mi-chemin entre la bibliothèque de fonctions et la machine virtuelle, mais possédait ses propres registres (les VAC). Les instructions offertes étaient essentiellement des fonctions trigonométriques, matricielles, et de calculs à double voire à triple précision.

L'AGC, d'une puissance dix fois moindre qu'un IMB PC XT, devait réaliser un ensemble de tâches de contrôle et de navigation en temps réel. Plus particulièrement : piloter automatiquement la capsule entre la Terre et la Lune, calculer sa position par rapport aux étoiles et au soleil, gérer l'altitude au-dessus de la Lune en phase d'approche, intégrer les données RADAR, intégrer les données envoyées par télémétrie depuis la Terre, et répondre aux interactions utilisateurs sur le *Diskey*.

Le premier réflexe des ingénieurs fut de concevoir un système où chaque tâche avait un temps fixe alloué, passant de l'une à l'autre au-delà de ce temps. Ce système a été jugé peu fiable face aux erreurs, et peu flexible. On le verra, ce choix sauvera la mission.

Le système d'ordonnancement était un système qui gérait les interruptions, et les priorités de tâches. Une seule tâche temps réel pouvait fonctionner au même moment, mais jusqu'à 7 tâches pouvaient être ordonnancées. Une seule avait la main sur le *Diskey*.

Au niveau de la gestion mémoire, chaque tâche nécessitait une allocation. La granularité mémoire pour une tâche était un *Core Set* de 12 mots (soit 24 octets) contenant la priorité, l'adresse de retour et les variables temporaires, et les *VAC Area* de 44 mots, utilisés en cas de calculs vectoriels. Les programmes disposaient de 7 Core Sets et 7 VAC sur tout le système.

Toute tâche avait une priorité, la tâche *Diskey* ayant toujours la plus haute priorité. Toutes les 20 ms, le système élisait la tâche de plus grande priorité et l'exécutait. Néanmoins, il ne s'agissait pas de multitâche préemptif, mais coopératif : la tâche devait finir son travail. Un multitâche préemptif était émulé grossièrement, en vérifiant toutes les 0,67 secondes qu'une tâche ne bloquait pas le système, et redémarrait la machine si besoin.

Le système était constitué de *«waypoints»* qui permettaient au système de reprendre l'exécution d'un processus en cours lors d'un *restart*. Le *« restart »* vidait la queue de processus à exécuter, à moins que celui-ci soit considéré comme vital. Lorsqu'il survenait, si le nombre de processus dans la queue était trop grand, seuls les processus vitaux étaient conservés et de la mémoire se trouvait libérée. Si plusieurs versions du même job étaient dans la queue, fut-il vital, seul le plus récent était redémarré.

Ce système permettait de protéger la mission contre un redémarrage complet de la machine avec perte de données. Sans mémoire de travail autre que la mémoire vive, il était vital à l'ordinateur de conserver les paramètres de vols.

Les erreurs étaient affichées via des voyants ou sur le *Diskey* via les indicateurs de verbes et de noms. Les erreurs non bloquantes étaient interrogeables par l'utilisateur via des commandes appropriées.

Les logiciels étaient adaptés au déroulement de chaque mission, la machine disposant d'une horloge permettant de connaître le *Mission Elapsed Time* (MET). En fonction de ce MET et des différentes informations relevées sur les périphériques de

mesure, des programmes différents pouvaient se déclencher. Dès le lancement de la fusée, l'AGC était lancé suivant une séquence bien définie, et suivait les paramètres de vol de la fusée Saturn V.

Tout au long des 100 heures de voyage de la Terre à la Lune, l'ordinateur va aider les astronautes à gérer l'activité des moteurs, la bonne orientation de la capsule, la vérification des systèmes, etc. Les ingénieurs concepteurs du système méritent d'être salués, car il est encore très bien conçu selon les normes actuelles, et fut conçu et réalisé à une époque où tout restait encore à inventer dans ce domaine.

1.7 En descendant vers la Lune, un bug…

Le 21 juillet 1969 à 2 h 40 (MET 102:33:05), Neil Armstrong et Buzz Aldrin entament la descente vers la Lune, laissant Collins en orbite. Le LEM (module lunaire) doit suivre une trajectoire ressemblant à un logarithme **[4]**. Il est couché par rapport au sol de la Lune et plonge lentement vers elle en se redressant, pour alunir debout.

Armstrong tape la séquence **V37E63E** (**Verbe 37** – **<Enter>** – **63** – **<Enter>**) qui lance le programme **63**. Ce programme calcule le moment où le moteur doit être démarré et détermine une altitude et une vélocité théorique. Au moment où le moteur-fusée doit être démarré, une alarme retentit. Le contrôle de lancement reste dans les mains de l'astronaute. Mais l'AGC émet l'erreur **500** : il a calculé que le radar d'approche est dans une mauvaise orientation. Aldrin et Armstrong passent outre : peut-être un problème de mauvaise capture des données.

À T - 12 minutes , l'affichage du *Diskey* s'efface, pour réapparaître cinq secondes plus tard. L'ordinateur, cinq secondes avant le moment de lancer le moteur-fusée, affiche un **Go**. Aldrin appuie sur la touche *Proceed*. Le *Diskey* affiche alors une vélocité de 1 694,5 m.s-1 et une vitesse verticale de 0,67 m.s-1.

À T-9 minutes, Armstrong, redresse le LEM qui approchait de la Lune couchée. Mais la rotation prend trop de temps, et il réalise alors que le pilote automatique (un sous-programme de l'AGC) avait choisi 5 degrés par seconde comme valeur. Armstrong la redéfinit à 25 degrés par seconde. Le radar d'alunissage signale alors que les données sont cohérentes. Le programme **63** de l'AGC calculait la trajectoire théorique et devait être corrigé à la main par l'équipage, qui relevait périodiquement les valeurs calculées par le radar.

À T - 6mn, Aldrin rentre la séquence **Verb 16 Noun 68** dans l'ordinateur, une fonction calculant la différence entre l'altitude théorique et celle calculée par le radar. Le résultat est une erreur de 883 mètres, qui restait dans la marge d'erreur escomptée.

C'est à ce moment, qu'un *« program alarm »* retentit et que le *Diskey* affiche **Verb 06 Noun 63**. Aldrin rentre alors la séquence **Verb 5 Noun 9** pour connaître l'erreur, et **1202** s'affiche. L'ordinateur n'arrive plus à gérer l'ensemble des tâches temps réel qu'il doit supporter : il n'a plus assez de mémoire vive.

Aldrin informe Houston de l'erreur.

À Cambridge, au MIT, les informaticiens suivent en temps réel la progression de la mission, prêts à intervenir en cas de problème. Le chef du guidage informatisé, Steve Bales, 26 ans, a quelques secondes pour dire si la mission est abandonnée ou non.

Une seule tentative d'approche est possible. S'il répond *«Abort»*, la mission est terminée et tout le monde rentre sur Terre. Steve Bales lance un *« go »* approuvé par Jack Garman de la NASA et balbutié par Russ Larson du MIT, paralysé par le stress. En répondant *« go »*, le directeur de la mission Gene Krantz envoie l'ordre d'alunissage : on continue la mission !

30 secondes après l'alarme, le *«go»* de Houston parvient aux oreilles d'Aldrin. Durant cet intervalle, le contrôle de mission valide l'erreur d'altitude mesurée 3 minutes plus tôt.

Aldrin valide la correction et remesure l'erreur : 274 mètres s'affichent. Mais une alarme **1202** retentit de nouveau… Toujours le *« go »* de Houston.

À T – 6mn, le moteur s'allume plein gaz. Aldrin s'exclame « Le moteur plein gaz pile à temps ! ». Poussé à son maximum, le moteur a quelques problèmes, la pleine puissance ne dure pas longtemps. Le module lunaire s'approchant de la surface, Armstrong s'aperçoit qu'ils s'approchent beaucoup trop vite de la Lune, ce que l'ordinateur n'a pas prévu. Le moteur n'est plus à son maximum. D'un autre côté, la dépense de fuel aurait été trop importante et Armstrong obligé de décider de repartir par manque de réserve pour le retour.

À T - 4mn, le LEM est à 2200 mètres de la surface et file à 38 m.s-1 de vitesse verticale. Aldrin indique à l'ordinateur d'utiliser le programme d'approche n°**64**. L'ordinateur indique maintenant un *« landing point designator »* (LPD), l'angle d'approche du LEM par rapport à la surface, histoire d'alunir debout. Une des fenêtres du LEM possédait une fenêtre graduée permettant de viser le point d'impact en utilisant l'angle donné par l'ordinateur.

À T - 3mn, l'alarme **1201** retentit : *« No VAC areas available »*. 24 secondes plus tard, une erreur **1202** retentit. La mission peut être annulée à tout moment. 16 secondes plus tard, une nouvelle erreur **1202**. Houston ordonne de continuer.

Le cœur d'Armstrong passe de 120 à 150 pulsations minute.

Si tout s'était bien passé, Armstrong et Aldrin essaieraient bien de se rapprocher du point d'alunissage prévu. Mais il s'agit là de savoir si l'on continue ou non, sachant qu'occupés à piloter le LEM, ils ne peuvent vérifier que le site est sûr.

À T - 2mn30, Armstrong se rend compte que l'angle d'impact proposé par l'ordinateur va les faire atterrir sur une zone pleine de rochers. Il décide de passer en pilote manuel. La main droite sur le joystick de contrôle des moteurs, Armstrong rentre à la main gauche la séquence de lancement du programme **66** à T - 2mn. Ce programme se contente de gérer la puissance moteur en phase d'approche debout.

À 2h56, le LEM est posé sur le sol lunaire, on connaît la suite...

On l'a vu, en phase de descente vers la Lune, le système a émis plusieurs alarmes à intervalles réguliers. Ces alarmes sont graves, car elles impliquent un redémarrage (*restart*) du système, une petite perte de temps dans ses calculs et dans la cueillette des données... Heureusement, celui-ci est rapide et les processus vitaux retrouvent leur point d'exécution grâce aux *waypoints* disposés dans le code.

Le bug provient d'un problème de synchronisation entre pulsations envoyées par l'ordinateur pour l'acquisition de données, et la pulsation, à la même fréquence, d'un périphérique de mesure, l'ATCA (*Attitude and Translation Control Assembly*).

Le radar de *rendezvous* (voir figure 5), chargé de mesurer la distance du LEM avec le module de commande afin de gérer au mieux l'approche entre les deux capsules, était connecté soit à l'ordinateur du module lunaire (un clone de l'AGC), soit déconnecté de l'ordinateur et connecté à l'ATCA. Dans ce dernier cas,

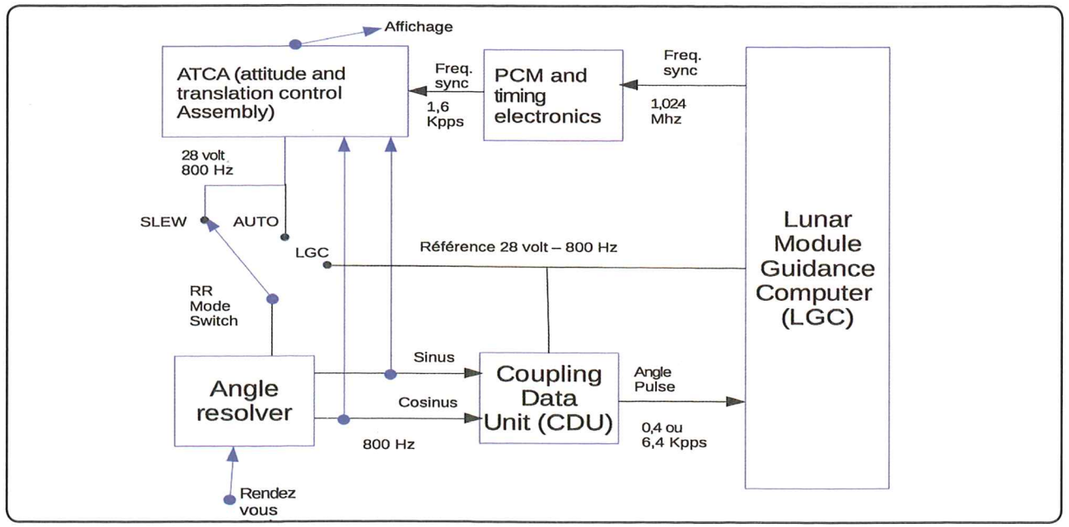


Fig. 5 : Schéma de principe du système de capture et d'analyse de données par le LGC.

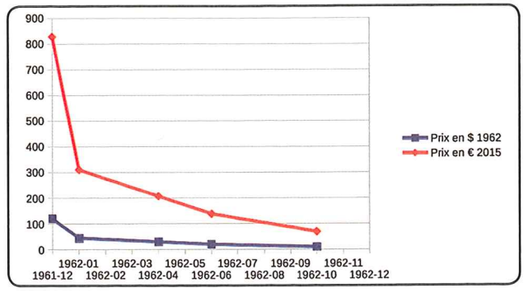


Fig. 6 : Évolution du prix unitaire des semi-conducteurs commandés par le MIT pour le compte de la NASA entre décembre 1961 et décembre 1962.

le radar utilisait comme référence la porteuse à 800Mhz fournie par l'ATCA (un système de mesure autonome avec affichage intégré). Or, dans les spécifications, le LGC et l'ATCA devaient échanger des informations sous 28 volts, 800 Hz, mais la spécification n'exigeait pas que la phase soit synchronisée ! Ainsi, lorsqu'en phase de descente, suivant la procédure de descente vers la Lune, Armstrong passe en *SLEW Mode*, et par conséquent, débranche le radar de la porteuse du LGC et la branche sur celle de l'ATCA, l'ordinateur continue à capturer les données sur celui-ci, mais une désynchronisation de phase apparaît.

À ce moment-là, le dispositif de numérisation des données, le CDU, intègre des données fausses de la part du *angle resolver*. Le LGC, essayant de retrouver le bon angle, intègre alors ces données 6400 fois par seconde, incrémentant ou décrémentant un compteur correspondant aux données du radar. Un cycle mémoire prenant 11,7 µs, c'est 10% de la charge du LGC qui se trouve occupée ! La machine surchargée n'a pas le temps d'intégrer d'autres données, et les processus non terminés s'accumulent, conduisant à un épuisement de la mémoire.

À chaque erreur **1201** ou **1202**, un *restart* est lancé, nettoyant la liste des processus en attente. Le bug était connu et documenté, mais trop tard pour être corrigé pour Apollo 11 : le logiciel et l'électronique sont fixés 10 mois avant le départ. Le bug sera corrigé dans les missions ultérieures.

Moralité, le très flexible système de processus inventé au MIT, qui a été (heureusement) préféré à un système de partage de blocs de temps fixe, ainsi que le très agile système de redémarrage a sauvé les astronautes d'un crash complet de l'ordinateur.

2 Circuits intégrés : l'avènement d'une industrie

En 1961, Fairchild SemiConductor n'était qu'une jeune start-up de la Silicon Valley, propriétaire avec Texas Instrument de la technique de photolithogravure, toujours utilisée aujourd'hui. La fabrication de circuits intégrés qui comprenait au mieux quatre transistors, était vendue $300 pièce. L'armée, occupée à la fabrication de son missile interbalistique *minuteman*, devint rapidement un client régulier de Fairchild dès la fin des années 1950.

La NASA devint aussi un gros client : occupée à fabriquer des AGC pour chacune de ses capsules, ce sont des dizaines de milliers de puces qui sont commandées. Doté d'argent pour investir en R&D et en amélioration de processus de fabrication, les prix vont fondre très vite passant de plusieurs centaines de dollars fin 1961 à la dizaine fin 1962 **[5]** (voir figure 6).

En 1969, les circuits intégrés ont déjà beaucoup progressé, proposant quelques centaines de transistors par puces, loin devant les 8 transistors par puces de l'AGC ! Outre les fonds importants mis à disposition, permettant à la R&D de travailler à plein régime, l'utilisation par la NASA des circuits intégrés a été une publicité formidable qui a eu raison des dernières résistances. L'industrie informatique commence alors sa progression exponentielle, surfant durant 50 ans sur la Loi de Moore.

Conclusion

Grâce à ses moyens quasi illimités et la nécessité de repousser la pointe de la technologie afin de rendre possible le rêve de fouler la Lune, le programme d'exploration spatiale Apollo a eu un grand rôle dans le développement de l'informatique moderne, en finançant l'avènement et la colonisation des circuits intégrés dans l'électronique, d'abord, mais aussi en apportant une expérience d'écriture de logiciels critiques, impliquant la nécessité méthodologique de définir des procédures rigoureuses de développement. Enfin, la nécessité d'écrire un système d'exploitation temps réel tolérant à la panne a fourni un bel exemple d'ingénierie réussie à l'industrie. ■

Références

[1] NASA, *« Computers in Spaceflight: The NASA Experience »* : http://history.nasa.gov/computers/Ch2-8.html

[2] R.-U. Hartmann, *« Core Rope Memory »* : http://drhart.ucoz.com/index/core_memory/0-123

[3] R. Burkey, *« Virtual AGC »* : http://www.ibiblio.org/apollo/index.html#Much_more_intensive_playing_with_the_LM

[4] D. Eyles, *« Tales from the lunar module guidance computer »* : http://www.doneyles.com/LM/Tales.html

[5] E. Hall, *« Integrated Circuits in the Apollo Guidance Computer »*, http://www.klabs.org/history/history_docs/integrated_circuits/ic4-po.pdf

LA COMPRESSION DANS TOUS SES ÉTATS

par ***Tristan Colombo***

Nous utilisons tous les jours des données compressées. Gzip, JPEG, PNG et autres PDF sont autant de formats de fichiers intégrant une compression. Il est devenu naturel de travailler avec de tels fichiers... mais vous êtes-vous déjà demandé comment les données initiales pouvaient tenir dans si peu de place ?

Il existe de très nombreux formats de compression et donc au moins autant d'algorithmes différents (qui parfois ne sont que des adaptations différentes d'un même algorithme). Pourtant chaque algorithme possède ses caractéristiques et ce n'est pas l'algorithme le plus efficace pour compresser des images qui sera utilisé pour du son ou du texte ! C'est en étudiant trois critères que l'on pourra déterminer l'efficacité d'un algorithme :

- son taux de compression bien sûr. Ce taux est obtenu en faisant le rapport de la taille du fichier compressé par la taille du fichier initial et il est donc inférieur à 1 (sinon votre algorithme a un très gros problème !). Par exemple, un taux de 0,3 indique que le fichier a été réduit de 70%.
- la qualité de la compression : certains algorithmes dégradent la qualité des données initiales.
- la vitesse de compression... et de décompression : il est bien pratique de manipuler des fichiers occupant moins de place sur les supports physiques, mais lors de l'utilisation des données il faut les décompresser !

Il existe trois types d'algorithmes de compression : les algorithmes statistiques, les algorithmes dynamiques et enfin les algorithmes heuristiques. Les formats de compression que nous utilisons quotidiennement ont recours aux deux premiers types pour lesquels nous étudierons un exemple concret. Mais pour commencer, voyons de manière générale quels peuvent être les problèmes rencontrés lors de la compression de données.

1 Les problèmes de la compression de données

Les données sont codées en machine en utilisant un système binaire. La table ASCII permet par exemple d'avoir la correspondance entre la valeur d'un octet (donc 8 bits) et une lettre. Dans cette table, la lettre `A` vaut `65` c'est-à-dire `01000001`. Or, s'il s'agit de coder un texte, nous savons que certaines lettres apparaissent plus fréquemment que d'autres (ce n'est pas pour rien que l'on gagne 10 points au Scrabble avec la lettre `W`). Donc si l'on trouve un système dans lequel on code sur moins de 8 bits les lettres « fréquentes », on devrait logiquement pouvoir compresser un fichier de texte assez simplement.

Cette approche fonctionne pour la compression. Supposons que nous déterminions le tableau de correspondance suivant :

Lettre	Fréquence	Code
D	3,75%	001111
U	6,25%	0011
R	7,25%	11

Pour écrire le mot `DUR`, nous n'aurons plus besoin de 24 bits (`3 x 8`), mais 12 bits (`6 + 4 + 2`). Nous obtenons un très bon taux de compression de 50% ! Voici la séquence de bits en machine qui correspondra au mot `DUR` : `001111001111`.

Si nous découpons notre code pour qu'il soit plus lisible, lors de la décompression nous retrouvons bien le mot initial :

```
001111 0011 11 = D U R
```

Mais en fait, rien ne nous indique qu'il faut couper le code après 6 bits puis encore après 4 bits... Nous pourrions également décompresser cette donnée en :

```
0011 11 0011 11 = U R U R
```

Un aspect essentiel de la compression est quand même de pouvoir retrouver les données de départ ! Cet exemple montre la nécessité de n'avoir aucune ambiguïté lors de la décompression (on parle de **code-préfixe** pour un codage dans lequel aucun code n'est le début d'un autre). Cette méthode naïve ne peut donc pas fonctionner. De plus, suivant les langues, la fréquence d'apparition de chaque lettre n'est pas la même. Notre méthode ne serait donc même pas universelle pour les fichiers texte.

D'un point de vue plus général, les étapes de compression et de décompression ne sont pas toujours équivalentes et, même si nous essayons de ne pas perdre d'informations entre ces deux opérations, cela n'est pas toujours le cas. Cela va nous amener à établir une classification parmi les algorithmes de compression.

1.1 Compression symétrique et asymétrique

Lorsque l'on utilise la même méthode pour compresser et décompresser les données, on parle de **compression symétrique**. Dans le cas contraire, bien entendu, il s'agira d'une **compression asymétrique** où l'une des deux opérations sera plus longue que l'autre à exécuter. Par exemple, pour des données auxquelles on accède souvent, mais qui seront peu modifiées, on préférera un algorithme rapide en décompression, quitte à ce que la compression, effectuée plus rarement, soit plus longue.

1.2 Compression sans/avec perte d'information

Les algorithmes de **compression sans perte** permettent de retrouver exactement les données de départ après avoir effectué un cycle de compression/décompression. Le problème de ces algorithmes c'est que leur taux moyen de compression (environ 40%) pour des données textuelles est insuffisant pour des données multimédia telles que des vidéos, des images ou encore du son. Dans ces cas, on permet la dégradation de l'information non perceptible à l'œil ou à l'oreille humaine pour obtenir un meilleur taux de compression. C'est le cas des formats MPEG, AVI, JPEG (un mode sans perte est disponible, mais rarement utilisé), etc. qui utilisent des algorithmes de **compression avec perte**.

Pour plus de lisibilité dans la suite, nous travaillerons exclusivement sur des données textuelles.

2 Un algorithme statistique : Huffman

En reprenant l'idée de notre algorithme naïf et en assurant la création d'un code-préfixe, nous allons obtenir l'algorithme de Huffman, algorithme utilisé en partie dans le format JPEG ou dans d'autres algorithmes de compression (gzip par exemple). Cet algorithme est un algorithme statistique asymétrique sans perte d'informations.

Comme précédemment, il va nous falloir calculer la fréquence d'apparition des différents caractères. Pour bien comprendre le processus, nous allons partir d'un exemple avant de présenter l'algorithme général. Considérons la phrase suivante : **Moi je lis GNU/Linux Magazine**. Cette phrase contient 29 caractères (bien entendu, on compte les espaces) et utilise donc 29 octets, soit 232 bits. Il y a 19 caractères différents. Pour obtenir la fréquence d'utilisation de chaque lettre, nous calculons un tableau de fréquences :

M	o	i		j	e	l	s	G	N	U	/	L	n	u	x	a	g	z
2	1	4	4	1	2	1	1	1	1	1	1	1	2	1	1	2	1	1

À partir de ces données, nous allons créer un arbre binaire par itérations successives. Au départ, nous aurons autant de nœuds que de caractères différents. À chaque nœud, nous associons une clé (le caractère) et une valeur (la fréquence). À chaque itération, nous sélectionnons les deux nœuds de plus faible valeur et nous formons un arbre binaire dont le nœud racine aura pour valeur la somme des valeurs des nœuds terminaux. Dans notre exemple, nous pourrions prendre par exemple `o` et `j` pour former l'arbre de la figure 1.

On construit ainsi par itérations successives un unique arbre binaire. Lorsque celui-ci est obtenu, il suffit de le parcourir en numérotant ses branches gauche `0` et ses branches droites `1` pour avoir un code-préfixe. Pour plus de

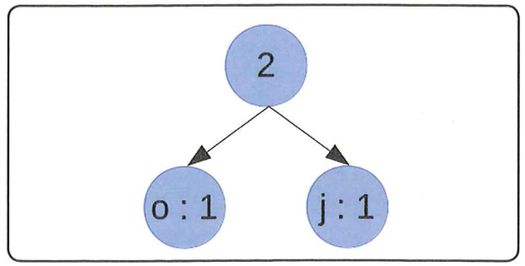


Fig. 1: Création d'un arbre binaire à partir de deux nœuds.

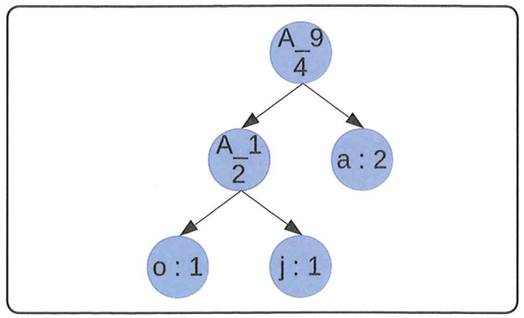


Fig. 3: Construction de l'arbre binaire (étape 9).

clarté, appliquons ce mécanisme sur notre exemple. Pour commencer, nous allons trier le tableau de fréquences par ordre croissant de manière à connaître les éléments les moins fréquents :

o	j	l	s	G	N	U	/	L	u	x	g	z	M	e	n	a	i	
1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	4	4

Lors de la première itération, nous allons obtenir un arbre composé des nœuds `o` et `j`. Nous appellerons son nœud racine `A_1`. Cet arbre étant déjà représenté en figure 1, je me contenterai de mettre à jour le tableau de fréquences :

l	s	G	N	U	/	L	u	x	g	z	M	e	n	a	A_1	i	
1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	4	4

Le prochain arbre à être créé sera celui composé de `l` et `s`. Nous le nommerons `A_2` (voir figure 2).

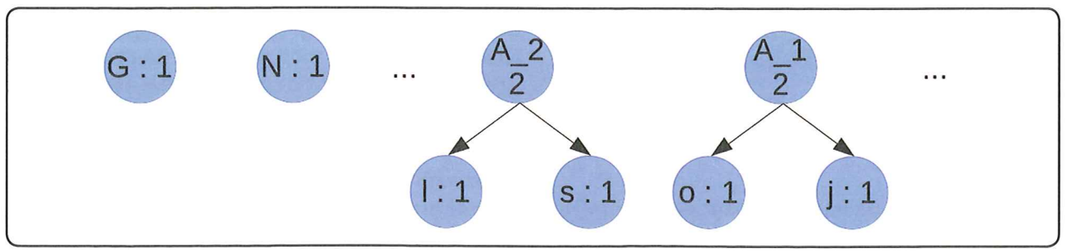


Fig. 2: Construction de l'arbre binaire (étape 2).

En sautant quelques étapes, nous arriverons jusqu'à la construction de `A_9` à partir des nœuds `a` et `A_1` (voir figure 3). Et ainsi de suite jusqu'à obtention de l'arbre final de la figure 4.

En parcourant cet arbre jusqu'à chaque nœud terminal nous pouvons lire le code binaire associé à chaque caractère :

Caractère	Code binaire
j	0000
N	0001
L	0010
M	0011
i	010
	011
n	1000
e	1001
a	1010
g	10110
u	10111
l	11000
G	11001
z	11010
/	11011
s	11100
U	11101
o	11110
x	11111

Ainsi, notre phrase initiale peut maintenant être codée sous la forme de 118 bits : `00111111001001100001001 01111000010111000111100100011110111011001001010001011111111101100111010101101010110100101000101001` (je vous laisse le soin de décompresser...). Le taux de compression est d'environ 50% (`118/232`).

Il faut souligner que pour la décompression nous devons fournir la table de codage, ce qui diminue forcément le taux de compression. Toutefois, en augmentant la taille du fichier nous obtiendrons tout de même en moyenne des taux de compression compris entre 30% et 60%.

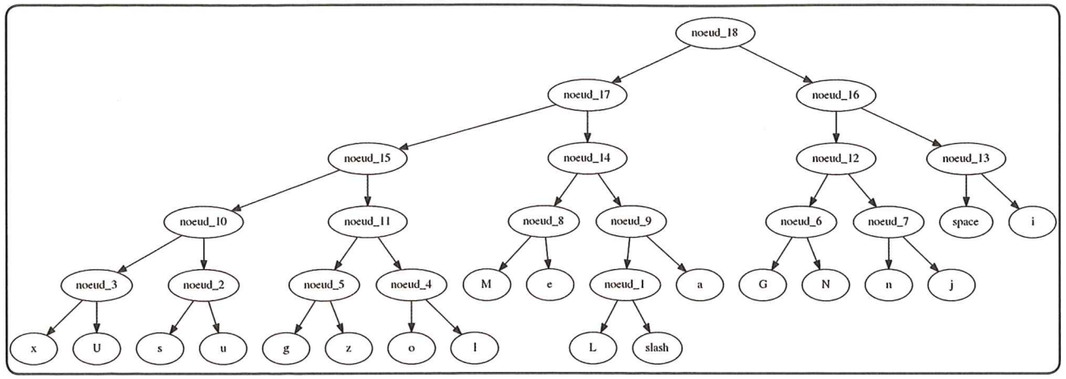


Fig. 4: Arbre binaire final (réalisé avec GraphViz).

Les arbres binaires

Un arbre binaire est un arbre dans lequel chaque nœud est soit connecté à deux autres nœuds soit à aucun (voir figure 5).

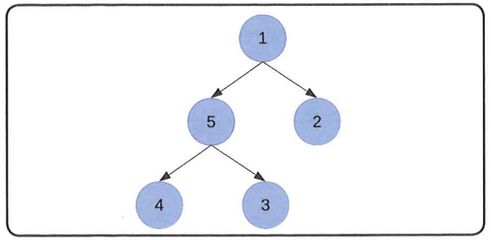


Fig. 5: Un arbre binaire.

2.1 Algorithme

Soit `T` un tableau contenant dans chaque cellule un champ `cle` (le caractère), un champ `valeur`, un champ `gauche` et un champ `droit` (pour la construction de l'arbre, initialement à `null`). La partie principale de l'algorithme consiste à générer l'arbre binaire permettant d'élaborer le code de chaque caractère :

```
01: Tant que taille(T) != 1 Faire
02:     Ordonner T suivant les T[i].valeur croissantes
03:     N1 <- T[0]
04:     N2 <- T[1]
05:     Retirer T[0] et T[1] du tableau
06:     Créer une cellule C telle que :
07:         C.cle <- 'noeud'
08:         C.valeur <- N1.valeur + N2.valeur
09:         C.gauche <- N1
10:         C.droit <- N2
11:     Ajouter C à T
12: Fin Tant que
```

Il faut ensuite être capable de parcourir cet arbre. Sans chercher d'optimisation, la manière la plus simple est d'opter pour un parcours infixe en programmation récursive (oui, si l'on recherche la performance il faudra passer un peu de temps sur cette partie pour la *dérécursifier*) :

```
01: Fonction parcours(arbre)
02:     Si arbre.gauche non vide Alors
03:         parcours(arbre.gauche)
04:     Fin Si
05:     Traitement infixe
06:     Si arbre.droit non vide Alors
07:         parcours(arbre.droit)
08:     Fin Si
```

2.2 Implémentation

Voici un petit exemple d'implémentation en Python. J'utiliserai ici un objet spécial : un `OrderedDict` qui est un dictionnaire ordonné (le dictionnaire « standard » est une structure non ordonnée).

```
01: from collections import OrderedDict
02:
03:
04: def tableau_freq(chaine):
05:     tab = {}
06:     for car in chaine:
07:         tab[car] = (tab.get(car, (0,))[0] + 1, None, None)
08:     return OrderedDict(sorted(tab.items(), key=lambda t:t[1]))
```

La ligne 1 permet de charger le module donnant accès aux dictionnaires ordonnés. Ensuite, nous segmentons le

problème : la première étape est de lire tous les caractères d'une chaîne et de compter le nombre d'occurrences de chaque lettre. Pour cela, j'utilise un dictionnaire « normal » (déclaré en ligne 5) et je parcours la chaîne passée en paramètre de la fonction `tableau_freq()`. Pour chaque caractère, s'il est absent du dictionnaire je l'ajoute et sinon j'augmente sa valeur. Chaque cellule du dictionnaire contiendra un tuple `(valeur, partie_gauche_arbre, partie_droite_arbre)`. Notez l'astuce de la syntaxe `tab.get(car, (0,))[0]` : la méthode `get()` renvoie la valeur du dictionnaire associée au premier paramètre qui lui est passé et si cette clé est absente, elle renvoie le deuxième paramètre. Ici `tab[car]`, s'il existe, est un tuple et sa valeur est contenue en position `0` (d'où le `[0]`). Pour pouvoir factoriser le code, le deuxième paramètre de `get()` est donc aussi un tuple (`(0,)`), ce qui permet de lui appliquer également l'index `[0]`. Pour finir, le dictionnaire est converti en `OrderedDict` et trié par valeurs croissantes.

```
11: def arbre_bin(chaine):
12:     T = tableau_freq(chaine)
13:     noeud = 1
14:
15:     while len(T) != 1:
16:         N1 = T.popitem(last=False)
17:         N2 = T.popitem(last=False)
18:         N  = (N1[1][0] + N2[1][0], N1, N2)
19:         T['noeud_'+str(noeud)] = N
20:         noeud += 1
21:         T = OrderedDict(sorted(T.items(), key=lambda t: t[1][0]))
22:
23:     return (T, noeud - 1)
```

Il faut ensuite construire l'arbre binaire à partir du tableau de fréquences ; c'est le rôle de la fonction `arbre_bin()`. En ligne 12, nous calculons ce fameux tableau et en ligne 13, nous initialisons la variable `noeud` à `1`. Cette variable va permettre de créer des labels pour les nœuds de l'arbre. Dans les lignes 15 à 21, nous appliquons ensuite l'algorithme vu précédemment : récupération des deux premières valeurs du tableau en lignes 16 et 17, création d'une nouvelle cellule en ligne 18 puis ajout au tableau en ligne 19.

À partir de cet arbre, il va falloir déterminer les codes associés à chaque caractère. Pour cela, j'ai codé deux fonctions :

```
26: def parcours(arbre_bin, codage, code='', verbose=False):
27:     if not arbre_bin[1][1] is None:
28:         parcours(arbre_bin[1][1], codage, code=code+'0')
29:     if not arbre_bin[0].startswith('noeud'):
30:         if verbose:
31:             print(arbre_bin[0], '=>', code)
32:         codage[arbre_bin[0]] = code
33:     if not arbre_bin[1][2] is None:
34:         parcours(arbre_bin[1][2], codage, code=code+'1')
35:
36:
37: def determine_code(arbre_bin, racine):
38:     codage = {}
39:     parcours(arbre_bin[racine][1], codage, code='0')
40:     parcours(arbre_bin[racine][2], codage, code='1')
41:     return codage
```

La fonction `parcours()` des lignes 26 à 34 correspond à l'algorithme de parcours infixe d'un arbre. La seule difficulté est de bien savoir quelles sont les données que l'on manipule de manière à ne pas se tromper dans les indices du tableau... Le code de chaque caractère est calculé et lors du parcours d'une branche gauche on y ajoute la valeur `0` alors que lors du parcours d'une branche droite on y ajoute la valeur `1`. Les codes sont stockés dans le dictionnaire `codage` qui est passé en paramètre.

La fonction `determine_code()` prend en paramètres l'arbre binaire et le label de sa racine. En effet, notre représentation peut poser problème pour le parcours de l'arbre et pour simplifier le traitement nous allons parcourir les branches gauche puis les branches droites en partant de la racine (lignes 39 et 40).

```
44: def compresser(chaine):
45:     A, num_racine = arbre_bin(chaine)
46:     codage = determine_code(A, 'noeud_' + str(num_racine))
47:     compresse = ''
48:
49:     for car in chaine:
50:         compresse += codage[car]
51:
52:     return compresse, {v: k for k, v in codage.items()}
```

La compression d'une chaîne n'est maintenant plus qu'une formalité : on calcule l'arbre binaire qui lui est associé (ligne 45), on détermine le tableau de codage (ligne 46), puis on crée la chaîne compressée en parcourant la chaîne initiale caractère à caractère et en insérant non le caractère, mais son code (lignes 47 à 50). En ligne 52, la fonction retourne la chaîne compressée ainsi que le tableau de codage inversé (les clés deviennent les valeurs et les valeurs deviennent les clés... ce sera plus simple pour la décompression).


```
55: def decompresser(chaine, codage):
56:     pos = 0
57:     courant = ''
58:     result = ''
59:
60:     while pos < len(chaine):
61:         courant += chaine[pos]
62:         car = codage.get(courant, None)
63:         if not car is None:
64:             result += car
65:             courant = ''
66:         pos += 1
67:
68:     return result
69:
70:
71: if __name__ == '__main__':
72:     c, codage = compresser('Moi je lis GNU/Linux Magazine')
73:     print(c)
74:     print(decompresser(c, codage))
```

La fonction de décompression ne présente pas de problème particulier et les lignes 71 à 74 permettent de tester le code. Bien entendu, nous nous contentons ici d'afficher sous forme de caractères les bits qui constituent la compression, mais pour une véritable exploitation il faudrait utiliser réellement des bits (voir le module **bitstring [1]**).

3 Un algorithme dynamique : LZW

L'algorithme LZW, est un algorithme dynamique de compression par dictionnaire. Son nom vient de ses créateurs Abraham Lempel pour le « L » et Jacob Ziv pour le « Z ». Les algorithmes LZ* ont vu plusieurs versions apparaître : LZ77, LZSS puis LZ78 plus connu sous le nom LZW et qui est une amélioration de LZSS proposée par Terry Welch (le « W » de LZW).

LZW est utilisé en association avec l'algorithme de Huffman dans le format gzip par exemple ou seul dans le format GIF. Cet algorithme est un algorithme dynamique asymétrique sans perte d'information.

Commençons là encore par un exemple. L'idée générale est simple : on va rechercher des séquences qui se répètent et associer un code unique à chacune d'elles (via un dictionnaire). La compression se fait ensuite en remplaçant les séquences par le code. Comme le dictionnaire est généré à la volée, le fichier compressé n'a pas besoin de le contenir (contrairement à l'algorithme de Huffman). Pour bien mettre en évidence cet algorithme sur une petite phrase d'exemple nous ne pourrons par réutiliser la phrase

employée avec l'algorithme de Huffman : il nous faut des séquences répétées. Voici donc notre nouvelle phrase : **titi itit titi**. On va lire caractère à caractère et construire le dictionnaire au fur et à mesure. Le tableau suivant montre les premières étapes de ce processus (le caractère _ sera utilisé pour représenter un espace) :

Caractères lus	Ajout dans le dictionnaire	Résultat de la compression
ti	t → 116 et ti → 256	116
it	i → 105 et it → 257	105
ti		
ti_	ti_ → 258	256
_i	_ → 32 et _i → 259	32
...	...	...

À la fin du processus, on obtient **116 105 256 32 257 257 32 256 256**. Vous vous interrogez sans doute sur la possibilité de représenter un code supérieur à **255** sur un octet... En fait il faut bien garder à l'esprit que la représentation utilisée ici est une simplification puisque les valeurs ne sont pas données en binaire et que l'on utilise le code ASCII pour représenter les caractères simples. En binaire, un code ASCII tient sur un octet, mais quand on dépasse **255**, il faut deux octets... ou au moins un bit de plus (donc 9 bits) pour pouvoir compter jusqu'à **511** (nous considérerons que 256 séquences suffisent et utiliseront donc 9 bits, mais en général on utilise plutôt 12 bits).

Pour calculer le taux de compression, il faut savoir que la phrase initiale comportait 14 caractères et tenait donc sur 14 octets soit 112 bits. Dans le résultat que nous avons obtenu, il faut penser à compter 9 bits par caractère. Le résultat est de 81 bits (**9 x 9**). Le taux de compression est donc d'environ 30% (**81/112**).

Plus le fichier sera long, plus il y aura de séquences et meilleur sera le taux de compression. Cette méthode est bien plus rapide qu'une méthode statistique puisque la compression se fait à la volée. Son autre avantage est le fait de ne pas avoir à transmettre de dictionnaire de codage pour la décompression et donc de ne pas alourdir un peu plus le fichier compressé.

3.1 Algorithme

Voici maintenant l'algorithme utilisé sur l'exemple précédent. Supposons que **C** soit la chaîne de caractères à compresser :

```
01: car <- C[0]
02: dico <- {}
03: code <- 256
04: Pour i variant de 1 à len(C) Faire
05:     suivant <- C[i]
06:     concat <- car + suivant (concaténation de car et de
suivant)
07:     Si concat est une clé de dico Alors
08:         car <- concat
09:     Sinon
10:         dico[concat] <- code
11:         code <- code + 1
12:         Écrire code associé à car
13:         car <- suivant
14:     Fin Si
15: Fin Pour
16: Écrire code associé à car
```

3.2 Implémentation

Dans notre implémentation nous n'allons encore pas utiliser des bits, mais des caractères qui nous permettront d'afficher les codes et de mieux comprendre comment fonctionne l'algorithme.

```
01: def compresser(chaine, verbose=True):
02:     car = chaine[0]
03:     dico = {}
04:     dico[car] = str(ord(car))
05:     code = 256
06:     resultat = None
07:
08:     for i in range(1, len(chaine)):
09:         suivant = chaine[i]
10:         if suivant not in dico:
11:             dico[suivant] = str(ord(suivant))
12:         concat = car + suivant
13:         if concat in dico:
14:             car = concat
15:         else:
16:             dico[concat] = str(code)
17:             code += 1
18:             if resultat is None:
19:                 resultat = dico[car]
20:             else:
21:                 resultat += ' ' + dico[car]
22:             car = suivant
23:
24:     resultat += ' ' + dico[car]
25:
26:     if verbose:
27:         for seq, code in dico.items():
28:             print('{:5s} => {:5s}'.format(seq, code))
29:
30:     return resultat
```

La fonction de compression est assez simple et suit scrupuleusement l'algorithme. Il faut utiliser la fonction **ord()** pour obtenir le code ASCII des caractères (lignes 4 et 11). En dehors de cela, nous parcourons la chaîne de caractères **chaine** passée en paramètre et construisons le dictionnaire pas à pas (lignes 11 et 16). C'est ce dictionnaire qui contient les correspondances de code avec les séquences obtenues par concaténation en ligne 12. Le paramètre **verbose** permet d'indiquer si l'on souhaite afficher le dictionnaire (lignes 26 à 28).

```
33: def decompresser(chaine, verbose=True):
34:     code = 256
35:     dico = {}
36:     data = list(map(int, chaine.split(' ')))
37:     car = chr(data[0])
38:     result = car
39:
40:     for i in range(1, len(data)):
41:         suivant = data[i]
42:         if suivant < 255:
43:             entree = chr(suivant)
44:         elif suivant in dico:
45:             entree = dico[suivant]
46:         elif suivant == code:
47:             entree = car + car[0]
48:         else:
49:             raise ValueError('Données mal compressées!')
50:         result += entree
51:
52:         dico[code] = car + entree[0]
53:         code += 1
54:
55:         car = entree
56:
57:     if verbose:
58:         for code, seq in dico.items():
59:             print('{:5d} => {:5s}'.format(code, seq))
60:
61:     return result
62:
63:
64: if __name__ == '__main__':
65:     c = compresser('titi itit titi')
66:     print(c)
67:     print(decompresser(c))
```

La méthode de décompression est très proche de la compression. Lorsqu'un code inférieur à **255** est lu, nous savons qu'il s'agit d'un code ASCII, donc il suffit d'effectuer une conversion grâce à la fonction **chr()** (ligne 43). Sinon, il faut chercher la signification du code dans le dictionnaire (ligne 45) sachant que si le code n'est pas présent, c'est que le fichier a été mal compressé (ligne 49). La création des séquences utiles pour la décompression se fait dans les lignes 47 et 52 (stockage dans le dictionnaire en ligne 52).

L'algorithme LZW ne faisant pas intervenir de structure complexe, il est plus rapide à écrire (bien qu'encore une fois la version que j'en donne puisse être optimisée). La partie qui peut éventuellement poser problème est la création des séquences dans la fonction de décompression (lignes 47 et 52). Si c'est le cas, pour bien voir ce qui se passe, je vous invite à dérouler un petit exemple à la main... Vous vous apercevrez qu'il n'y a rien de magique !

Conclusion

En fonction des données, un algorithme « maison » est parfois bien plus efficace qu'un algorithme éprouvé. En effet, certains jeux de données ont des caractéristiques très particulières qui, bien exploitées, permettent de très forts taux de compression et parfois même une utilisation sans décompression, en tant que données compressées.

Si nous prenons les bases azotées d'une chaîne d'ADN qui sont au nombre de 4 (**A**, **T**, **C** et **G**), il paraît évident qu'il y a beaucoup de place perdue à utiliser un octet pour stocker chaque base sur une séquence complète. En effet, deux bits sont largement suffisants ! De plus, d'un point de vue biologique les bases s'apparient deux à deux : **A** avec **T** et **C** avec **G**... et cela peut être représenté un peu comme un Lego ! Si **A** vaut **00** et **T** vaut **11**, **A** et **T** « s'emboîtent ». De même avec **C** qui vaut **01** et **G** qui vaut **10**. On peut même aller plus loin puisque si l'on veut obtenir le brin complémentaire (ie la séquence de bases « opposées »), il suffit de prendre le complément binaire de notre séquence compressée.

Le taux de compression est simple à calculer puisque sur un octet, au lieu de faire tenir une seule base, nous pouvons loger quatre bases. Le taux s'approchera donc de 75% (il faudra ajouter un octet au départ pour indiquer le nombre de bases présentes sur le dernier octet... les génomes n'ont pas forcément des tailles qui sont des multiples de quatre...). La méthode que nous aurons déterminée pour traiter ces données ne sera valable que pour celles-ci, mais ce sera également la plus efficace ! Comme toujours lorsque l'on travaille avec des données, il est essentiel de bien les comprendre... ■

Références

[1] Module bitstring : https://code.google.com/p/python-bitstring/

DES BITS COMME S'IL EN PLEUVAIT : UNE APPLIANCE SAN/NAS SOUS NETBSD

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

À Noël, je suis le désespoir de ma famille. En effet, là où d'autres demandent des Blu-ray, de quoi s'habiller en hiver ou encore une nouvelle poêle à frire, moi je demande des composants. Des disques, des cartes, des trucs super pas excitants pour les personnes qui vous les offrent. Cette année, je voulais plus de place sur mon NAS [1], plein de bits pour y coller des documents à haute teneur multimédia.

Mon NAS (voir en figure 1 un exemple), depuis quelques années, c'est un serveur NetBSD 6.1/amd64. Pas un foudre de guerre, un petit Intel Core 2 muni de 4Gi de RAM, qui me sert également de machine centrale de développement. Cette machine est relativement « sûre », comprendre que le système d'exploitation et le volume de stockage sont tous deux sur des grappes en RAID1. Le volume système est muni de deux disques de 500Gi, et le stockage sur deux disques de 2Ti à faible coût. J'ai souhaité doubler la quantité de données que je pouvais sauvegarder sur le volume de stockage afin d'amener mon réseau local à utiliser de plus en plus des facilités de *boot-on-nas* et éliminer ainsi les pannes potentielles de disques uniques dans les machines de mon LAN.

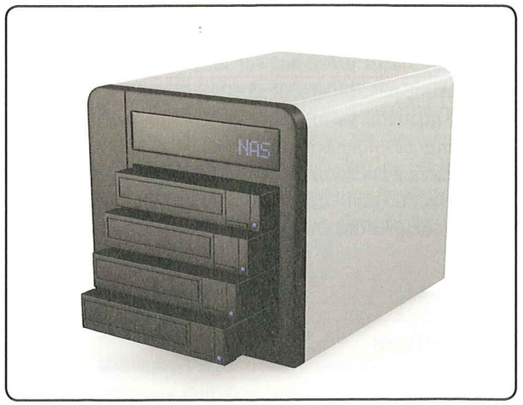


Fig. 1: Exemple de NAS.

1 RAIDframe

Mes machines d' « infrastructure perso » sont généralement sous NetBSD. À part l'affection particulière que je voue à l'ancestral UNIX Libre, c'est un système compact, fiable, sans surprises et respectant les standards de façon quasi religieuse. Les possibilités pour se construire un NAS performant articulé autour d'un système UNIX Libre sont multiples. Citons par exemple les capacités de GNU/Linux avec `mdadm` ou encore FreeNAS **[2]**, solution dédiée et de niveau professionnel architecturée autour de FreeBSD et ZFS. Ici nous allons utiliser une fonctionnalité du noyau NetBSD intégrée il y a plus de 10 ans : RAIDframe **[3]**.

RAIDFrame n'est pas un projet spécifique à NetBSD : initialement, il s'agit d'un logiciel de prototypage de structures RAID développé par le *Parallel Data Laboratory* à l'Université de *Carnegie Mellon*. L'objectif était de créer un simulateur de RAID pour un grand nombre d'architectures et un pilote pour Digital UNIX (*Tru64*) **[4]**. Ce dernier a été porté sous NetBSD par Greg Oster et fait partie des fonctions supportées par le noyau depuis les versions 1.4 (mai 1999 !).

Le driver RAIDframe supporte les niveaux de RAID suivants :

- RAID 0, ou *striping*, permettant d'agréger des disques entre eux et ainsi augmenter capacité et performances ;
- RAID 1, ou *mirroring*, dans lequel tous les disques contiennent les mêmes données, répliquées ;
- RAID 4, *striping* avec un disque dédié à la parité ;
- RAID 5, *striping* dans lequel la parité est répartie sur différents composants.

Le pilote est inclus dans le noyau par défaut depuis des années, on s'en assure grâce à la commande :

```
$ grep -i raidframe /var/run/dmesg.boot
Kernelized RAIDframe activated
```

L'outil permettant de manipuler ce driver est `raidctl` et fait partie du *basesystem* : inutile d'installer un quelconque paquet tiers.

2 Les équipes du RAID 1

2.1 Préparation de la grappe

Des 1001 scénarios imaginables dans la mise en place d'un système RAID, j'ai pris le parti de considérer une installation initiale mono-disque que l'on va transformer en système RAID. Il m'apparaît que c'est la probabilité la plus élevée, et elle nous fera manipuler plusieurs briques essentielles que vous pourriez être amené à rencontrer dans vos aventures NetBSDiennes.

En premier lieu, et considérant que les deux disques sont effectivement branchés, nous les identifions à l'aide de la séquence de démarrage suivante :

```
$ dmesg|grep ^wd
wd0 at atabus5 drive 0
wd0: <WDC WD5000AADS-00L4B1>
wd0: drive supports 16-sector PIO transfers, LBA48 addressing
wd0: 465 GB, 969021 cyl, 16 head, 63 sec, 512 bytes/sect x
976773168 sectors
wd0: 32-bit data port
wd0: drive supports PIO mode 4, DMA mode 2, Ultra-DMA mode 6
(Ultra/133)
wd1 at atabus6 drive 0
wd1: <WDC WD5000AADS-00L4B1>
wd1: drive supports 16-sector PIO transfers, LBA48 addressing
wd1: 465 GB, 969021 cyl, 16 head, 63 sec, 512 bytes/sect x
976773168 sectors
wd1: 32-bit data port
wd1: drive supports PIO mode 4, DMA mode 2, Ultra-DMA mode 6
(Ultra/133)
```

Nos deux disques sont bien pris en charge par le noyau : il s'agit de deux disques SATA, identifiés par le nom du pilote `wd` et un numéro. S'il s'était agi de disques SCSI, ils auraient été détectés avec un identifiant `sd[0-9]`.

Nous considérerons que le système d'exploitation est installé sur le premier disque (ce qui est très probablement le cas) `wd0`, et glanons quelques informations le concernant à l'aide de la commande `fdisk` :

```
# fdisk /dev/wd0
Disk: /dev/wd0d
NetBSD disklabel disk geometry:
cylinders: 969021, heads: 16, sectors/track: 63 (1008 sectors/
cylinder)
total sectors: 976773168

BIOS disk geometry:
cylinders: 1024, heads: 81, sectors/track: 63 (5103 sectors/
cylinder)
total sectors: 976773168

Partitions aligned to 5103 sector boundaries, offset 63

Partition table:
0: NetBSD (sysid 169)
    start 63, size 976773105 (476940 MB, Cyls 0-191411/44/63),
Active
1: <UNUSED>
2: <UNUSED>
3: <UNUSED>
Bootselector disabled.
First active partition: 0
```

Comme on peut le constater, NetBSD est installé sur la première partition du disque `wd0` ; aussi, nous allons effacer tout contenu potentiel de la table de partitions du disque 2, `wd1`.

Comme `rwd1d` représente l'ensemble du disque en mode *character device*, il faut exécuter :

```
# dd if=/dev/zero of=/dev/rwd1d bs=8k count=1
```

Puis créer et activer la même partition sur ce dernier :

```
# fdisk -0ua /dev/wd1
```

Les paramètres passés ici sont :

- **0** : nous allons agir sur la partition **0** ;
- **u** (*update*) : nous allons mettre à jour la partition indiquée ;
- **a** (*activate*) : nous « activons » la partition indiquée, si cela n'est pas fait notre système ne pourra pas démarrer.

Dès lors, les deux disques doivent être partitionnés de la même façon, on s'en assurera à l'aide de la même commande :

```
# fdisk /dev/wd1
Disk: /dev/wd1d
NetBSD disklabel disk geometry:
cylinders: 969021, heads: 16, sectors/track: 63 (1008 sectors/
cylinder)
total sectors: 976773168

BIOS disk geometry:
cylinders: 1024, heads: 81, sectors/track: 63 (5103 sectors/
cylinder)
total sectors: 976773168

Partitions aligned to 5103 sector boundaries, offset 63

Partition table:
0: NetBSD (sysid 169)
    start 63, size 976773105 (476940 MB, Cyls 0-191411/44/63),
Active
1: <UNUSED>
2: <UNUSED>
3: <UNUSED>
Bootselector disabled.
First active partition: 0
```

Reste maintenant à invoquer **diskabel**(8) afin de déclarer un volume RAID sur ce nouveau disque :

```
# disklabel -r -e -I wd1
# /dev/rwd1d:
type: ESDI
disk: WDC WD5000AADS-0
label: Disk1
flags:
bytes/sector: 512
sectors/track: 63
tracks/cylinder: 16
sectors/cylinder: 1008
```

```
# disklabel -r -e -I wd1
# /dev/rwd1d:
type: ESDI
disk: WDC WD5000AADS-0
label: Disk1
flags:
bytes/sector: 512
sectors/track: 63
tracks/cylinder: 16
sectors/cylinder: 1008
```

Les paramètres utilisés sont :

- **-e** qui fait apparaître un éditeur, celui spécifié dans la variable **${EDITOR}** de votre environnement. Si aucun éditeur n'est spécifié, c'est le vénérable **vi** qui apparaîtra ;
- **-r** qui permet de manipuler le disque directement plutôt que de passer par des requêtes **ioctl**(2) ;
- **-I** implique que si l'on ne détecte pas de label, on utilisera celui par défaut, fourni par le noyau.

Après avoir quitté l'éditeur, les changements seront inscrits dans le label du disque. Notre secteur d'amorce étant maintenant prêt, nous allons pouvoir passer à la préparation du volume RAID à proprement parler. Cette dernière prend tout d'abord la forme d'un fichier de configuration qu'on nommera de façon arbitraire **/var/tmp/raid0.conf**. Le nôtre ressemble à ceci :

```
START array
1 2 0

START disks
absent
/dev/wd1a

START layout
128 1 1 1

START queue
fifo 100
```

Quelques explications s'imposent. Chaque section du fichier de configuration démarre par le mot clé **START** suivi du nom de la section. Dans la première section, **array**, on définit le nombre de lignes, colonnes, et disques de *spare* (rechange) dans l'agrégat. Ici, nous définissons une ligne munie de deux disques sans disque de rechange. C'est une configuration RAID 1 assez classique.

Dans la section suivante, on déclare les disques qui composeront notre grappe. Notez la subtilité : le premier disque est déclaré **absent**, c'est à cette place que viendra

s'insérer le disque **wd0** lorsque notre RAID sera fonctionnel et que nous pourrons définitivement migrer dessus.

La section **layout** est la plus complexe. La première valeur décrit (traduction libre du man **raidctl**) :

« le nombre de secteurs par unités de stripe (bande). Comprendre, le nombre de secteurs contigus sur lesquels ont peut écrire dans chaque membre du RAID pour un seul stripe. »

Vous avez rien compris, pas vrai ? :)

Au début, moi non plus. Et je me suis contenté de copier bêtement le **raid0.conf** jeté en pâture dans la documentation officielle. Seulement voilà, ce paramètre peut influer de façon dramatique sur les performances de votre grappe ; en ayant fait les frais, j'ai épluché minutieusement les multiples fils sur le sujet.

Revoyons la scène au ralenti : **disklabel**(8) nous informe qu'un secteur est composé de 512 octets (0.5 Kibioctet, donc). Une valeur de 128 suppose donc 64k pour un seul disque. Dans le cas présent, nous ciblons un système de *mirroring*, et cette valeur n'influera pas les performances, car aucun calcul de parité ne viendra ralentir la réplication des données. Cependant, lors de la mise en place d'un système articulé autour du standard RAID 5, l'impact est tout autre ; en effet, un mauvais alignement du premier bloc ou un mauvais choix de taille de bloc lors de la création du système de fichiers impliqueraient un chevauchement des données sur plusieurs bandes et donc des recalculs de parité permanents.

Les deux valeurs suivantes, *stripe units per parity unit* et *stripe units per reconstruction* sont normalement toujours placées à **1**. Enfin, le dernier paramètre, le moins abscons, définit le type de RAID que nous mettons en place, ici **1**.

La dernière section de notre fichier de configuration décrit la méthode de file d'attente. Dans le cas présent, ainsi que dans tous les exemples qu'il m'a été donné de lire, la méthode est *fifo* avec une file d'attente limitée à 100 requêtes.

Munis de ces paramètres, nous allons pour la première fois invoquer **raidctl**(8) :

```
# raidctl -v -C /var/tmp/raid0.conf raid0
```

Cette commande configurera le périphérique **raid0** avec la configuration précédemment écrite. Lorsqu'une nouvelle grappe est déclarée, il est nécessaire de lui fournir un numéro de série qui sera utilisé afin de savoir si un composant fait partie d'une grappe. On peut par exemple utiliser la date du jour :

```
# raidctl -v -I 20140301 raid0
```

Puis nous instruisons le périphérique qui doit s'autoconfigurer au démarrage de la machine, sans présence de fichier **raid[0-9].conf** :

```
# raidctl -A yes raid0
```

Enfin, nous initialisons notre grappe :

```
# raidctl -v -i raid0
```

On peut à tout moment s'enquérir du statut des opérations ainsi que de l'état de notre grappe à l'aide de la commande suivante :

```
# raidctl -v -s raid0
```

2.2 Création du système de fichiers

Notre nouveau périphérique est désormais en ligne, et comme avec un disque physique, nous devons créer un label avec **disklabel**(8). Son partitionnement est évidemment à votre discrétion. Voici par exemple l'allure de mon système :

```
# disklabel raid0
# /dev/rraid0d:
type: RAID
disk: raid
label: fictitious
flags:
bytes/sector: 512
sectors/track: 128
tracks/cylinder: 8
sectors/cylinder: 1024
cylinders: 953879
total sectors: 976772992
rpm: 3600
interleave: 1
trackskew: 0
cylinderskew: 0
headswitch: 0           # microseconds
track-to-track seek: 0  # microseconds
drivedata: 0
```

```
16 partitions:
#        size    offset     fstype [fsize bsize cpg/sgs]
 a:  20971520         0     4.2BSD      0     0     0  # (Cyl.
0 -  20479)
 b:   8388608  20971520       swap                    # (Cyl.
20480 -  28671)
 d: 976772992         0     unused      0     0        # (Cyl.
0 - 953879*)
 e: 104857600  29360128     4.2BSD      0     0     0  # (Cyl.
28672 - 131071)
 f:  20971520 134217728     4.2BSD      0     0     0  # (Cyl.
131072 - 151551)
 g: 821583744 155189248     4.2BSD      0     0     0  # (Cyl.
151552 - 953879*)
```

Ici :

- **a:** correspond à **/** et pèse 10GB ;
- **b:** est la partition de **swap** et pèse 4GB ;
- **c:** et **d:** sont réservés et ne peuvent être utilisés ;
- **e:** correspond à **/usr** et pèse 50GB ;
- **f:** correspond à **/var** et pèse 10GB ;
- **g:** correspond à **/home** et pèse 400GB.

Notez que les tailles, dans **disklabel**(8) sont spécifiées en bloc, ces derniers faisant 512 octets chacun, il faut multiplier par deux la valeur souhaitée. Par exemple, 10GB correspond à **10*2*1024*1024**, soit 20 971 520 blocs.

Une fois le périphérique labellisé, on crée tout naturellement un système de fichiers en invoquant la commande **newfs**(8) :

```
# newfs -O 2 -b 8192 -f 1024 /dev/raid0a
# newfs -O 2 -b 8192 -f 1024 /dev/raid0e
# newfs -O 2 -b 8192 -f 1024 /dev/raid0f
# newfs -O 2 -b 8192 -f 1024 /dev/raid0g
# newfs -O 2 -b 8192 -f 1024 /dev/raid0h
```

Le système de fichiers choisi est FFSv2, avec une taille de bloc de 8k et une taille de fragment conseillée de **8:1**, soit 1k.

Le premier disque de la grappe **raid0** est prêt à être monté :

```
# mount /dev/raid0a /mnt
# mount /dev/raid0e /mnt/usr
# mount /dev/raid0f /mnt/var
# mount /dev/raid0g /mnt/home
```

Et l'on peut commencer à le peupler de données qu'il partagera bientôt de façon synchrone avec son voisin. Nous allons dans un premier temps synchroniser manuellement les données présentes sur le disque dans lequel est actuellement posé le système en cours d'utilisation vers le second disque que nous venons de configurer. J'utilise personnellement pour réaliser ce type de copie en masse le couteau suisse **rsync**, mais des logiciels comme **pax**(1) ou encore le couple **dump**(8) / **restore**(8) conviendraient tout à fait. On trouve d'ailleurs dans la littérature en ligne de nombreux exemples d'utilisation des outils en question dans le même contexte.

Le nom du périphérique ne sera plus le même, il est donc indispensable de modifier le fichier **/etc/fstab** afin d'y remplacer **wd0** par **raid0**. Afin d'éviter les erreurs de parité sur le périphérique RAID, nous faisons en sorte que le swap soit déconfiguré au moment de l'arrêt de la machine. Ceci est fait par une simple directive dans le fichier **/etc/rc.conf** :

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

L'ultime étape à effectuer pour rendre notre premier disque système RAID parfaitement opérationnel est de l'affubler d'un secteur d'amorce, i.e. de rendre ce disque *bootable*. On réalise ceci à l'aide de la commande **installboot**(8) (oui, dans le monde BSD on aime bien les noms compliqués) :

```
# /usr/sbin/installboot -o timeout=30 -v /dev/rwd1a /usr/mdec/
bootxx_ffsv2
```

Notez que la partition accueillant le noyau a bien été formatée en FFSv2, le système de fichiers par défaut depuis NetBSD 6. Le moment est venu de démarrer notre système sur le nouveau disque RAID. ATTENTION, ne redémarrez pas votre système RAID à l'aide de la commande **reboot**(8), : celle-ci n'appelle pas correctement les *RC scripts*. Préférez-lui la commande UNIX **shutdown -r now**.

Afin de démarrer sur le second disque RAID, il sera nécessaire de temporairement modifier l'ordre de boot dans votre BIOS/UEFI et de déclarer le disque numéro deux comme disque de boot.

2.3 L'ajout du second disque

Vous devez avoir démarré sur le second disque, et en particulier sur le périphérique RAID logiciel. On s'en assure à l'aide de la commande :


```
# grep -Ei 'raid|root' /var/run/dmesg.boot
raid0: RAID Level 1
raid0: Components: component0[**FAILED**] /dev/wd1a
raid0: Total Sectors: Total Sectors: 976772992 (476939 MB)
boot device: raid0
root on raid0a dumps on raid0b
root file system type: ffs
```

On peut constater que le composant **0** est marqué **FAILED**, ce qui est tout à fait normal, puisque pour le moment notre RAID 1 ne dispose que d'un seul disque. Nous allons remédier à cela immédiatement. En premier lieu, il s'agit de relabelliser le disque numéro un avec strictement les mêmes informations que le disque deux. Ceci est très simplement fait par **disklabel**(8) :

```
# disklabel /dev/wd1 > disklabel.wd1
# disklabel -R -r /dev/wd0 disklabel.wd1
# disklabel /dev/wd0
```

Dans l'ordre :

- On exporte le fichier de label du disque #2.
- On l'inscrit directement dans le disque #1.
- On vérifie que les informations correspondent.

On ajoute alors le disque un dans le *RAID set* en tant que remplacement à chaud (*hot spare*) :

```
# raidctl -v -a /dev/wd0a raid0
```

Et on commande une reconstruction sur un *hot spare* disponible :

```
# raidctl -F component0 raid0
```

Cette dernière opération va prendre... plusieurs heures. Notez que votre système est parfaitement utilisable pendant cette période, mais plus vous y apporterez des modifications, plus la reconstruction sera longue ! Vous pouvez à tout moment vous enquérir de l'évolution de la reconstruction à l'aide de la commande suivante :

```
# raidctl -S raid0
Reconstruction is 0% complete.
Parity Re-write is 100% complete.
Copyback is 100% complete.
Reconstruction status:
  14% |*****                                  | ETA: 04:02 -
```

Lorsque la reconstruction est terminée, il sera judicieux d'installer également un secteur d'amorce sur le premier disque :

```
# /usr/sbin/installboot -o timeout=15 -v /dev/rwd0a /usr/mdec/
bootxx_ffsv2
```

Et de remodifier l'ordre des périphériques de boot dans votre BIOS.

> **Note**
>
> N'oubliez pas : pas de **reboot** ni de **shutdown -r now** !

Félicitations, votre système d'exploitation est maintenant (un peu plus) sécurisé à l'aide d'un RAID 1.

3 Et mes films de vacances qualité HD ?

Par souci de flexibilité, j'ai souhaité distinguer système et données, aussi ne vais-je pas transformer notre système RAID 1 en RAID 5. Cela, à l'aide de nos nouveaux disques, ne serait pas plus complexe que ce que nous avons mis en œuvre dans le paragraphe précédent.

Mon *setup* initial était composé de deux disques de 500Gi dédiés au système ainsi qu'à mon *home directory*, et de deux disques de 2Ti destinés à accueillir des médias (musique libre, films de vacances, documentations sous Creative Commons...) également configurés en mode RAID 1. Finalement, la qualité des caméras équipant les téléphones mobiles devenant incroyablement bonne, nos photos et autres panoramiques de plage occupent une place de plus en plus grande et nos 2Ti se remplissent bien plus vite que je ne le prévoyais. Aussi ai-je demandé à Noël non pas un, mais deux disques de même marque et de même taille afin de transformer mon simple volume média RAID 1 en RAID 5.

Afin d'assurer une certaine sécurité dans mon NAS, j'ai décidé de dédier trois disques aux données effectives et un *hot spare*, inutilisé donc, simplement présent en cas de défaillance de l'un de ses compères. Trois disques en RAID 5 cela nous donne 4Ti utiles. Cette taille nous posera un problème un peu plus tard, nous y reviendrons.

Le disque *hot spare* n'étant pas réellement utilisé, je m'en suis servi comme tampon. En effet, ce dernier a servi de stockage temporaire des données présentes sur la grappe média en RAID 1 le temps de sa transformation en RAID 5. Là encore, c'est le logiciel **rsync** qui m'a permis de faire l'aller-retour des données.

Une fois nos données transférées dans ce stockage tampon, nous pouvons procéder à la préparation de la nouvelle grappe. De façon identique à notre précédente manipulation, nous écrasons un éventuel label précédent et créons un nouveau label sur les trois disques afin de les déclarer comme volumes RAID :

```
# dd if=/dev/zero of=/dev/rwd2d bs=8k count=1
# disklabel -r -e -I wd2
# # on remplace le type de système de fichiers 4.2BSD par RAID
# # on répète ces deux opérations pour tous les disques à
inclure
```

Nous n'aurons pas besoin ici de déclarer de table de partitions sur les disques physiques puisque nous ne démarrerons jamais sur cette grappe destinée au stockage brut. On poursuit la mise en place avec la création d'un fichier de configuration que nous appellerons par exemple **raid1.conf** :

```
START array
# numRow numCol numSpare
1 3 0

START disks
/dev/wd2a
/dev/wd3a
/dev/wd4a

START layout
# sectPerSU SUsPerParityUnit SUsPerReconUnit RAID_level_5
32 1 1 5

START queue
fifo 100
```

Notons quelques changements par rapport à notre précédente configuration :

- Cette fois nous disposons de trois disques, déclarés comme trois colonnes.
- Les trois disques sont décrits dans la section **disks**.
- Le **layout** précise qu'il s'agit de RAID 5 et on notera avec attention la valeur de la fameuse variable **sectPerSU**. Pour cette dernière valeur, il est important de comprendre que le calcul est le suivant : **32+32+0**. Soit deux fois 32 secteurs : un par disque contenant de la donnée d'une bande puisque le troisième, pour la même bande, contiendra uniquement les informations de parité. Finalement, ce sont 64 secteurs, donc 32k, qui seront occupés. Notez que la valeur du paramètre **sectPerSu** n'est pas une vérité absolue, il est de bon aloi de réaliser quelques tests de performance avant de graver cette valeur dans le marbre. Il est cependant globalement établi que pour un stockage RAID 5 composé de 3 disques, une valeur de 32 ou 64 est généralement conseillée.

De façon tout à fait similaire au disque système, nous créons la nouvelle grappe :

```
# raidctl -v -C /var/tmp/raid1.conf raid1
# raidctl -v -I 20140306 raid1
# raidctl -A yes raid1
# raidctl -v -i raid1
```

Il est temps de créer la table de partitions du périphérique RAID. Un périphérique d'un fort beau gabarit, puisque ce dernier pèse 4Ti, donc plus de 2Ti. Et il se passe quoi à plus de 2Ti ? Il se passe que **disklabel**(8), tout comme l'ancestral **fdisk** sont impuissants, bridés qu'ils sont par la limitation introduite par IBM en 1981 pour qui dépasser des unités de stockage de 2Ti paraissait inconcevable. Il existe bien heureusement un outil permettant de contourner cette limitation et utiliser une norme plus récente, GPT **[5]**, qui repousse les possibilités de stockage à 9.4Zi. Faites pas les malins, on y arrivera tôt ou tard. L'outil en question se nomme - attention on a fait fort - **gpt**(8). Oui je sais ça fait beaucoup de commandes aux noms très compliqués à retenir.

Son utilisation est relativement triviale. On crée une table de partitions vide :

```
# gpt create raid1
```

Et on ajoute une partition qui commence au secteur qui suit le **flag -b**, comme vous l'aurez deviné, la valeur qui suit ce *flag* est le premier secteur de la partition, dont nous souhaitons évidemment qu'elle démarre sur un multiple de notre fameux **sectPerSU**, soit **128** :

```
# gpt add -b 128 raid1
```

La sortie de **gpt**(8) vous indiquera la taille de la partition. Copiez-la. Ici, elle vaut **7814058015**. Afin de nommer et typer notre partition de stockage, nous utilisons le programme **dkctl**(8) et créons un *wedge* (traduction anglaise de morceau) :


```
# dkctl raid1 addwedge export 128 7814058015 ffs
```

Ici on spécifie le nom du périphérique, l'action à y mener, le bloc de démarrage, le nombre de blocs et le type de partition. Pour rendre ce nouveau média utilisable, on l'affuble d'un système de fichiers dont on s'assure que la taille de blocs correspond à la valeur **sectPerSu** définie dans le fichier **raid1.conf**, soient 64 secteurs fois 512 octets :

```
# newfs -O2 -b32k -I dk0
```

Nous pouvons alors tenter de monter cette nouvelle grappe :

```
# mkdir /export
# mount /dev/dk0 /export
```

Et si tout se passe correctement, l'ajouter à votre fichier **/etc/fstab** avec quelques options bien senties :

```
/dev/dk0 /export ffs rw,log,noatime 0 0
```

Note

Attention : ne *jamais* cumuler l'option **log** (journalisation) avec l'option **async**, les deux sont parfaitement antagonistes et leur juxtaposition pourrait simplement amener à un état totalement instable de votre système de fichiers.

Un **shutdown -r now** plus loin, votre monstre à données est fin prêt à accueillir un nombre de bits difficile à concevoir.

4 « Et là, c'est moi à la plage »

Quatre téraoctets sous le coude, il va en falloir des documents à forte teneur multimédia pour en venir à bout. Et justement, comment allons-nous mettre cette débauche d'inodes à disposition de notre réseau local ? Plusieurs options s'offrent à nous, toutes bien évidemment supportées par NetBSD depuis des lustres :

- Présenter un fichier brut comme périphérique de type bloc au réseau via iSCSI **[6]**, en mode SAN donc ;
- Exposer un ou plusieurs répertoires en CIFS/SMB **[7]** via Samba **[8]**, en mode NAS ;
- Exposer un ou plusieurs répertoires via NFS **[9]**, également en mode NAS.

Pour la première option, une longue prose est inutile, la mise en place d'une cible iSCSI sous NetBSD est triviale. On renseigne le fichier **/etc/iscsi/targets** de cette façon :

```
# extent        file or device          start           length
extent0         /export/iscsi/target0     0               1000MB

# target        flags   storage         netmask
target0         rw      extent0         192.168.1.0/24
```

On active le service dans le fichier **/etc/rc.conf** :

```
# echo "iscsi_target=YES" >> /etc/rc.conf
```

Et on démarre le service :

```
$ sudo /etc/rc.d/iscsi_target start
Starting iscsi_target.
Reading configuration from `/etc/iscsi/targets'
target0:rw:192.168.1.0/24
        extent0:/export/iscsi/target0:0:1048576000
DISK: 1 logical unit (2048000 blocks, 512 bytes/block), type
iscsi fs
DISK: LUN 0: 1000 MB disk storage for "target0"
TARGET: iSCSI Qualified Name (IQN) is iqn.1994-04.org.netbsd.
iscsi-target
```

Le « disque » est créé automatiquement, on vérifie la disponibilité du disque distant, par exemple sur un poste GNU/Linux, en ayant préalablement installé le paquet **open-iscsi** et démarré le service **/etc/init.d/open-iscsi** :

```
imil@tatooine:~$ sudo iscsiadm -m discovery -t sendtargets -p
192.168.1.2:3260
192.168.1.2:3260,1 iqn.1994-04.org.netbsd.iscsi-target:target0
```

On ouvre alors une session sur la cible :

```
imil@tatooine:~$ sudo iscsiadm  -m node  --targetname "iqn.1994-
04.org.netbsd.iscsi-target:target0" --portal 192.168.1.2 --login
Logging in to [iface: default, target: iqn.1994-04.org.netbsd.
iscsi-target:target0, portal: 192.168.1.2,3260] (multiple)
Login to [iface: default, target: iqn.1994-04.org.netbsd.iscsi-
target:target0, portal: 192.168.1.2,3260] successful.
```

Ce qui provoquera l'apparition d'un nouveau périphérique de type bloc :

```
Mar  8 20:50:30 coruscant iscsi-target: > iSCSI Discovery
login  successful from iqn.1993-08.org.debian:01:9137d4d18f9d on
192.168.1.1 disk -1, ISID 9613344768, TSIH 8
Mar  8 20:54:11 coruscant iscsi-target: > iSCSI Normal login
successful from iqn.1993-08.org.debian:01:9137d4d18f9d on
192.168.1.1 disk 0, ISID 9613410304, TSIH 9
Mar  8 20:56:42 tatooine kernel: [4862352.226463] scsi78 : iSCSI
Initiator over TCP/IP
Mar  8 20:56:42 tatooine kernel: [4862352.480458] scsi 78:0:0:0:
Direct-Access     NetBSD   NetBSD iSCSI     0    PQ: 0 ANSI: 3
Mar  8 20:56:42 tatooine kernel: [4862352.480604] sd 78:0:0:0:
Attached scsi generic sg5 type 0
Mar  8 20:56:42 tatooine kernel: [4862352.481600] sd 78:0:0:0:
[sdf] 2048000 512-byte logical blocks: (1048 MB/1000 MiB)
Mar  8 20:56:42 tatooine kernel: [4862352.481957] sd 78:0:0:0:
[sdf] Write Protect is off
Mar  8 20:56:42 tatooine kernel: [4862352.486505]  sdf: unknown
partition table
Mar  8 20:56:42 tatooine kernel: [4862352.488948] sd 78:0:0:0:
[sdf] Attached SCSI disk
```

Ce dernier est alors manipulable de la même façon qu'un classique disque physiquement attaché à votre station. Il est ici identifié comme **/dev/sdf** et automatiquement logiquement lié à : **/dev/disk/by-path/ip-192.168.1.2:3260-iscsi-iqn.1994-04.org.netbsd.iscsi-target:target0-lun-0**. Il faudra, pour utiliser ce disque SCSI en réseau, le manipuler avec les outils habituels : **fdisk** et **mkfs.<système de fichiers>**.

La seconde option de partage consiste en la mise en place du mode NAS/CIFS. Elle se traduit par l'installation du paquet **samba** sur la machine de stockage. On ajoute naturellement les directives de démarrage automatique du service dans le fichier **/etc/rc.conf** :

```
# cat >> /etc/rc.conf << EOF
smbd=YES
nmbd=YES
EOF
```

Puis on démarre le serveur Samba ainsi que le serveur de noms NetBIOS :

```
# /etc/rc.d/nmbd
# /etc/rc.d/smbd
```

Le logiciel étant livré avec l'interface SWAT, la création des partages peut se faire aisément en se connectant sur le

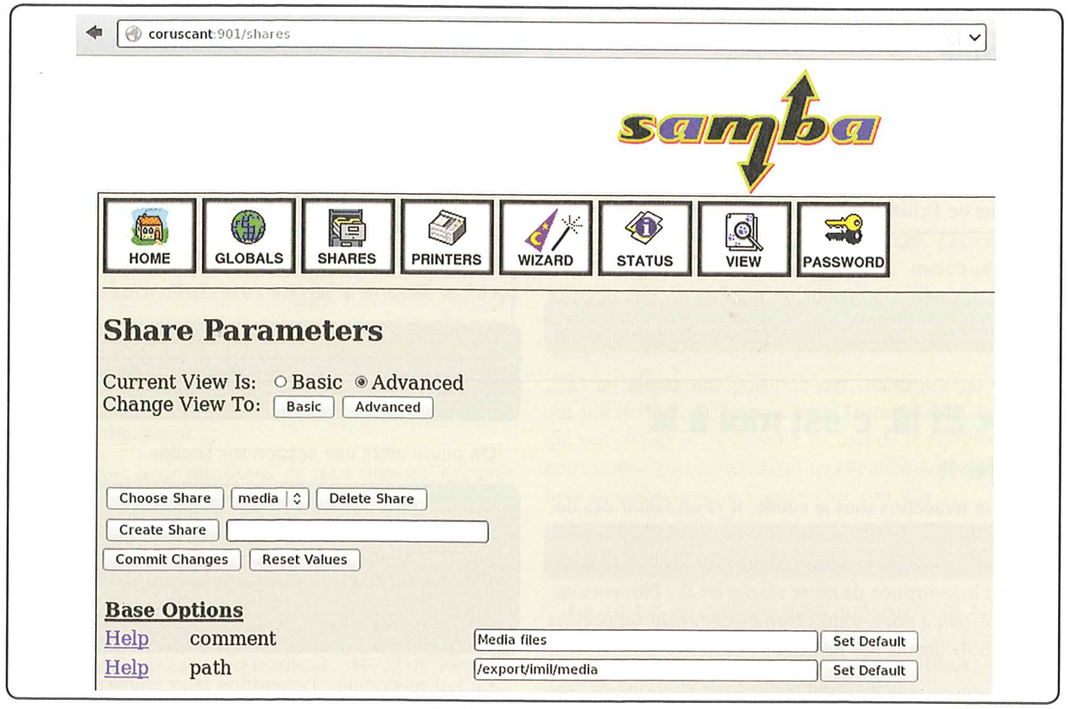


Fig. 2: Interface Web SWAT

port **901** de votre serveur de stockage à l'aide d'un navigateur quelconque comme le montre la figure 2.

Le fichier de configuration produit par l'interface Web pour la déclaration basique d'un partage non protégé est le suivant (fichier **/usr/pkg/etc/samba/smb.conf**) :

```
# Samba config file created using SWAT
# from UNKNOWN (192.168.1.1)
# Date: 2013/02/24 00:24:29

[global]
        workgroup = EMPIRE
        realm = CORUSCANT.HOME.IMIL.NET
        server string = Samba %v (%h)
        idmap config * : range =
        idmap config * : backend = tdb

[homes]
        comment = Home Directories
        read only = No
        browseable = No

[media]
        comment = Media files
        path = /export/imil/media
        guest ok = Yes
        locking = No
```

On peut s'assurer du bon fonctionnement du partage à l'aide de la commande **smbclient** exécutée sur un poste quelconque du réseau :

```
imil@tatooine:~$ smbclient //coruscant/media
Anonymous login successful
Domain=[EMPIRE] OS=[Unix] Server=[Samba 3.6.22]
smb: \> pwd
Current directory is \\coruscant\media\
```

Attention, il s'agit ici d'une configuration très épurée, sans authentification. Il sera souhaitable de placer au minimum quelques listes d'accès relatives à votre réseau local.

Enfin, mon choix de prédilection (c'est la méthode que j'utilise à profusion sur mon réseau), la troisième option consiste à exposer des points de montage NFS. NetBSD dispose dans son noyau GENERIC ainsi que dans son système de base de tous les outils nécessaires à le transformer en serveur de fichiers NFS : aucun paquet particulier à installer, il suffit comme à l'accoutumée de démarrer les bons services et de les configurer correctement. Ici, nous aurons besoin des démons **rpcbind**, **mountd**, **nfsd**, **lockd** et **statd**. On commande l'exécution automatique de ces serveurs au démarrage en ajoutant ceci au fichier **/etc/rc.conf** :

```
# cat >> /etc/rc.conf << EOF
rpcbind=YES
mountd=YES
nfs_server=YES
lockd=YES
statd=YES
EOF
```

On configure les points de montage à exposer, par exemple dans mon cas :

```
# j'exporte pkgsrc ainsi que les sources du système sur le réseau
/usr/pkgsrc     -maproot=root:wheel -network 192.168.1.0/24
/usr/src        -maproot=root:wheel -network 192.168.1.0/24

# je rend accessible en lecture seule les backups réguliers de
mes machines
/home/backup    -alldirs -ro -mapall=nobody -network
192.168.1.0/24

# j'exporte mon home directory pour ma station de travail
/home/imil      tatooine

# et enfin, l'ensemble du volume RAID 5 contenant mes photos de
vacances.
# L'option "-noresvport" est là pour satisfaire l'osx de ma femme
qui
# bloque lamentablement les transactions NFS lorsqu'on lui
demande de
# faire parvenir ses requêtes RPC depuis des ports réservés, ce
qui
# devrait être son comportement normal...
/export         -alldirs -noresvport -maproot=root:wheel -network
192.168.1.0/24
```

Reste à démarrer les démons déclarés précédemment :

```
# /etc/rc.d/rpcbind start
# /etc/rc.d/mountd start
# /etc/rc.d/nfsd start
# /etc/rc.d/nfslocking start
```

Et de s'assurer que les exports NFS sont bien accessibles depuis le réseau :

```
imil@tatooine:~$ showmount -e coruscant
Export list for coruscant:
/export      192.168.1.0
/home/imil   tatooine.home.imil.net
/home/backup 192.168.1.0
/usr/pkgsrc  192.168.1.0
/usr/src     192.168.1.0
```

Toujours côté client, j'ai opté, sur mes machines GNU/Linux, pour le très pratique **autofs** qui me permet de monter les exports NFS à la volée, la configuration se traduit ainsi :

```
imil@tatooine:~$ cat /etc/auto.master
/net    /etc/auto.net
```

Après avoir redémarré le service **autofs**, on dispose sans plus de précautions de nos exports NFS dans le répertoire **/net** :

```
imil@tatooine:~$ ls /net/coruscant
export  home  usr
imil@tatooine:~$ ls /net/coruscant/export
imil  iscsi  netboot
```

Bien. Il est l'heure de vider les *sdcard* de nos mobiles.

5 Et pour quelques kilos de plus

Comme je l'annonçais au début de cet article, le stockage de masse n'était pas l'unique finalité de cette orgie de disques. En effet, ayant récemment migré mon réseau 100BaseT en réseau Gigabit, je nourrissais l'envie de me passer de disques locaux dans les machines de mon réseau domestique. Disposant désormais de toute la place et des protocoles nécessaires pour accueillir différents systèmes sur un montage NFS, il ne manque plus pour réaliser ce fantasme que d'un simple serveur DHCP/PXE qui indiquera aux machines leur adresse IP, leur fournira un noyau ainsi qu'un *root filesystem*. J'utilise pour cela le minuscule, mais très puissant Dnsmasq **[10]**. On installe le logiciel sur le NAS à l'aide de **pkgin** :

```
# pkgin in dnsmasq
```

Et l'on produit un fichier de configuration de quelques lignes :

```
interface=re1
dhcp-ignore=tag:!known
dhcp-range=192.168.1.1,192.168.1.254,255.255.255.0,1h
dhcp-host=00:0a:22:Ce:13:A2,krayt,192.168.1.200,set:netbsd32
dhcp-host=00:21:16:ee:56:a0,openbox,192.168.1.201,set:openbox
dhcp-boot=tag:netbsd32,pxelinux.0
dhcp-boot=tag:openbox,NB6-MAIN-R3.3.4-OPEN
dhcp-option=tag:netbsd32,17,/export/netboot/krayt/root
enable-tftp
tftp-root=/tftpboot
```

Dans cet exemple, je fais démarrer deux machines via PXE/NFS, j'ignore les machines inconnues de la configuration grâce à la directive **dhcp-ignore=tag:!known** pour laisser le réel serveur DHCP de mon réseau prendre le relais. Je donne un tag aux machines connues, ici une station qui démarre NetBSD en 32 bits et une NeufBoite dont j'ai fait l'acquisition à des fins de bidouillage. C'est grâce à ce tag que nous choisirons l'amorce de nos machines *diskless*. Enfin, toujours grâce à l'identification par tag, nous passons l'option **17** (**root path**) qui indiquera à la station **krayt** où se situe la racine de son système de fichiers à monter en NFS. Bien évidemment, on peut ici déclarer tout type de système sans disque. Il faudra malgré tout garder à l'esprit que les transferts seront limités à la capacité de la carte réseau ainsi que des équipements connectés, capacité qu'il est possible de décupler si le commutateur sur lequel sont branchées les machines supporte le protocole LACP (802.3ad) **[11]** permettant, entre autres, l'agrégation de plusieurs liens physiques en un lien logique.

6 La solution du pauvre ? plutôt celle de l'ourson malin

J'ai récemment été confronté à une panne disque : ce dernier était branché sur un contrôleur RAID matériel, le constructeur importe peu, mais j'ai lutté pendant plusieurs heures pour juste **comprendre** qu'un disque était simplement défectueux. En effet, il est regrettable de constater qu'encore aujourd'hui, les outils fournis par les constructeurs sont au mieux des binaires propriétaires qu'il faut aller chercher au fin fond d'un lien obscur dans la section « support » cachée derrière 28 formulaires, et au pire inexistants pour votre système. Je ne parlerai que très succinctement de la convivialité des outils en question, jugez plutôt :

```
# MegaCli64 -PDList -aALL
```

Cette commande affiche les informations concernant la santé des disques physiques branchés au contrôleur dont vous devinerez le nom. Intuitif, hein. Trêve de bave inutile, un *tweet* que je recevais récemment résume bien la situation : *« @iMilnb après avoir lu ça et les 1001 pbs des contrôleurs RAID, autant passer au software raid ... »*

Oui, très clairement, car au vu de l'évolution des pilotes RAID logiciels, des capacités de branchement à chaud du matériel récent et de la puissance disponible dans une machine de bureau basique, les différenciateurs qui favorisaient le RAID hardware ne semblent plus vraiment d'actualité. D'ailleurs, précisons que le *setup* que nous venons de balayer n'est pas exactement original puisque la société Wasabi systems **[12]** fournit des SAN/NAS professionnels articulés autour du système NetBSD, lequel doit justement son système de journalisation à la société Wasabi, employeur d'un certain nombre de développeurs officiels du Projet.

Sur ce, joyeux backups ;) ■

Références

[1] Définition de NAS : http://fr.wikipedia.org/wiki/Serveur_de_stockage_en_r%C3%A9seau

[2] FreeNAS : http://www.freenas.org/

[3] RAIDFrame : http://www.pdl.cmu.edu/RAIDframe/

[4] Digital UNIX : http://en.wikipedia.org/wiki/Tru64_UNIX#Digital_UNIX

[5] GUID Partition Table : http://fr.wikipedia.org/wiki/GUID_Partition_Table

[6] iSCSI : http://fr.wikipedia.org/wiki/ISCSI

[7] Server Message Block : http://fr.wikipedia.org/wiki/Server_Message_Block

[8] Samba : http://fr.wikipedia.org/wiki/Samba_%28informatique%29

[9] Network File System : http://fr.wikipedia.org/wiki/Network_File_System

[10] Dnsmasq : http://dnsmasq.org/

[11] IEEE 802.3ad : http://fr.wikipedia.org/wiki/IEEE_802.3ad

[12] Wasabi Systems : http://wasabisystems.com/

AUTHOMATIC : PYTHON, OAUTH ET RÉSEAUX SOCIAUX

par **Benjamin Zores** *[Directeur technique @ Alcatel-Lucent Enterprise]*

Utiles ou purement agaçants, les réseaux sociaux font désormais partie intégrante de nos vies et du Web moderne. En tant que développeur (Web), il peut sembler judicieux de permettre à vos utilisateurs de se connecter à votre application au travers de leurs réseaux de prédilection. Qu'il s'agisse uniquement d'authentifier vos utilisateurs via les fonctionnalités de connexion de ces derniers ou pour en récupérer diverses informations et interagir avec, voyons comment Python et le framework Authomatic peuvent vous simplifier la vie.

Il est aujourd'hui très simple de développer rapidement un site Web dynamique. Certains développeurs utiliseront PHP, les vrais (« troll inside ») se tourneront naturellement vers Python et un framework tel que **Flask** (déjà présenté dans *GNU/Linux Magazine* **[1]**). Rapidement, vous souhaiterez probablement ajouter une fonctionnalité d'authentification de vos utilisateurs. Et plutôt que de réinventer encore une fois la roue, pourquoi ne pas proposer d'identifier et authentifier vos utilisateurs au travers d'un de leurs réseaux sociaux ? Plus besoin de formulaire d'enregistrement spécifique à votre site, pénible pour l'utilisateur qui devra encore se souvenir d'un énième identifiant/mot de passe supplémentaire. La très grande majorité des réseaux sociaux (Google+, Facebook, Twitter, Yahoo, Amazon, PayPal, Microsoft, GitHub, Evernote, Linkedin ...) utilise désormais le protocole d'authentification **OAuth** (en version 1 ou 2, selon les réseaux).

1 OAuth

À proprement parler, **OAuth** n'est pas réellement un protocole d'authentification. Il s'agit plutôt d'un protocole d'autorisation (à coupler à un protocole d'authentification donc, tel que **OpenID** ou **SAML**) dont le but est d'autoriser un site Web à utiliser l'API sécurisée d'un autre site Web pour le compte d'un utilisateur. Typiquement, il s'agit de la page Web qui s'affiche après avoir saisi votre identifiant et mot de passe de connexion, et qui, selon la demande du site Web en question, vous informe que le site requiert vos informations personnelles, votre liste de contact, votre numéro de sécurité sociale, votre numéro de carte bleue, etc. (vous imaginez la suite). Les permissions sont diverses et variées et libres à chaque site ou application d'être plus ou moins intrusif relativement à votre vie privée (mais surtout à vous de lui concéder ou non ces droits). Dans le cas le plus simple (celui qui nous intéresse ici), il peut s'agir de requérir une autorisation d'accès basique (votre nom, prénom, e-mail, avatar...), ce qui est déjà pas mal. Dans le principe, une fois l'autorisation accordée de votre part, le site pourra ainsi utiliser l'API du *provider* OAuth (celui sur lequel vous vous êtes identifié) pour accéder à vos informations (et ainsi vous identifier si besoin). Dans les faits, OAuth 1.0 a été finalisé en octobre 2007 et était relativement compliqué à implémenter, car supportait la signature électronique, le chiffrement et la vérification client directement au sein du protocole. Octobre 2012 a vu l'avènement de la version 2.0 du protocole (après de très nombreux *drafts* de RFC) où le protocole a été très largement revu et simplifié (la sécurité du protocole a été supprimée pour ne plus dépendre que de SSL) et la très grande majorité des sites majeurs du Web 2.0 sont désormais *provider* OAuth 2.0. Là où le bât blesse, c'est que les différents experts en sécurité sont relativement unanimes sur la faiblesse du protocole, mais ce n'est pas bien grave après tout, il ne s'agit que d'autoriser un accès à votre vie privée. De même, pour simplifier la vie, chaque *provider* y allant de son implémentation (certains se basent sur un *draft*, d'autres sur la version finale), impossible de faire un client générique. Il devra donc être adapté pour quasiment chaque *provider*. Que du bonheur ...

2 Authomatic

Mais réjouissez-vous, Python est, comme à l'accoutumée, là pour vous simplifier la vie. Le framework **Authomatic [2]** est en effet là pour vous simplifier la vie et l'accès à tous les *providers* OAuth. Son but est en effet d'automatiser la connexion aux réseaux sociaux via OAuth (1.0 ou 2.0), identifier vos utilisateurs et autoriser votre application Web à accéder à leurs données. Attention cependant, l'accès aux données utilisateur autorisé, votre application est capable de les utiliser au sein de la session utilisateur. En aucun cas vous ne pourrez récupérer son mot de passe de connexion, mais vous pourrez utiliser ses informations. Libre à vous de les stocker ou non, et d'en faire ce que bon vous semble, mais ça, c'est un contrat de confiance entre vous et vos utilisateurs (politique de respect de la vie privée), et qui donc ravira plus ou moins ces derniers.

Notez que pour l'instant Authomatic ne fonctionne pas encore avec Python 3. L'installation se fait via `pip` :

```
# pip install authomatic
```

L'utilisation et la configuration d'Authomatic sont fort simples comme nous allons le voir. Concrètement, commençons par créer une application Web avec **Flask** :

```
from flask import Flask, render_template, request, make_response

app = Flask(__name__)
if __name__ == '__main__':
    app.run(debug=True, port=80)
```

Avec ces quelques lignes, vous disposez d'un serveur Web tournant sur le port `80` et en attente de connexions. Ajoutons les quelques lignes suivantes qui nous permettront d'afficher la page `login.html` pour toutes les requêtes sur l'URL `http://myapp.com/login` :

```
@app.route('/login', methods=['GET'])
def login():
    return render_template('login.html')
```

Ceci permet de capturer les requêtes vers `/login` pour exécuter la fonction `login()` qui va générer une page Web via la directive de rendu de template **Jinja2** de **Flask**. Créez donc un fichier `login.html` au sein du répertoire `templates`, dont le contenu peut ressembler à quelque chose du type suivant

```
<div class="container">
  <div class="col-sm-3">
    <div class="row">
      <a class="btn btn-block btn-facebook btn-material-indigo"
href="login/fb">
        <i class="fa fa-facebook"></i> Sign in with Facebook
      </a>
   </div>
  <br/>
  <div class="row">
    <a class="btn btn-block btn-twitter btn-material-lightblue"
href="login/tw">
      <i class="fa fa-twitter"></i> Sign in with Twitter
    </a>
  </div>
  <br/>
  <div class="row">
    <a class="btn btn-block btn-google-plus btn-material-red"
href="login/google">
      <i class="fa fa-google-plus"></i> Sign in with Google+
    </a>
  </div>
</div>
```

Quelques gouttes d'HTML/CSS saupoudrées d'un framework Web tel que **Bootstrap [3]** et cela nous fait une petite page de connexion comme illustrée au sein de la figure 1.

Ces jolis boutons nous redirigent ainsi sur les pages `/login/fb`, `/login/tw` et `/login/google`, en fonction du réseau social (et donc du provider OAuth) désiré, à savoir

Figure 1 : Fenêtre de connexion aux réseaux sociaux.

respectivement **Facebook**, **Twitter** et **Google+**. C'est là qu'Authomatic entre en jeu ! Ajoutez donc les quelques lignes de Python suivantes :

```
from authomatic.adapters import WerkzeugAdapter
from authomatic import Authomatic
import json

from config import CONFIG
from config import SECRET

AUTHOMATIC_STATE = 'authomatic:fb:state'
app.config['SECRET_KEY'] = SECRET
oauth_sessions= {}

authomatic = Authomatic(CONFIG, SECRET, report_errors=False)
```

Avec cela, notre framework est instancié, mais pas forcément configuré (nous allons y revenir tout de suite). Profitons-en pour directement ajouter à **Flask** la directive de traitement des pages de type **`/login/<provider>`** qui seront appelées par notre page **`login.html`**. Dans le cas d'un accès à ces pages, les lignes suivantes se chargeront de demander au framework Authomatic une connexion OAuth vers le *provider* demandé, de capturer la session, de récupérer les différentes informations demandées sur l'utilisateur (au sein de l'objet **`result.user`**) et de nous rediriger vers la page racine (à savoir **`/`**) de l'application Web.

```
@app.route('/login/<provider_name>/', methods=['GET', 'POST'])
def login_provider(provider_name):
    response = make_response()
    result = authomatic.login(WerkzeugAdapter(request, response),
                              provider_name,
                              session=session,
                              session_saver=lambda: app.save_
session(session, response))
    if result:
        if result.user:
            result.user.update()
            state = session[AUTHOMATIC_STATE]
            oauth_sessions[state] = result
        return redirect('/')
    return response
```

Dans le principe, l'utilisateur se connectera donc à la page **`/login.html`**, choisira son réseau social et sera redirigé vers **`/login/<provider>`**. Cette dernière, via **Authomatic**, vous renverra vers la page de demande d'identification et d'autorisation du *provider* demandé. Une fois validée (ou autorisée), la redirection se fait vers la page émettrice (i.e. votre application Web), avec les informations demandées sur l'utilisateur (i.e. son identité). À vous d'en faire ce que bon vous semble. Mais revenons quelques instants sur la configuration du framework **Authomatic**. Vous l'avez vu, nous importons les symboles **`CONFIG`** et **`SECRET`** depuis le fichier **`config.py`**. Voyons donc son contenu :

```
from authomatic.providers import oauth2, oauth1, openid

SECRET = "CLE_SECRET_ALEATOIRE"

CONFIG = {
    'tw': {
        'short_name': 1,
        'class_': oauth1.Twitter,
        'consumer_key': 'MA_CLE_TWITTER',
        'consumer_secret': 'MON_SECRET_TWITTER',
    },
    'fb': {
        'class_': oauth2.Facebook,
        'short_name': 2,
        'consumer_key': 'MA_CLE_FACEBOOK',
        'consumer_secret': 'MON_SECRET_FACEBOOK',
        'scope': ['user_about_me', 'email', 'publish_stream'],
    },
    'google': {
        'class_': oauth2.Google,
        'short_name': 3,
        'consumer_key': 'MA_CLE_GOOGLE.apps.googleusercontent.
com',
        'consumer_secret': 'MON_SECRET_GOOGLE',
        'scope': ['profile', 'https://www.googleapis.com/auth/
plus.login', 'email']
    },
}
```

La variable **`SECRET`** contient une clé secrète purement aléatoire qui décrit votre application de manière unique. Pour faire simple, je vous suggère de générer un UUID et de l'utiliser tel quel. Pour la variable **`CONFIG`**, les choses se compliquent. À vous de lister ici tous les *providers* que vous souhaitez supporter (ici **`tw`**, **`fb`** et **`google`**). Pour chaque *provider*, il vous faudra renseigner une clé et un mot de passe (ou secret) spécifique à votre application. Ces dernières sont uniques à votre application et doivent être déclarées auprès de chacun des *providers* que vous souhaitez utiliser. Il vous est en effet maintenant nécessaire de déclarer votre application auprès des différents réseaux sociaux au sein des différentes APIs réservées aux développeurs, comme nous allons le voir dans la suite. Enfin, le champ **`scope`** permet de déclarer au provider quelles sont les informations auxquelles votre application souhaite accéder. Ces dernières sont spécifiques à chaque *provider* (référez-vous donc aux pages de documentation pour développeurs respectives à chaque réseau social) et permettront à ce dernier d'afficher à l'utilisateur (lors de la phase d'identification/autorisation) quelles sont les informations auxquelles il est censé vous conférer des droits d'accès.

3 Créer une application pour Twitter

Pour ce faire, rien de très compliqué. Rendez-vous sur la page de gestion des applications **Twitter [4]** et validez le bouton ***Create New App***. Validez les conditions d'utilisations,

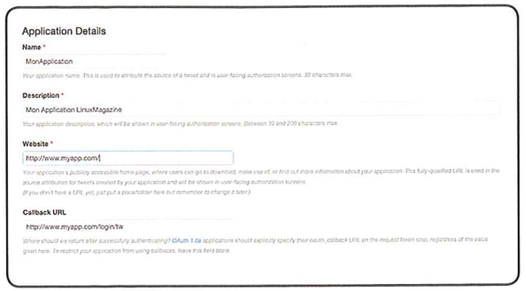


Figure 2 : Enregistrement d'application Twitter.

donnez un nom à votre application, une description sommaire, l'adresse Web de votre application et surtout l'adresse de retour (ou « *Callback URL* »), comme présenté au sein de la figure 2.

L'adresse de retour est très importante. Si vous vous souvenez bien, votre application vous a redirigé vers le provider **OAuth**. Une fois la connexion validée, ce dernier doit vous renvoyer vers votre application. Il doit donc connaître l'adresse de retour. Dans notre exemple, il s'agira de http://myapp.com/login/tw. Ceci implique un petit détail : pour que cela soit possible, votre application doit être publiquement accessible (il ne peut s'agir d'une adresse IP de votre LAN). Assurez-vous donc de disposer du DNS adéquat. Sinon, en phase de développement (et donc avant publication), éditez simplement votre fichier **`/etc/hosts`** pour tromper votre navigateur et ajoutez-y la ligne suivante :

```
127.0.0.1 myapp.com
```

Enfin, choisissez les droits d'accès de votre application (typiquement lecture seule ou lecture/écriture). Pas la peine d'effrayer vos utilisateurs en leur demandant les droits de publier des messages en leur nom si vous souhaitez juste récupérer leur identité. Votre application est maintenant enregistrée chez **Twitter**. En la sélectionnant dans la liste des applications (vous pouvez en avoir plusieurs) et en accédant à l'onglet *« Keys and Access Tokens »* vous devriez retrouver votre *« Consumer Key (API Key) »* et *« Consumer Secret (API Secret) »* que vous pourrez maintenant renseigner dans votre fichier **`config.py`** (respectivement à la place de **`MA_CLE_TWITTER`** et **`MON_SECRET_TWITTER`**).

Figure 3 : Enregistrement d'application Facebook.

4 Créer une application pour Facebook

Comme dans le cas de Twitter, rendez-vous sur la page **Facebook** pour développeurs **[5]** et cliquez sur l'onglet *« My Apps »*, puis *« Add a New App »* et choisissez une application de type *« Site Web »*. Donnez un nom à votre application pour créer un nouvel identifiant unique d'application (*« App ID »*), choisissez une catégorie et saisissez l'adresse de votre application Web. Si vous avez suivi toutes les étapes, vous devriez retrouver des informations proches de la figure 3 au sein du menu ***Settings / Basic***.

Notez que le champ *« AppID »* correspond à la valeur à renseigner **`MA_CLE_FACEBOOK`** de notre fichier **`config.py`** et le bouton ***AppSecret*** vous affichera la valeur à renseigner pour **`MON_SECRET_FACEBOOK`**. Rendez-vous maintenant dans le champ *« Settings / Advanced »* et renseignez le champ *« Valid OAuth redirect URIs »* de la section *« Security »* par http://myapp.com/login/fb afin de renseigner **Facebook** dans votre adresse de retour (« *Callback URL* »), comme nous l'avons fait avec Twitter.

5 Créer une application pour Google

Finissons le tour des réseaux sociaux avec **Google**. Le principe est le même que pour ses deux confrères. Rendez-vous donc sur la page de console pour développeurs **Google [6]** et validez le bouton ***Create Project***. Donnez un nom à votre application et choisissez un identifiant unique. L'application créée, reste à la configurer. Naviguez au sein du menu ***APIs & auth / API*** et ajoutez le support (au moins) des APIs Google *« Contacts API »*, *« Google+ API »*, *« Identity Toolkit API »*, comme vous pouvez le voir sur la figure 4.

Libre à vous d'ajouter autant d'APIs que nécessaire selon votre besoin, mais normalement vous devriez couvrir les bases nécessaires. Rendez-vous maintenant dans le menu ***Credentials*** et déclarez un nouveau client de type *« Web Application »* via le bouton ***Create a new Client ID***. Renseignez ensuite votre adresse e-mail, le nom de votre application, et l'adresse de votre application. Enfin, dans le champ *« Authorized Redirect URIs »*, saisissez l'adresse de retour désirée (« *Callback URL* »), à savoir http://myapp.com/login/google dans notre cas. Si tout s'est bien passé, vous devriez vous retrouver avec une page semblable à celle de la figure 5.

Le champ *« Client ID »* correspondra ainsi à **`MA_CLE_GOOGLE`** tandis que le champ *« Client Secret »* correspondra à la variable **`MON_SECRET_GOOGLE`** de notre fichier **`config.py`**. Tout cela

Figure 4 : Enregistrement d'application Google.

Figure 5 : Configuration d'application Google.

terminé, notre site est fonctionnel et la page de connexion devrait vous renvoyer vers les différents *providers* OAuth. À titre d'exemple, la première connexion d'un utilisateur à votre application via **Google** devrait aboutir à une page similaire à celle de la figure 6.

6 Sécurité et persistance d'authentification

Revenons maintenant à notre code Python. Si vous avez bien suivi, une fois l'autorisation accordée (et l'identification terminée par le même biais), vos utilisateurs sont redirigés vers votre application (et dans notre cas, on renvoie l'utilisateur vers la page racine, à savoir `/`). Oui, mais voilà, si l'on requiert une authentification et une autorisation de l'utilisateur, autant l'utiliser pour sécuriser l'accès à nos pages. Comment peut-on en interdire l'accès aux utilisateurs anonymes ? C'est, ma foi, fort simple ;-)

Pour ce faire, créez une fonction de routage **Flask** classique vers la racine (page « `/` ») à laquelle nous adjoindrons un second décorateur qui forcera l'authentification.

```
@app.route('/')
@login_required
def index(oauth):
    email = oauth.user.email
    if email is None:
        email = ""
    return render_template('index.html', result=oauth,
name=oauth.user.name, email=email)
```

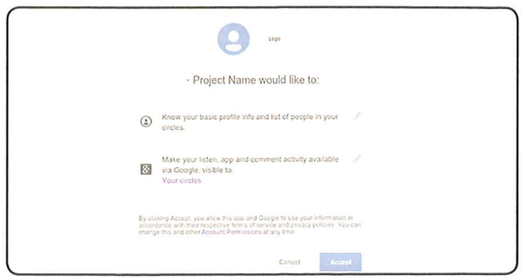


Figure 6 : Exemple d'autorisation d'accès aux informations Google.

La fonction `index()` est classique. Elle fera appel à la fonction `render_template()` de **Flask** pour générer une page HTML. Ce qui est moins classique, c'est l'ajout du décorateur `@login_required` dont voici le code :

```
import functools

def oauth_valid_session(session):
    state = None
    if AUTHOMATIC_STATE in session:
        state = session[AUTHOMATIC_STATE]
    if state in oauth_sessions:
        return True
    return False

def login_required(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        if not oauth_valid_session(session):
            return redirect('/login')
        state = session[AUTHOMATIC_STATE]
        kwargs['oauth'] = oauth_sessions[state]
        return func(*args, **kwargs)
    return wrapper
```

La fonction `login_required()` semble complexe, mais est finalement fort simple. Lors du retour avec succès de la page `/login`, nous avons demandé au framework **Authomatic** d'enregistrer la session utilisateur (et les informations associées) au sein du dictionnaire `oauth_sessions`. Nous vérifions donc dans la fonction `login_required()` si une session existe. Si tel est le cas, nous la réutilisons, afin de permettre de naviguer entre les pages du site Web ou de les recharger sans être obligé de se réauthentifier. Dans le cas contraire (session OAuth invalide), nous forçons la redirection du navigateur vers la page `/login`. En décorant toutes nos pages Flask par `@login_required`, il devient ainsi impossible à un utilisateur anonyme d'accéder à l'application.

7 Accès aux données

Mais revenons à l'essentiel, l'accès à notre application ;-) La fonction `index()`, via `render_template()` va s'occuper de

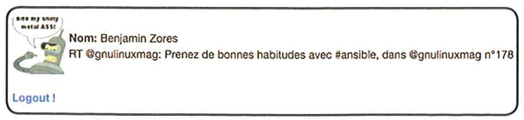


Figure 7 : Affichage d'informations « sociales » de notre utilisateur.

générer du code HTML à partir du fichier `templates/index.html`. L'objet `oauth` que nous lui passons en paramètre contient toutes les informations utilisateur nécessaires à la construction de la page. Nous allons donc créer une simple page qui affiche l'avatar de l'utilisateur, son nom et le dernier message qu'il a publié, toutes ces informations étant extraites du réseau social depuis lequel il s'est connecté. Ceci devrait ressembler de près ou de loin à la figure 7.

Pour cela, jouons un peu avec le template **Jinja2** de **Flask** et déclarons les lignes suivantes :

```
    {% if result.user.credentials %}

      {% if result.provider.name == 'tw' %}
        {% set url = 'https://api.twitter.com/1.1/users/lookup.
json?screen_name={0}'.format(result.user.username) %}
        {% set response = result.provider.access(url) %}
        {% set nm = response.data[0]['name'] %}
        {% set av = response.data[0]['profile_image_url_https']
%}
    {% set url = 'https://api.twitter.com/1.1/statuses/user_
timeline.json?count=1&user_id={0}'.format(result.user.username)
%}
        {% set response = result.provider.access(url) %}
        {% set msg = response.data[0].text %}
      {%endif %}

      {% if result.provider.name == 'fb' %}
        {% set url = 'https://graph.facebook.com/v2.2/
me?fields=id,name,picture' %}
        {% set response = result.provider.access(url) %}
        {% set nm = response.data.name %}
        {% set av = response.data.picture.data.url %}
        {% set url = 'https://graph.facebook.com/v2.2/me/
links?count=1' %}
        {% set response = result.provider.access(url) %}
        {% set msg = response.data.data[0].comments.data[0].
message %}
      {%endif %}

      {% if result.provider.name == 'google' %}
        {% set url = 'https://www.googleapis.com/plus/v1/people/
{0}'.format(result.user.id) %}
        {% set response = result.provider.access(url) %}
        {% set nm = response.data.displayName %}
        {% set av = response.data.image.url %}
    {% set url = 'https://www.googleapis.com/plus/v1/people/{0}/
activities/public?maxResults=1'.format(result.user.id) %}
        {% set response = result.provider.access(url) %}
        {% set msg = response.data.items()[3][1][0].title %}
      {%endif %}
    {%endif %}
```

Ces lignes permettent de différencier le réseau social utilisé pour se connecter via la variable `result.provider.name`. L'utilisateur étant désormais authentifié, notre application peut utiliser l'API du fournisseur (ici **Twitter**, **Facebook** ou **Google**) pour récupérer l'avatar, le nom de l'utilisateur ou encore son dernier message publié, via des requêtes successives au moyen de `result.provider.access(url)`. Les variables **Jinja2** `av`, `nm` et `msg` que nous venons de calculer sont ensuite affichées au sein d'un simple tableau HTML comme présenté ci-dessous :

```
    <table cellpadding="5">
      <td><img src="{{av}}" height="72" width="72"/></td>
      <td><b>Nom:</b> {{nm}}<br/>{{msg}}</td>
```

Notez que nous en profitons pour rajouter un lien permettant à l'utilisateur de se déconnecter.

8 Déconnexion

Eh oui, car si nous proposons à nos utilisateurs de se connecter à notre application, il serait judicieux de leur proposer la possibilité d'en sortir. Ceci se fait très simplement par l'implémentation d'une nouvelle routine `logout()` pour **Flask**, réagissant en *callback* à la requête `/logout`. Rien de très compliqué avec cette dernière, comme on peut le voir ci-dessous :

```
@app.route('/logout', methods=['GET'])
def logout():
    state = session[AUTHOMATIC_STATE]
    if state in oauth_sessions:
        del oauth_sessions[state]
    return redirect('/')
```

La fonction s'occupe simplement de récupérer la session en cours et de la supprimer de la liste des sessions OAuth. On redirige ensuite l'utilisateur vers la page racine (i.e. `/`), qui est associée au décorateur `login_required()` et, comme l'utilisateur n'est plus authentifié, il sera automatiquement redirigé vers la page `/login` et la boucle est bouclée :-)

Conclusion

Voici donc un moyen simple de créer une application Web avec Python et le framework **Authomatic** pour vous connecter aux différents fournisseurs **OAuth** et donc accéder à toutes les informations de vos réseaux sociaux préférés/détestés :-) À vous la conquête du Web ! ■

Références

[1] E. Heitor, « *Introduction à Flask le micro système maousse costaud* », *GNU/Linux Magazine n°166*, décembre 2013 : http://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-166/Introduction-a-Flask-le-micro-systeme-maousse-costaud

[2] Framework Authomatic : http://peterhudec.github.io/authomatic/

[3] Framework Twitter Bootstrap : http://getbootstrap.com/

[4] Gestion d'applications Twitter : https://apps.twitter.com/

[5] Gestion d'applications Facebook : https://developers.facebook.com/apps/

[6] Gestion d'applications Google : https://console.developers.google.com/

CORDOVA : QUOI DE NEUF ?

par ***Tristan Colombo***

Nous avons déjà parlé du projet Apache Cordova dans ces pages [1], mais cela fait quelque temps que nous n'avons pas suivi ses dernières avancées. Voyons comment le projet a évolué et s'il permet de créer des applications Web multiplateformes encore plus simplement.

Si vous ne connaissez pas encore Apache Cordova, c'est un projet qui permet de développer des applications Web et de les distribuer sur différentes plateformes telles qu'Android, iOS, Windows Phone, BlackBerry 10, ou encore Firefox OS. Les applications, développées en HTML et JavaScript sont ensuite « empaquetées » dans des applications natives (à l'aide des *WebViews*). Nous allons nous concentrer ici sur le processus de développement d'une application Cordova.

1 Installation

Je supposerai que vous avez déjà installé et configuré votre environnement avec le SDK Android.

L'installation de Cordova se fait très simplement en ligne de commandes en passant par nodeJS (bien sûr, il est toujours possible de télécharger directement les sources sur le site officiel). Je vais décrire ici l'installation par nodeJS qu'il faudra avoir installé auparavant. Sur une distribution basée sur Debian, cela donne :

```
$ sudo aptitude install nodejs npm
```

On utilise ensuite **npm** (Node Package Manager) pour télécharger et installer Cordova :

```
$ sudo npm -g cordova
```

Vous disposez alors de la version 4.2.0 (à l'heure où ces lignes sont écrites, le lien du site n'est pas à jour et indique que c'est la version 4.0.0 qui est proposée en téléchargement, ce qui est faux).

Voilà, c'est tout pour l'installation, nous avons maintenant accès à Cordova depuis la ligne de commandes.

2 Création d'un hello world

Nous ne nous attarderons pas sur la création d'une énième application avec Cordova, mais le parcours rapide d'un petit hello world nous montrera le processus global de développement et l'apport des nouveautés.

Tout d'abord, il suffit d'une ligne pour initialiser un projet :

```
$ cordova create hello_glmf com.gnulinuxmag.hello HelloGLMF
Creating a new cordova project with name "HelloGLMF" and id
"com.gnulinuxmag.hello" at location "hello_glmf"
```

En parcourant le répertoire **hello_glmf**, on s'aperçoit qu'une nouvelle architecture a été adoptée avec la présence de trois répertoires vides :

- **hooks** : les *hooks* fonctionnent comme avec git, ce sont des scripts que vous pouvez lancer automatiquement avant ou après des opérations effectuées par Cordova (compilation par exemple) ;
- **platforms** : contiendra les fichiers relatifs aux plateformes cibles du projet ;
- **plugins** : contiendra les extensions permettant l'accès à des composants natifs requis par votre application (on ne charge plus de code inutile).

On trouve également un fichier de configuration **config.xml** et un répertoire **www** contenant un fichier **index.html** qui n'est plus une copie d'un fichier d'exemple, mais enfin un véritable fichier minimal !

Pour l'instant, nous n'avons encore ciblé aucune plateforme, donc si vous ouvrez le fichier **index.html** dans un navigateur, vous n'obtiendrez qu'une image vous indiquant que Cordova essaye de se connecter au périphérique (voir figure 1).

2.1 Définition des plateformes cibles

L'intérêt de Cordova est de pouvoir ajouter plusieurs types de plateformes. On peut se contenter de n'ajouter qu'Android, mais tant qu'à faire je vais également ajouter le support d'une autre plateforme, en l'occurrence Firefox OS. Notez qu'il faut se placer dans le répertoire du projet sous peine d'obtenir le message d'erreur *« Current working directory is not a Cordova-based project »*.

```
$ cd hello_glmf/
$ cordova platform add android
Creating android project...
Creating Cordova project for the Android platform:
        Path: platforms/android
        Package: com.gnulinuxmag.hello
        Name: HelloGLMF
        Android target: android-19
Copying template files...
Project successfully created.
$ cordova platform add firefoxos
Creating firefoxos project...
Project Path platforms/firefoxos
Package Name com.gnulinuxmag.hello
Project Name HelloGLMF
```

La commande **cordova platform ls** vous permettra d'afficher la liste des plateformes utilisées par votre projet et celles supportées par Cordova (le résultat dépend des SDK installés sur votre machine).

Si, après réflexion, vous souhaitez retirer le support de l'une des plateformes de votre application, il faudra utiliser la commande **rm** :

```
$ cordova platform rm windows8
```

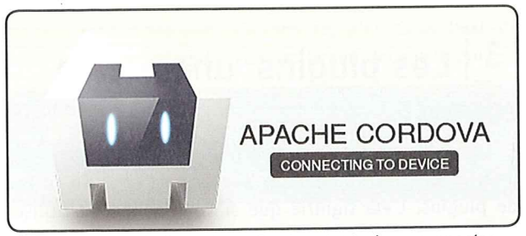


Fig. 1: Image indiquant que Cordova tente de se connecter à un périphérique.

Après ajout du support des plateformes, vous pourrez constater la présence de nouveaux sous-répertoires dans le répertoire **platforms**. Ces répertoires portent le nom des plateformes ajoutées et une copie du répertoire **www** y a été insérée : pour Android, il s'agit de **platforms/android/assets/www** et pour Firefox OS, ce sera **platforms/firefoxos/www**. Bien sûr, il ne s'agit pas ensuite d'aller modifier le code de l'application pour chaque plateforme, Cordova n'aurait plus aucun intérêt ! C'est simplement une adaptation aux différentes plateformes du code du répertoire **www**.

2.2 L'application proprement dite

Je l'ai dit, il ne s'agit pas d'une application complète, mais d'un simple hello world. Les modifications du fichier **www/index.html** seront donc minimes et nous nous contenterons d'ajouter du texte à un message existant :

```
...
35:         <p class="event received">Device is Ready<br />Hello
GNU/Linux Magazine !</p>
...
```

2.3 Compilation de l'application

La compilation d'un projet se fait en deux étapes : la « préparation » et la compilation proprement dite. Les commandes associées à ces processus sont **prepare** et **compile** :

```
$ cordova prepare
$ cordova compile
```

La commande **build** est un raccourci permettant d'effectuer directement ces deux opérations à la suite.

Note

Si vous ne souhaitez effectuer la compilation que pour une seule plateforme, faites suivre votre commande du nom de la plateforme cible :

```
$ cordova prepare android
```

2.4 Test de l'application

Pour pouvoir tester l'application, il faut lancer l'émulateur de la plateforme cible. Là encore, tout se passe depuis la commande **cordova** :

```
$ cordova emulate android
```

Attention : sans indication du nom de l'émulateur, ce sera l'émulateur **default** qui sera utilisé. Assurez-vous que celui-ci est correctement configuré. Pour spécifier un nom d'émulateur, il faudra utiliser l'option **--target**. Voici un exemple avec une AVD (*Android Virtual Device*) nommée **Nexus_5** :

```
$ cordova emulate --target=Nexus_5 android
```

Sachez également que la commande **emulate** inclus les commandes **prepare** et **compile** précédentes et qu'elle est équivalente à :

```
$cordova run android --emulator
```

Pour mettre à jour l'application après modification vous n'avez plus qu'à taper :

```
$ cordova emulate
```

Votre application doit être présente dans le menu de l'émulateur avec le logo de Cordova (puisque nous ne l'avons pas modifié). Vous n'avez qu'à cliquer dessus pour lancer l'application (voir figure 2).

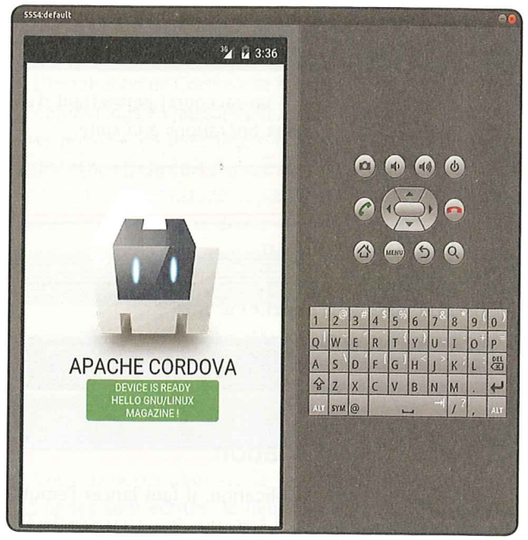


Fig. 2: Exécution de notre application de test dans une AVD.

2.5 Installer l'application sur un périphérique

Pour tester l'application sur un smartphone par exemple, vous devez tout d'abord relier celui-ci à l'ordinateur. Vérifiez que le débugage USB est activé et que la connexion se fasse bien en MTP (icône « Connecté en tant que périphérique USB » dans la barre de notifications Android). Si tout est correct, la commande **adb devices** devrait vous le signaler :

```
$ adb devices
List of devices attached
f668a249        device
```

Dans le cas contraire (une connexion PTP par exemple), vous obtiendrez :

```
$ adb devices
List of devices attached
f668a249        unauthorized
```

Il n'y a plus alors qu'une seule commande à taper pour compiler et installer l'application sur le smartphone :

```
$ cordova run android
```

Force est de constater qu'un très gros travail a été réalisé au niveau du mode ligne de commandes. On peut effectuer toutes les opérations du cycle de développement très simplement : un véritable bonheur pour ceux qui ont connu les premières versions de Cordova ! Pour obtenir de l'aide, vous pouvez utiliser l'option **--help** après le nom d'une commande. Par exemple, pour l'aide de **run** :

```
$ cordova run --help
```

Intéressons-nous maintenant à la gestion des plugins... mais d'ailleurs, qu'est-ce qu'un plugin Cordova ?

3 Les plugins, une bonne idée ?

Les APIs permettant d'accéder aux différents périphériques sont maintenant accessibles sous la forme de plugins. Cela signifie que si vous souhaitez utiliser le gyroscope, il faudra ajouter à votre projet le plugin **org.jhurliman.cordova.gyroscope** ou encore, pour avoir accès à la console de débugage il faudra ajouter le plugin **org.apache.cordova.console**. Tout cela vous semble bien lourd ? Présenté de la sorte, effectivement on s'imagine tout de suite en train de rechercher dans les méandres d'internet le nom du plugin permettant enfin d'accéder à telle ou telle fonctionnalité d'un smartphone... En fait, cela se fait très simplement à partir de la ligne de commandes en invoquant **cordova** suivi de **plugin search** et du ou des mots clés recherchés. Par exemple, pour connaître le nom complet du plugin permettant d'accéder au gyroscope :

```
$ cordova plugin search gyroscope
org.jhurliman.cordova.gyroscope - Gyroscope Plugin
```

En fonction des critères, vous pourrez avoir le choix entre plusieurs plugins. Voici un exemple pour le GPS :

```
$ cordova plugin search gps
au.com.cathis.ma.gpslocation - Exposes GPS Location related
values to a Cordova application.
com.dataforpeople.plugins.gpssettings - Display the location
settings page
com.vitorventurin.gps - Cordova GPS plugin - iOS CoreLocation
LocationService / Android LocationService GPS
fr.louisbl.cordova.gpslocation - Android geolocation plugin
using GPS provider
org.bluetooth.gps - Cordova plugin for connecting external GPS
via Bluetooth
sk.tamex.locationandsettings - Open WirelessSettings,WifiSetti
ngs,LocationSettings. Check if Gps or Wireless Network Location
is enabled.
sk.tamex.plugins.gpssettings - Display the gps location settings
```

Lisez bien la description de chaque plugin. Dans le cas d'une utilisation basique du GPS, il y a fort à parier que le plugin recherché soit en fait un plugin de *geolocation* :

```
$ cordova plugin search geolocation
...
org.apache.cordova.geolocation - Cordova Geolocation Plugin
```

Tous les plugins dont le nom commence par **org.apache.cordova** sont des *cores plugins*, les plugins de base du système. Vous obtiendrez naturellement leur liste par :

```
$ cordova plugin search org.apache.cordova
org.apache.cordova.battery-status - Cordova Battery Plugin
...
org.apache.cordova.vibration - Cordova Vibration Plugin
```

Cela signifie donc qu'il y a des plugins de plus haut niveau, possédant des fonctionnalités plus avancées ! Vous voulez scanner un code-barre et vous pensiez utiliser le plugin d'accès à la caméra ? Il y a plus simple :

```
$ cordova plugin search bar code
...
com.phonegap.plugins.barcodescanner - You can use the
BarcodeScanner plugin to scan different types of barcodes (using
the device's camera) and get the metadata encoded in them for
processing within your application.
...
```

Pour l'instant, nous n'avons fait que rechercher des plugins. Pour pouvoir les utiliser dans votre application, il va falloir les installer.

3.1 Installation d'un plugin

Une fois que vous connaissez le nom du plugin que vous souhaitez utiliser, la commande **cordova plugin add** permettra d'ajouter le plugin à votre application :

```
$ cordova plugin add com.phonegap.plugins.barcodescanner
Fetching plugin "com.phonegap.plugins.barcodescanner" via plugin
registry
Installing "com.phonegap.plugins.barcodescanner" for android
Installing "com.phonegap.plugins.barcodescanner" for firefoxos
```

Vous noterez que le plugin est installé automatiquement sur chacune des plateformes déclarées (et les permissions sont mises à jour comme vous pourrez le constater dans le fichier **platforms/android/AndroidManifest.xml**). Vous pouvez ainsi ajouter autant de plugins que vous le souhaitez. Pour savoir quels sont les plugins installés pour votre application, vous pourrez faire un **ls** :

```
$ cordova plugin ls
com.phonegap.plugins.barcodescanner 2.0.1 "BarcodeScanner"
```

Dans la même logique, la suppression d'un plugin se fera par **rm** :

```
$ cordova plugin rm org.apache.cordova.geolocation
Uninstalling org.apache.cordova.geolocation from android
Uninstalling org.apache.cordova.geolocation from firefoxos
Removing "org.apache.cordova.geolocation"
```

3.2 Et la documentation ?

Installer des plugins c'est bien, pouvoir les utiliser c'est mieux ! La documentation des *cores plugins* est accessible sur le site GitHub d'Apache : https://github.com/apache.

Pensez à appliquer un filtre sur le mot-clé **cordova** pour ne pas avoir à rechercher votre documentation dans les 30 pages de projets...

Pour les autres plugins, il faudra passer par votre moteur de recherche favori. Par exemple, pour le plugin BarcodeScanner, on aboutit à la page https://github.com/wildabeast/BarcodeScanner/tree/c3090dc après un passage par le site officiel de Phonegap. Ceci va nous permettre de tester ce plugin.

3.3 Un petit exemple

Nous allons simplement créer un bouton permettant de scanner un code et d'afficher ses informations. Notre code ne sera pas très élégant, vous pourrez toujours l'améliorer par la suite s'il vous sert de base pour un projet.

On commence par éditer le fichier **www/index.html** pour ajouter le bouton :

```
35:                 <p class="event received">Device is Ready<br />
36:                 <button onclick="app.scan();">Scan !</button>
```

Lorsque l'on cliquera sur le bouton, nous déclencherons l'exécution de la méthode **scan()** de l'objet **app**. Il faut donc définir cette méthode dans le fichier **www/js/index.js** :

```
01: /*
...
19: var app = {
49:     // Scan function
50:     scan: function() {
51:         cordova.plugins.barcodeScanner.scan(
52:             function (result) {
53:                 alert("Code barre détecté\n" +
54:                     "Données: " + result.text + "\n" +
55:                     "Format: " + result.format + "\n" +
56:                     "Annulé lors du scan: " + result.cancelled);
57:             },
58:             function (error) {
59:                 alert("Erreur lors du scan: " + error);
60:             }
61:         );
62:     }
...
```

Vous n'avez plus qu'à brancher votre smartphone sur l'ordinateur et à lancer la commande (pour une exécution sous Android) :

```
$ cordova run android
```

En appuyant sur le bouton « Scan ! », vous passerez en mode caméra pour scanner un code-barre. Si vous annulez l'opération avant le scan, la variable **result.cancelled** vaudra **false** et **result.text** ainsi que **result.format** ne contiendront aucune donnée. Dans le cas contraire, sauf cas d'erreur, vous obtiendrez le code lu (**result.text**) ainsi que son format (**result.format**). La figure 3 montre le résultat obtenu en scannant le code-barre de *GNU/Linux Magazine n°179*.

Je pense que l'on peut dire qu'effectivement, l'ajout du mécanisme des plugins est une très bonne idée permettant d'ajouter des fonctionnalités en toute transparence sans avoir à gérer les permissions (et risquer d'en demander beaucoup trop).

En ce qui concerne le développement de vos propres plugins, nous y reviendrons dans un prochain article.

4 Les hooks

Les *hooks* sont des scripts qui s'exécutent avant et après les différentes étapes d'une application développée en utilisant le mode CLI de Cordova. Ces scripts peuvent être écrits dans n'importe quel langage, à partir du moment où ils sont exécutables (il faudra penser à ajouter la ligne de *shebang* pour indiquer au shell l'interpréteur à utiliser).

Les hooks doivent être placés dans le répertoire **hooks** et dans un sous-répertoire dont le nom est de la forme **before_cmd** ou **after_cmd** où **cmd** représente le nom d'une commande CLI (ie **prepare**, **build**, **run**, etc.). Une liste complète des commandes cibles est donnée dans le fichier **hooks/README.md**.

Pour tester ce mécanisme, je vous propose de mettre en place un hook très simple qui permettra de compiler une feuille de style less en css avant exécution de la commande **prepare**.

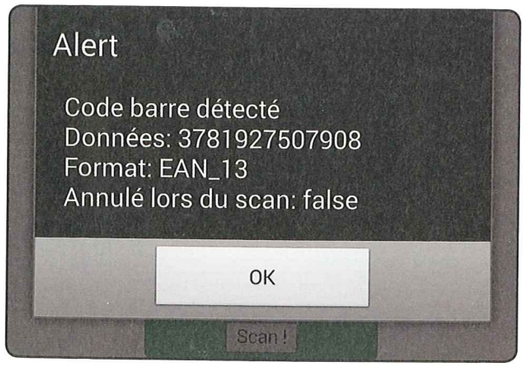


Fig. 3: Résultat du scan d'un code-barre.

Less css

Less est un pré-processeur de css : il permet d'utiliser des variables et des fonctions pour écrire plus rapidement des feuilles de style. Un fichier less doit être compilé sous forme de fichier css avant de pouvoir être utilisé.

L'installation de less se fait à l'aide du *node package manager* :

```
$ npm install -g less
```

La compilation utilise la commande **lessc** :

```
$ lessc fichier.less > fichier.css
```

Nous allons reprendre le fichier **www/css/index.css** pour en faire un fichier **index.less** dans lequel nous allons utiliser simplement des noms de variables pour toutes les couleurs (en modifiant les couleurs pour s'assurer que la génération du fichier css fonctionne correctement) :

```
01: /*
...
20: // Colors
21: @c1: #b6b6b6;
22: @c2: #f3f3f3;
23: @c3: #000;
24: @c4: #4b856e;
25:
26: * {
27:     -webkit-tap-highlight-color: rgba(0,0,0,0); /* make
transparent link selection, adjust last value opacity 0 to 1.0 */
28: }
29:
30: body {
...
34:     background-color:@c2;
35:     background-image:linear-gradient(top, @c1 0%, @c2 51%);
...
70:     @media screen and (min-aspect-ratio: "1/1") and (min-
width:400px) {
...
```

Pensez à ajouter les guillemets en ligne 70 à cause d'un bug de **lessc**.

Il faut maintenant créer le hook. Pour cela, nous devons créer un nouveau répertoire :

```
$ mkdir hooks/before_prepare
```

Ensuite, nous indiquons comment compiler le fichier less dans un script **hooks/before_prepare/before_prepare_compile_less.sh** :

```
01: #!/bin/bash
02:
03: echo "Compilation de index.less en index.css"
04:
05: lessc www/css/index.less > www/css/index.css
```

Notez que nous aurions pu construire quelque chose de bien plus élégant si **lessc** avait été bien écrit et renvoyait la valeur **0** lorsque tout se passe bien (au lieu de **8997** tout le temps ?!?!). Nous aurions alors pu ajouter :

```
07: if [[ $! == 0 ]]; then
08:     echo "[OK]"
09: else
10:     echo "[FAILED]"
11: fi
```

Ici ces dernières lignes ne serviront à rien. N'oubliez surtout pas de rendre le script exécutable :

```
$ chmod ugo+x hooks/before_prepare/before_prepare_compile_less.sh
```

Au lancement de la commande **prepare**, nous obtiendrons désormais la compilation de notre fichier less :

```
$ cordova prepare
Running command: hello_glmf/hooks/before_prepare/before_prepare_
compile_less.sh hello_glmf
Compilation de index.less en index.css
```

Voilà encore un mécanisme bien pratique !

Conclusion

Je n'avais plus utilisé Cordova depuis quelque temps et il faut avouer que j'ai été très agréablement surpris par les multiples améliorations présentes dans cette version 4.2.0. Les plugins et les hooks apportent une très grande flexibilité et fiabilité aux développements d'applications. Par exemple, comme nous avons pu le voir avec le petit hook que nous avons créé, vous n'oublierez plus jamais de compiler vos fichiers less. Ça ne vous donne pas envie de développer des applications avec Cordova ? Moi oui ! ■

Références

[1] T. Colombo, *« Développer une application pour Firefox OS »*, *GNU/Linux Magazine n°161*, juin 2013.

[2] Site officiel d'Apache Cordova : http://cordova.apache.org/

GÉNÉREZ LA DOCUMENTATION DE VOS APIS AVEC APIDOC

*par **Frédéric Le Roy** [Ingénieur en informatique, membre du groupe Domogik, touche à tout]*

Vous voulez documenter les APIs de votre projet, qu'il soit en Java, JavaScript, Python, etc. ? Ne cherchez plus et adoptez apidoc pour séduire vos utilisateurs ou vos clients !

Apidoc permet de rédiger, comme d'autres outils, la documentation de vos APIs dans le code. Apidoc a l'avantage de fonctionner avec différents langages : JavaScript, Java, PHP, Python, etc. Les avantages que j'ai trouvé à cet outil par rapport à d'autres sont :

- la syntaxe claire ;
- le contenu généré au format HTML qui est très sympathique à lire ;
- la possibilité de versionner l'API et de pouvoir comparer simplement deux versions : un vrai plus pour les contributeurs ;
- même si le but premier est de documenter des APIs Web (REST), il est possible de détourner l'outil pour documenter l'utilisation d'une *message queue* (file de messages).

Il y a par contre à mon avis un petit défaut : **node.js** est nécessaire en tant que dépendance. Tout le monde ne sera pas forcément fan :).

1 Installation

Tout d'abord, il va falloir installer `node.js` :

```
$ sudo apt-get update
$ sudo apt-get install nodejs
```

Ensuite, nous allons installer **npm**, le gestionnaire de paquets de `node.js` :

```
$ sudo apt-get install npm
```

Et nous pouvons enfin installer apidoc :

```
$ npm install apidoc -g
```

2 Commençons à documenter notre code

2.1 Un code, quel code ?

Pour illustrer l'article, nous allons voir un exemple fictif en Python. Le langage n'a que peu d'importance, la syntaxe d'apidoc étant la même. Seule la manière d'écrire les commentaires va changer : `//` ou `/* … */` ou `#` ou `%` suivant le langage choisi.

Nous allons considérer pour l'exemple, un morceau de projet créé avec Flask (qui permet de créer entre autres des APIs REST). Voici le code sans commentaires :

```
class Voitures():

    def get(self, id):
        """
        """
        pass

    def post(self):
        """
        """
        pass

    def put(self, id):
        """
        """
        pass

    def delete(self, id):
        """
        """
        pass
```

Ajoutons de la documentation à la fonction `get()` :

```
    def get(self, id):
        """
        @api {get} /voitures/:id Récupérer les informations sur
une voiture
        @apiName GetVoitures
        @apiGroup Voitures

        @apiParam {Number} id Identifiant d'une voiture

        @apiSuccess {String} marque Marque de la voiture
        @apiSuccess {String} modele  Modèle de la voiture
        @apiSuccess {Number} annee  Année de la voiture
        @apiSuccess {String} immatriculation  Immatriculation de
la voiture

        @apiSuccessExample Success-Response:
            HTTP/1.1 200 OK
            {
              "marque" : "Citroen",
              "modele" : "DS3",
              "marque" : 2013,
              "immatriculation" : "AB 999 CD"
            }

        @apiError (404) VoitureNonTrouvee La voiture n'a pas été
trouvée

        @apiErrorExample Error-Response:
            HTTP/1.1 404 Not Found
            {
              "error": "VoitureNonTrouvee"
            }
        """
        pass
```

En restant modestes, nous pouvons déjà remarquer qu'un gros paquet de lignes a été ajouté au code ! Paradoxalement, je ne trouve pas que cela rende le code illisible : on voit clairement qu'il ne s'agit pas de code et la structure est claire et intuitive.

Revenons sur les différents éléments du bloc de commentaires :

- `@api` : ici le type d'appel est décrit en premier, à savoir GET : `{get}`. Il y a ensuite l'URL à appeler avec le paramètre contenu dans l'url `:id`. À la fin, on retrouve la description.
- `@apiName` est le nom de l'API. La documentation incite à concaténer le nom de la méthode (`Get`, `Put`, `Post`, `Delete`…) et le chemin ou le nom de la fonctionnalité, soit ici `Get+Voitures`. Ce nom sera utilisé dans le menu de la documentation générée.
- `@apiGroup` définit le groupe auquel la méthode appartient. Ici elle appartient au groupe `Voitures` comme le feront les méthodes `put`, `post` et `delete` de la classe `Voitures`. Le groupe sera utilisé en tant qu'élément de menu dans le menu.
- `@apiParam` permet de décrire les différents paramètres de la méthode. Nous retrouvons d'abord le type `{Number}` (`Boolean`, `Number`, `String`, `Object`, …), puis l'identifiant dans l'URL `id` et pour finir la description. Il est possible de définir un paramètre comme optionnel en l'encadrant de crochets, par exemple : `@apiParam {Number} [id] Identifiant de la voiture`.
- `@apiSuccess` va décrire les différents éléments renvoyés par la méthode : tout d'abord le type, puis le nom du champ tel qu'on le retrouve dans la réponse pour finir par la description.
- `@apiSuccessExample` va permettre de montrer un exemple de données renvoyées par l'API.
- `@apiError` va nous permettre de lister les différents messages d'erreur possibles et de les décrire. Il y a d'abord le code HTTP de l'erreur entre parenthèses : `(404)` ; puis le code applicatif de l'erreur et enfin sa description.
- `@apiErrorExample` va permettre de montrer un exemple de retour en erreur.

Maintenant, soyons jeunes, fougueux et pressés et lançons une première génération de la documentation ! Pour ceci, rien de plus simple, il suffit de se placer dans le dossier contenant le fichier python et de lancer :

```
$ apidoc
…
apidoc: Please create an apidoc.json.
…
```

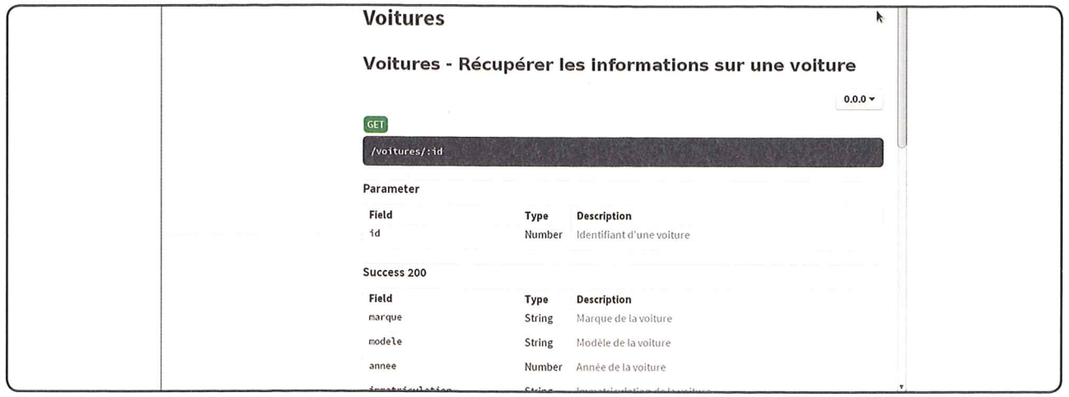


Fig. 1 : Première génération.

Parmi toutes les lignes qui n'ont que peu d'intérêt, une seule attire le regard (c'est facile, elle est en couleur) : celle que j'ai laissé dans la sortie ci-dessus. Pressés que nous sommes nous avons sauté le fichier de configuration. Peu importe et voyons ce que donne la génération qui se trouve dans le dossier `./doc/` (voir figure 1).

Tout d'abord, nous retrouvons un menu à gauche avec des catégories (Voitures) et des liens par catégorie qui correspondent à chacune des méthodes. La description est utilisée pour l'affichage et si vous regardez de plus près, le lien utilise le champ `apiName` : file:/-../doc/index.html#api-Voitures-GetVoitures.

Remarquez le titre contenant des **undefined** dans l'onglet du navigateur et une liste déroulante à droite contenant une version 0.0.0. Il va falloir paramétrer tout cela maintenant !

2.2 D'autres options ?

Il existe d'autres options pour compléter vos blocs de commentaires. Je vous invite à creuser la documentation du site **[1]** qui contient en plus quelques exemples afin de trouver ce qui correspond le plus à votre projet.

2.3 Le fichier de configuration principal

Afin de configurer le titre, la version et d'autres broutilles, nous allons créer un fichier, nommé `apidoc.json`.

Voici un exemple simple :

```
{
  "name": "Gestion des voitures",
  "version": "0.1",
  "description": "Le super outil de gestion des voitures qui est vraiment génial",
  "title": "Mes voitures"
}
```

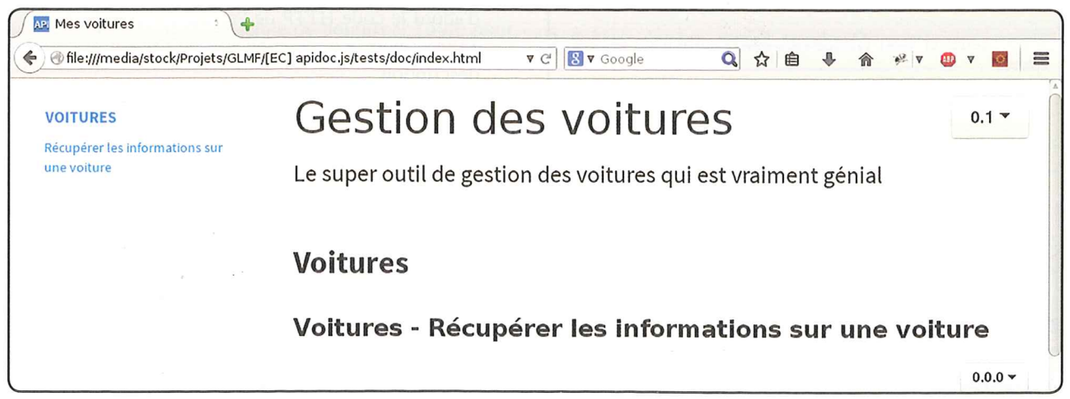


Fig. 2 : Ajout du fichier de configuration.

Ces éléments sont principalement cosmétiques. `name` et `description` sont utilisés dans l'en-tête de la page. `title` est utilisé comme titre d'onglet et `version`...et bien il s'agit de la version courante.

Régénérons une seconde fois la documentation et nous obtenons la page présentée en figure 2.

Nous voyons bien les différents titres et la version en haut à droite, mais... Dans la liste déroulante en bas à droite nous avons toujours une version `0.0.0` qui résiste !

Notez que ce fichier peut être complété avec d'autres options qui ont un intérêt principalement cosmétique. Je vous laisse consulter la documentation par vous-même.

3 Gestion des versions

3.1 Ajout de la version

Ajoutons la version dans le bloc de commentaires (pour rappel, notre code est en version `0.1`) :

```
    def get(self, id):
        """
        @api {get} /voitures/:id Récupérer les informations sur une voiture
        @apiVersion 0.1
        @apiName GetVoitures
        @apiGroup Voitures
```

Générons la documentation :

```
$ apidoc
...
apidoc: Error: "@apiVersion" in file "./exemple2_chap_4.py" block number 1 version "0.1" not valid.
    at Parser._parseBlockElements (/usr/local/lib/node_modules/apidoc/lib/parser.js:166:12)
...
apidoc: Error: "@apiVersion" in file "./exemple2_chap_4.py" block number 1 version "0.1" not valid.
```

Oups ! Relisons mieux la doc : *Simple versioning supported (major.minor.patch)*. Bon OK, il faut trois chiffres… Nous allons ajouter un troisième chiffre du coup : `0.1.0`. Ce qui donne maintenant :

```
        @apiVersion 0.1.0
```

Avant de régénérer la documentation, nous allons aussi mettre à jour la version de `0.1` à `0.1.0` dans le fichier `apidoc.json` :

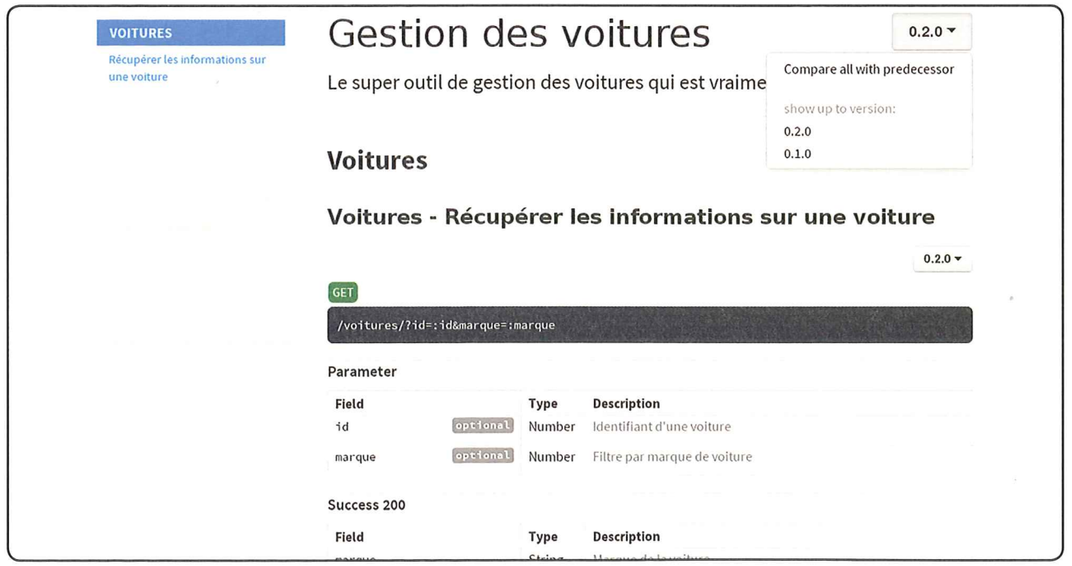


Fig. 3 : Les différentes versions.

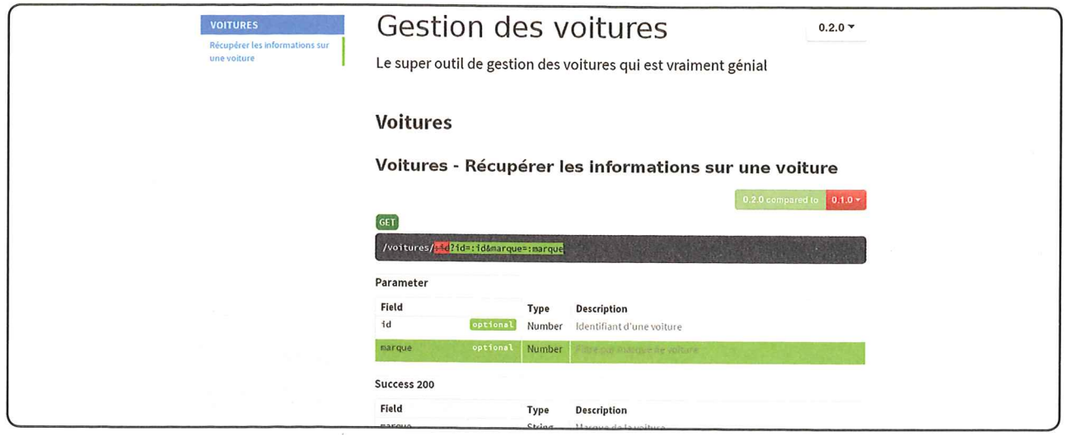


Fig. 4 : Comparaison des versions de l'API.

```
"version": "0.1.0",
```

Nous pouvons maintenant régénérer sans souci la documentation. Si vous regardez les listes déroulantes liées à la version à droite, elles sont désormais bien remplies.

3.2 Mise à jour de l'API

Notre super service REST de gestion des voitures a été livré en version **0.1** et nous allons maintenant créer une évolution dans la version **0.2** qui va modifier l'API de la requête GET. Pour gérer ceci, nous allons simplement ajouter un second bloc de commentaires à la fonction :

```
    def get(self, id):
        """
        @api {get} /voitures/:id/ :m Récupérer les informations
sur une voiture
        @apiVersion 0.1.0
        @apiName GetVoitures
        @apiGroup Voitures
       @apiGroup Voitures

        """
        """
        @api {get} /voitures/?id=:id&marque=:marque Récupérer les
informations sur une voiture
        @apiVersion 0.2.0
        @apiName GetVoitures
        @apiGroup Voitures

        @apiParam {Number} [id] Identifiant d'une voiture
        @apiParam {Number} [marque] Filtre par marque de voiture

        """
```

Il y a ici deux modifications : le paramètre **id** devient optionnel et le nouveau paramètre **marque** apparaît. Cette manière de gérer les versions des APIs peut être très verbeuse pour des APIs qui bougent beaucoup, mais cela reste quand même très clair dans le code.

Nous allons également modifier le fichier **apidoc.json** pour mettre à jour la version :

```
"version": "0.2.0",
```

Lançons la génération de la documentation. On voit sur la figure 3 qu'il est possible d'afficher la version de l'API de son choix, ce qui est très pratique. Mais ce qui est encore plus pratique, c'est de pouvoir comparer deux versions d'une API comme illustré sur la figure 4 : les éléments mis à jour sont surlignés ou barrés.

Avec une telle documentation, il sera donc très simple pour des utilisateurs de l'API de se maintenir à jour et de faire évoluer leurs applicatifs clients d'une version à une autre.

4 Documenter autre chose qu'une API REST/Web

Dans le cadre d'un projet, nous avons en plus de notre API REST, une *message queue*. Nous avons choisi de la documenter également avec apidoc. Avec une MQ, point de GET/POST/… Les messages sont toutefois catégorisés : REQ/REP (requête/réponse) et PUB/SUB (publication, abonnement).

Il y a également une différence au niveau de l'implémentation. Là où le code des fonctions entrantes d'un service REST est regroupé à peu près au même endroit, dans le cadre de notre *message queue*, le code est réparti un peu partout en fonction des composants. Afin de documenter simplement l'usage, nous avons créé un dossier d'exemples qui contient un script python par usage possible de la *message queue* sur le projet. Voici un exemple (en anglais, désolé) :

```
#!/usr/bin/python
"""
@api {REQ} plugin.start.do Request to start a plugin
@apiVersion 0.4.0
@apiName plugin.start.do
@apiGroup Plugin
@apiDescription This request is used to ask Domogik's manager to
start a plugin
* Source client : Domogik admin, any other interface which can
manage the plugins
* Target client : always 'manager'

@apiExample {python} Example usage:
    cli = MQSyncReq(zmq.Context())
    msg = MQMessage()
    msg.set_action('plugin.start.do')
    msg.add_data('name', 'diskfree')
    msg.add_data('host', 'darkstar')
   print(cli.request('manager', msg.get(), timeout=10).get())

@apiParam {String} name The plugin name to start
@apiParam {String} host The host on which is hosted the plugin

@apiSuccessExample {json} Success-Response:
['plugin.start.result', '{"status": true, "host": "darkstar",
"reason": "", "name": "diskfree"}']

@apiErrorExample {json} Error-Response:
['plugin.start.result', '{"status": false, "host": "darkstar",
"reason": "Plugin \'xxx\' does not exist on this host", "
name": "xxx"}']
"""
import zmq
from zmq.eventloop.ioloop import IOLoop
from domogikmq.reqrep.client import MQSyncReq
from domogikmq.message import MQMessage

cli = MQSyncReq(zmq.Context())
msg = MQMessage()
msg.set_action('plugin.start.do')
msg.add_data('name', 'diskfree')
msg.add_data('host', 'darkstar')
print(cli.request('manager', msg.get(), timeout=10).get())
```

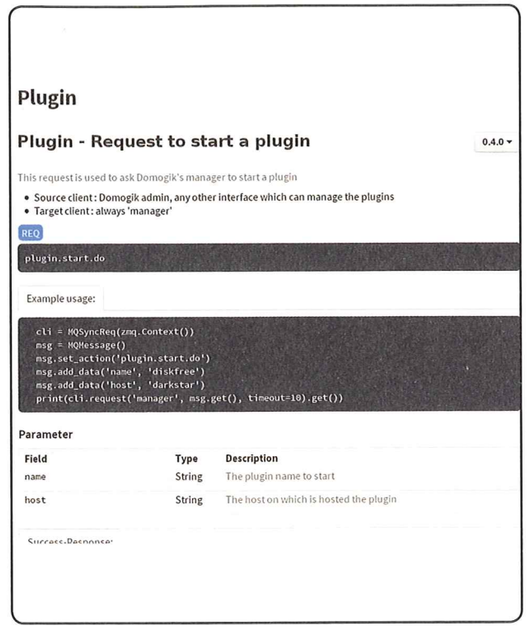


Fig. 5 : Documentation d'une MQ.

Notez tout d'abord dans **@api** le remplacement des habituelles fonctions GET/… par le type de message utilisé pour la message queue (ici **REQ**). Ce changement est totalement transparent pour apidoc qui va l'afficher de la même manière qu'il afficherait GET. Ensuite, pour pallier l'absence d'URL, nous avons pu utiliser **@apiExample** qui permet d'illustrer l'usage avec un exemple de code. Le reste est tout ce qu'il y a de plus classique.

Sur la figure 5, vous pouvez voir le rendu qui est vraiment sympathique !

Conclusion

Apidoc est pour moi un outil vraiment superbe pour documenter des APIs ! Le rendu est sexy, plus que pratique et intuitif. C'est un outil idéal pour séduire des contributeurs ou utilisateurs d'un projet. ■

Référence

[1] Documentation d'apidoc : http://apidocjs.com/

INTRODUCTION À POSTSCRIPT – ÉLÉMENTS DU LANGAGE

par ***Sébastien Namèche*** *[Humble locataire de la planète Terre]*

PostScript, vous connaissez tous ce nom. Mais lequel d'entre vous a eu la curiosité de soulever le capot de son imprimante pour en savoir en peu plus ? Et pourtant, il y a à découvrir ! En effet, PostScript implémente des concepts peu communs dans les autres langages et qui méritent le détour. Plongez dans l'exotisme, suivez le guide...

Il y a quelques mois, j'ai présenté dans ces colonnes le langage fonctionnel Erlang, bien éloigné des langages impératifs ou orientés objet traditionnels. Aujourd'hui, je viens vous parler d'un autre langage tout aussi mal connu qu'intéressant. Ce qui me fascine dans ces langages méconnus pour lesquels j'ai développé un attrait irrationnel, ce sont les concepts que nous n'avons pas l'habitude de rencontrer dans les langages que nous pratiquons habituellement.

PostScript est présenté comme un langage de description de page. Mais au-delà de ces fonctions spécialisées, il possède des caractéristiques plus étonnantes : notamment, il s'agit d'un langage interprété en polonaise inverse sur pile. C'est ce qui fait son charme. L'usage des dictionnaires est également un aspect intéressant.

1 Ghostscript et impression

Pour vous essayer au langage, il vous faut un interpréteur PostScript. Vous pouvez en trouver un dans une imprimante PostScript si vous en possédez, mais l'accès à l'interpréteur en mode interactif n'est pas aisé et il vous faudra imprimer une feuille pour chaque essai de votre programme. Plus simple et plus pratique, il existe plusieurs interpréteurs PostScript logiciels dont le très connu et très vénérable Ghostscript. Le nom du paquetage à installer est `ghostscript` dans la plupart des distributions Linux. L'usage de Ghostscript est très simple et, pour le démontrer, consacrons à l'usage et saluons le monde :

```
$ gs
GPL Ghostscript 9.10 (2013-08-30)
Copyright (C) 2013 Artifex Software, Inc.  All rights reserved.
This software comes with NO WARRANTY: see the file PUBLIC for
details.
GS>/Times-Roman findfont
Loading NimbusRomNo9L-Regu font from /opt/local/share/
ghostscript/9.10/Resource/Font/NimbusRomNo9L-Regu... 3685004
2183179 3833600 2536344 1 done.
GS<1>20 scalefont
GS<1>setfont
GS>400 200 moveto
GS>(Bonjour Monde!) show
GS>showpage
>>showpage, press <return> to continue<<
```

Lorsque vous démarrez l'interpréteur Ghostscript en mode interactif avec la commande `gs`, une fenêtre graphique s'ouvre et représente l'état de la page courante qui est dans un premier temps vide. Le petit programme précédent recherche une police de caractères (`findfont`), change la taille de ses caractères (`scalefont`) et l'active comme police courante (`setfont`). Puis il déplace le point courant aux coordonnées `400` sur l'axe horizontal et `200` sur l'axe vertical (`moveto`). Ces coordonnées sont exprimées en points pica, très utilisés dans l'imprimerie et dont la valeur vaut 1/72ème de pouce. Ce dernier valant 2,54 centimètres, le point pica mesure donc environ 0,353 millimètres. Enfin, la chaîne de caractères « Bonjour Monde! » est affichée sur la page (`show`) et celle-ci est imprimée (`showpage`).

Mais n'est-ce pas curieux ? Il semblerait que les arguments des procédures PostScript soient placés avant celles-ci : `/Times-Roman` devant `findfont`, `20` devant `scalefont`, `400` et `200` devant `moveto`, etc. Et quel est l'argument de la procédure `setfont` ? Il s'agit là de la plus importante des particularités du langage, mais nous y reviendrons.

La seconde remarque qu'inspire cet exemple est la nature interprétée du langage. Nul besoin de compilateur. L'interpréteur PostScript exécute les commandes à mesure que celles-ci lui sont présentées.

Maintenant, comment faire pour imprimer cette œuvre sur une imprimante réelle ? Commencez par placer notre magnifique programme dans un fichier, par exemple `ex01.ps`, puis plusieurs solutions s'offrent à vous suivant le cas de figure :

1) Que vous disposiez d'une imprimante PostScript ou non, vous pouvez utiliser un visualiseur de fichiers PostScript tel que **gv** sous Linux (au si délicieux look rétro) ou **Aperçu** sous Apple, il en existe bien évidemment d'autres. Utilisez ensuite la fonction d'impression de ces programmes.

2) Si vous avez une imprimante PostScript sous la main, il est possible de lui envoyer directement notre petit programme. Attention toutefois, à l'envoyer en mode *raw*. Sous Linux, par exemple, avec CUPS cela peut se faire ainsi : `lpr -l ex01.ps`, l'option `-l` signifiant que le fichier est déjà formaté et qu'il ne doit pas être passé à un filtre.

Un dernier mot sur Ghostscript : pour interpréter un fichier, par exemple `ex01.ps`, il est possible de le lui passer sur la ligne de commandes ainsi : `gs ex01.ps` ou bien, en mode interactif, en utilisant l'opérateur `run` de PostScript. Voici un exemple d'appel interactif dans `gs` :

```
GS>(ex01.ps) run
Loading NimbusRomNo9L-Regu font from /opt/local/share/
ghostscript/9.10/Resource/Font/NimbusRomNo9L-Regu... 3685004
2183179 3833600 2536344 1 done.
>>showpage, press <return> to continue<<
```

Bien, foin des préliminaires, passons aux choses sérieuses.

Bref historique

PostScript est né en 1982. La dernière version des spécifications du langage (la troisième) **[1]** date de l'année 1997. Depuis, seul un supplément paru en 1999 **[2]** est venu compléter certains aspects de l'interpréteur. Adobe a déclaré officiellement stopper le développement du langage en 2007 et pousse désormais le format PDF comme successeur à PostScript. Cependant, ce dernier semble avoir la peau dure et il se passera du temps avant qu'il ne soit complètement abandonné.

2 Éléments du langage

La grammaire du langage PostScript est confondante de simplicité :

- seuls les caractères de l'encodage ASCII sont autorisés ;
- les éléments du programme sont séparés par un ou plusieurs blancs qui sont, très classiquement : le caractère nul (code ASCII `0`), la tabulation (code ASCII `9`), le saut de ligne (`LF`, code ASCII `10`), le saut de page (`FF`, code ASCII `12`), le retour chariot (`CR`, code ASCII `13`) et l'espace (code ASCII `32`) ;
- tout élément qui ressemble à un nombre est un nombre (entier, réel ou exprimé dans une base non décimale) ;
- les caractères `(`, `)`, `<`, `>`, `[`, `]`, `{`, `}`, `/`, et `%` sont spéciaux et introduisent des types composés, un nom littéral ou un commentaire ;
- tous les autres groupes de caractères sont des noms.

Pas de structures de bloc, pas de mots réservés, pas d'opérateurs tels que `+`, `-`, etc.

PostScript est un langage interprété et, comme souvent avec ce genre de langages, le typage est dynamique. Le type stocké par une variable est résolu lors de l'exécution. Il existe deux grandes familles d'objets :

- les types simples qui portent en eux-mêmes leur valeur ;
- les types composés qui sont des références vers une zone de mémoire stockant leur valeur, ce qui leur vaut d'avoir des propriétés parfois étonnantes comme nous le verrons plus loin.

2.1 Les nombres

Il existe classiquement deux types de nombres : les entiers et les réels. Dans les deux cas, il s'agit de types simples.

Les entiers s'expriment sous leur forme décimale (`0`, `3`, `-4`, etc.) ou dans une base non décimale : `2#10010` (binaire), `8#176` (octal), `16#12AF5` (hexadécimal).

Et les réels s'expriment sous leur forme classique telle que `3.14` ou scientifique : `2,41e22` ou `2.817e-15` (qui sont, histoire de vous donner le vertige, la distance en mètres qui nous sépare de la galaxie non naine la plus proche, la galaxie d'Andromède, et le rayon, toujours exprimé en mètres, d'un électron, qui n'est pas la plus petite des particules identifiées à ce jour).

2.2 Les commentaires

Le caractère `%` introduit un commentaire qui se poursuit jusqu'à la fin de la ligne. Pour les commentaires sur plusieurs lignes, il faut utiliser autant de caractères `%`.

2.3 Les chaînes de caractères

Les chaînes de caractères sont entourées par des parenthèses : `(Ceci est une chaine.)`.

Dans une chaîne, des parenthèses en correspondance seront gérées comme faisant partie de la chaîne : `(Ceci est (toujours) une chaine)`. Dans le cas contraire, il est nécessaire de placer un caractère d'échappement devant : `(Ceci est un smiley ;-\) dans une chaine)`.

En plus des parenthèses, le caractère d'échappement `\` sert également à introduire :

- lui-même : `\\` ;
- les sauts de ligne : `\n` ;
- les retours chariot : `\r` ;
- les tabulations : `\t` ;
- n'importe quel caractère exprimé sous forme octale : `\123` (« S »).

Les chaînes de caractères sont des types composés. Nous reviendrons sur les autres types composés introduits par les caractères `<`, `>`, `[`, `]`, `{` et `}` un peu plus tard.

2.4 Les noms

Tous les autres groupes de caractères délimités par un ou plusieurs blancs sont des noms. Les noms sont des types simples. Ainsi, tous ces noms sont valides : `abc`, `Abc`, `a@b`, `@#$`, `12h45`, `*123`, etc. Mais il est conseillé de rester classique et de ne pas (trop) laisser s'exprimer sa créativité, eu égard aux collègues qui reliront votre code.

> **Note**
>
> En fait, pour le fun, `123` est également un nom valide et il est possible de déclarer une variable portant ce nom, mais il sera difficile d'y faire référence par la suite.

Les noms sont sensibles à la casse.

Enfin, les noms peuvent être introduits de deux manières : sous leur forme classique où ils seront exécutés immédiatement (l'interpréteur cherchera une variable portant ce nom et récupérera son contenu pour l'interpréter immédiatement) et sous leur forme littérale qui est introduite par le caractère `/`. Sous cette forme, l'interprétation des noms est déférée. Elle permet donc d'utiliser un nom de variable qui n'est pas encore défini alors que la forme classique introduit forcément un nom de variable existante.

3 Les piles

De toutes les particularités du langage PostScript la notion de pile est la plus caractéristique. Une pile est un espace de stockage d'objets et les piles PostScript sont des piles LIFO (« *Last In, First Out* ») : l'élément déposé en dernier sur la pile sera le premier à en sortir. Oui, tout comme dans une pile d'assiettes.

Le langage PostScript utilise cinq piles :

- la pile des opérandes, le plus souvent appelée « pile » tout simplement ;
- la pile des dictionnaires ;
- la pile d'exécution ;
- la pile des états graphiques ;
- la pile des pochoirs.

Laissons de côté pour l'instant les trois dernières.

3.1 La pile des opérandes

Ceci va rappeler des vieux souvenirs à ceux d'entre vous qui, tout comme moi au lycée, se gaussaient de leurs petits camarades exhibant fièrement leurs méga-Casio et autres hypra-Texas Instrument et ne juraient que par une seule marque de calculatrices scientifiques : Hewlett-Packard. Ces engins, notamment les très fameuses séries HP28 et HP48, utilisaient également la notion de pile d'opérandes, même si leur langage de programmation n'avait pas poussé le concept de manière aussi avancée que PostScript.

L'idée est très simple : lorsque l'interpréteur PostScript identifie un nouvel élément du programme, il place celui-ci sur la pile sauf s'il s'agit d'un nom (non littéral). Nous allons ainsi empiler les différents éléments de notre programme. Lorsque l'interpréteur identifie un nom sous sa forme non littérale, il cherche (nous verrons bientôt où) la valeur associée à ce nom :

- si ce nom n'est pas défini, une erreur `/undefined` est générée ;
- si ce nom est associé à un tableau exécutable, il s'agit d'une procédure et celle-ci est exécutée ;
- sinon la valeur associée au nom est placée elle aussi sur la pile.

Les procédures prennent zéro ou plusieurs arguments sur la pile et y déposent l'éventuel résultat de leur travail. Il est donc important de connaître très exactement le nombre d'éléments de la pile consommés par une procédure ainsi que le nombre d'éléments déposés sur la pile par celle-ci (tout comme il est important de connaître le prototype des fonctions que vous employez dans les autres langages).

Prenons pour exemple le programme suivant :

```
10 4 sub 7 mul
```

Que j'aurais pu écrire ainsi :

```
10
4
sub
7
mul
```

L'interpréteur l'analyse ainsi :

- `10` est un entier, je le place sur la pile ;
- `4` est un entier, je le place sur la pile ;
- `sub` est un nom, je cherche s'il existe... tiens, oui, je l'ai trouvé, que contient-il ? Ah, un tableau exécutable, bon j'exécute la procédure ;
- `7` est un entier, je le place sur la pile ;
- `mul` est un nom, je cherche s'il existe... tiens, oui, je l'ai trouvé, que contient-il ? Ah, un tableau exécutable, bon j'exécute la procédure.

La sobriété de la grammaire du langage rend le travail de l'interpréteur particulièrement aisé.

Concernant les procédures référencées par les noms `sub` et `mul`, elles contiennent chacune un programme qui consomme deux éléments de la pile et y dépose en retour un élément. Plus précisément, elles attendent deux nombres (entiers et/ou réels) et renvoient également un nombre. Votre sagacité vous aura permis de deviner que la première soustrait ces deux nombres et la seconde en effectue le produit.

Voici une représentation du contenu de la pile pendant l'exécution du programme (le haut de la pile représentant l'élément le plus récent) :

```
1:  10   4   6   7   42
2:       10      6
```

Notre programme est l'expression traduite en PostScript de la formule mathématique : `(10 - 4) x 7`.

La pile permet donc de passer des arguments aux procédures et de récupérer le fruit de leur travail, mais également de conserver des résultats intermédiaires (comme, par exemple, le nombre `6` de notre exemple). D'où la qualification que j'utilisais en introduction à cet article de langage utilisant une notation polonaise inverse sur pile. La notation polonaise faisant référence au travail du mathématicien polonais Jan ¤ukasiewicz qui, désireux de se débarrasser de ces inesthétiques parenthèses, démontra qu'il était possible de les supprimer en plaçant les opérateurs avant les opérandes. L'expression mathématique précédente s'écrivant alors ainsi : « x - 10 4 7 ». Ce que le langage LISP traduira plus tard en rajoutant des parenthèses (un comble, le pauvre Jan doit faire la toupie dans sa tombe) : `(* (- 10 4) 7)`. PostScript quant à lui place l'opérateur après les opérandes et introduit l'usage de la pile pour stocker les opérandes et résultats intermédiaires. Nous voici de nouveau débarrassés des parenthèses.

Il faut s'y faire, j'en conviens... mais, personnellement, je suis fan !

3.1.1 Manipuler la pile des opérandes

Lorsque vous êtes dans l'interpréteur de Ghostscript, les procédures `==` et `pstack` vous permettent de visualiser le contenu de la pile. `==` affiche l'élément du haut de la pile, mais, attention, il consomme cet élément (il le retire de la pile) alors que `pstack` affiche le contenu de toute la pile (l'élément le plus récent en haut) sans modifier celle-ci.

Lorsque la pile n'est pas vide, l'invite de commandes de Ghostscript indique le nombre d'éléments présents dans la pile entre les caractères `<` et `>`. Démonstration :

```
GS>10
GS<1>4
GS<2>pstack
4
10
GS<2>sub
GS<1>pstack
6a
GS<1>7
GS<2>pstack
7
6
GS<2>mul
GS<1>==
42
GS>
```

42 est donc bien la réponse... à notre expression mathématique.

D'autres procédures sont utiles pour manipuler la pile :

- **clear** vide la pile ;
- **count** retourne (sur la pile) le nombre d'éléments présents dans la pile ;
- **pop** dépile le premier élément (il est supprimé) ;
- **exch** échange la place des deux premiers éléments ;
- **dup** duplique le premier élément de la pile ;
- **copy** consomme un entier et permet de dupliquer un nombre arbitraire des premiers éléments de la pile, **1 copy** est équivalent à **dup** ;
- **index** consomme également un entier et est utilisé pour dupliquer n'importe quel élément de la pile pour le placer en haut de celle-ci, **0 index** est équivalent à **dup** (comme tout langage informatique qui se respecte, PostScript compte à partir de **0**) ;
- **roll** retire deux entiers de la pile **n** et **j** et effectue **j** rotations des **n** premiers éléments de la pile, l'élément en tête étant déplacé vers le fond de la pile si **j** est positif et dans le sens inverse sinon (je vous laisse deviner ce qui se passe si **j** est nul).

Quelques exemples pour illustrer ces trois dernières procédures :

```
GS>11 22 33
GS<3>2 copy  % duplique les 2 premiers elements
GS<5>pstack
33
22
33
22
11
GS<5>clear
GS>11 22 33 44
GS<4>2 index  % duplique le troisieme element
GS<5>pstack
22
44
33
22
11
GS<5>clear
GS>11 22 33 44
GS<4>4 2 roll  % permute 2 fois les 4 premiers elements
GS<4>pstack
22
11
44
33
GS<4>4 -1 roll  % permute -1 fois les 4 premiers elements
GS<4>pstack
33
22
11
44
```

3.2 La pile des dictionnaires

La seconde pile très employée par PostScript est la pile des dictionnaires qui, contrairement à la pile des opérandes, n'empile qu'un seul type d'objets : je vous le donne en mille, des dictionnaires. Les dictionnaires peuvent également être empilés sur la pile des opérandes.

Un dictionnaire associe des noms à des valeurs. Dans un autre langage, nous appellerions cela un tableau associatif. Il s'agit d'un type composé.

Lorsque PostScript cherche à interpréter un nom sous sa forme non littérale, il commence par chercher ce nom dans le dictionnaire placé en haut de la pile des dictionnaires. S'il ne l'y trouve pas, il continue sa recherche dans le dictionnaire placé en seconde position, et ainsi de suite jusqu'à ce qu'il trouve le nom dans l'un des dictionnaires ou qu'il arrive au dernier dictionnaire de la pile sans avoir trouvé ce nom dans aucun des dictionnaires. Dans ce dernier cas, l'erreur **/undefined** est générée.

Par défaut, au démarrage de l'interpréteur il existe trois dictionnaires sur la pile des dictionnaires. Du haut vers le bas : **userdict**, **globaldict** et **systemdict** (ce dernier étant en lecture seule). **systemdict** contient toutes les procédures intégrées à l'interpréteur PostScript, notamment **sub** et **mul** que nous avons rencontrées dans la section précédente. Mais, comme il est placé tout en bas de la pile, ce sera le dernier endroit où PostScript ira chercher un nom. Il est donc possible de redéfinir toutes les procédures standards de PostScript (tousse… tousse…).

Bien que, pour le langage, il n'existe aucune différence entre les procédures définies dans le dictionnaire **systemdict** et celles qui le sont dans un autre dictionnaire, dans la suite de ces articles, j'utiliserai le terme « opérateur » pour qualifier les procédures standards du langage, c'est-à-dire celles qui sont définies dans le dictionnaire **systemdict**. Le mot « procédure » fera donc, lui, référence aux procédures que nous aurons écrites nous-mêmes.

L'opérateur **def** permet d'associer un nom à un élément (valeur). Il prend deux éléments sur la pile (des opérandes) : un nom littéral et un élément de n'importe quelle nature et stocke l'association entre ce nom et cette valeur dans le dictionnaire qui se trouve en haut de la pile (des dictionnaires). Elle ne retourne rien sur la pile. Par exemple, pour associer la valeur **0,7** (un nombre réel) au nom **epaisseur_tige** dans le dictionnaire courant (**userdict** par défaut au démarrage de l'interpréteur) :

```
/epaisseur_tige 0.7 def
```

Avec la procédure **begin**, il est possible d'empiler un dictionnaire personnalisé sur la pile des dictionnaires pour y stocker des variables et implémenter ainsi ce que nous appellerions un espace de nom dans un autre langage. La procédure **end** dépile (supprime) le dictionnaire qui se trouve en haut de la pile des dictionnaires fermant ainsi l'espace de nom.

Il existe deux méthodes pour créer un dictionnaire :

- avec l'opérateur **dict** qui consomme un entier **n** sur la pile et y dépose un dictionnaire vide ayant la capacité de stocker **n** associations entre nom et valeur ;
- en utilisant la construction **<< >>** qui crée un dictionnaire à partir d'un ensemble de couples nom littéral/valeur.

Par exemple :

```
GS<< /glop true /pasglop false /glopglop (super!) >>
GS<1>pstack
-dict-
GS<1>begin  % Le dictionnaire est empilé sur la pile des
dictionnaires
GS>glop ==
true
GS>glopglop ==
(super!)
GS>end  % Le dictionnaire est retiré de la pile des
dictionnaires
GS>glop
Error: /undefined in glop
.../...
```

Notez, dans cet exemple, l'introduction du type simple booléen.

Je ne sais pas pour vous, mais moi je trouve ce procédé simple, cohérent et particulièrement élégant. Cependant, pour des raisons qui sortent du cadre de cet article, il ne faut pas trop abuser des dictionnaires. Il serait par exemple tentant d'activer un nouveau dictionnaire au début de chaque procédure, permettant ainsi de définir un espace de noms spécifique et vierge à chaque appel de la procédure (contrairement à beaucoup de langages, il n'existe pas de notion de portées des variables limitées aux procédures ou blocs en PostScript). Ceci serait en fait désastreux en terme de performance et d'usage de la mémoire. Il est bien plus efficace d'utiliser la pile pour stocker des résultats intermédiaires, mais il est vrai que cela devient complexe lorsqu'il faut jongler avec beaucoup de valeurs temporaires.

Deux dernières remarques concernant les dictionnaires : ils sont également employés à d'autres usages que la pile des dictionnaires, par exemple les polices de caractères dans PostScript sont des dictionnaires. Les dictionnaires sont des types composés, au même titre que les chaînes de caractères, nous rencontrerons dans la section qui suit deux autres représentants de la famille des types composés.

4 Tableaux et procédures

En plus des chaînes et des dictionnaires, il existe deux autres formes de types composés : les tableaux et les tableaux exécutables (procédures). En réalité, il y en a encore au moins quatre autres, mais nous ne les aborderons pas ici.

4.1 Tableaux

Un tableau est simplement un ensemble ordonné d'éléments. Le type de ces éléments peut ne pas être homogène. En PostScript, les tableaux n'ont qu'une seule dimension, mais il est possible de créer un tableau qui contient des tableaux pour émuler des tableaux à plusieurs dimensions.

Un tableau peut être construit avec les caractères **[** et **]** :

```
/Peace
[ (peace ) (paix ) (shalom ) (salam )
  (paz ) (paco ) (peoc'h ) (fafa ) ]
def
```

Ces quelques lignes construisent un tableau contenant huit chaînes exprimant le mot « paix » en anglais, français, hébreux, arabe, espagnol, espéranto, breton et mina (sud Togo) et l'associe dans le dictionnaire courant au nom **Peace**.

Il y a une particularité étrange, mais importante concernant les tableaux : comme tous les types composés, un tableau est un pointeur vers une zone en mémoire et modifier un tableau revient donc à modifier cette zone de mémoire. Pour illustrer cela, nous allons utiliser l'opérateur **put** qui consomme sur la pile un tableau, un entier **n** et un élément et qui remplace l'élément en énième position dans le tableau par l'élément pris sur la pile. Étonnamment, cet opérateur ne remet rien sur la pile. Vous pourriez donc imaginer que le travail réalisé sur tableau est perdu. L'explication suit, via cet exemple :

```
GS>[ 1 2 3 4 ]
GS<1>dup
GS<2>pstack
[1 2 3 4]
[1 2 3 4]
GS<2>1 99 put
GS<1>pstack
[1 99 3 4]
```

L'opérateur **dup** semble avoir dupliqué le tableau. Cependant, un tableau étant une référence vers une zone de mémoire, c'est cette référence que nous avons dupliquée. Tout à fait comme un pointeur en C. L'opérateur **put** a donc agit non pas sur le tableau stocké sur la pile (puisque les objets de type composé ne sont pas stockés sur la pile), mais sur le tableau stocké dans une zone de mémoire référencée par un pointeur présent sur la pile.

Petite remarque au passage : nul besoin avec PostScript de gérer l'allocation et la libération de la mémoire. L'interpréteur s'en charge avec, notamment, un mécanisme pour récupérer l'espace associé à des objets de type composé qui ne sont plus référencés (« *garbage collector* »).

Un dernier mot sur les caractères **[** et **]** : bien qu'il s'agisse dans la grammaire PostScript de représentants des quelques rares caractères à la signification spéciale, ils se comportent en réalité comme des procédures. Le premier, **[**, correspond à l'opérateur **mark**. Cet opérateur dépose sur la pile un objet de type simple marque (« *mark* ») dont le domaine de définition ne contient que la valeur **mark**. Le caractère **]** quant à lui correspond à un opérateur qui, contrairement à **[**, ne possède pas un autre nom. Celui-ci consomme sur la pile tous les objets jusqu'au premier objet marque (donc forcément la valeur **mark**) rencontré et dépose sur la pile un tableau contenant tous ces objets. Nous pourrions donc construire un tableau ainsi :

```
GS>mark 1 2 3 4 ]
GS<1>pstack
[1 2 3 4]
```

La seule différence entre le caractère **[** et l'opérateur **mark** réside dans l'espace qui peut suivre ou non le caractère **[** alors qu'il est requis après le nom **mark** comme après tous les noms.

Le comportement de la séquence de caractères **<<** qui introduit un dictionnaire est exactement identique à celui du caractère **[** :

```
GS>[
GS<1><<
GS<2>pstack
-mark-
-mark-
```

Si bien que nous pourrions construire un tableau en utilisant la syntaxe suivante tout aussi contre-intuitive que valide dans le langage :

```
GS><< 123 ]
GS<1>pstack
[1 2 3]
```

4.2 Procédures

Une autre forme de tableau est le tableau exécutable qui est construit avec les caractères **{** et **}**. Cette syntaxe construit un tableau tout comme les caractères **[** et **]** à une différence près : l'évaluation des noms qui sont compris entre ces caractères est déférée. Pour le reste, c'est un tableau et nous pouvons le manipuler comme tel :


```
GS>{ 1 2 add }
GS<1>pstack
{1 2 add}
GS<1>dup 1 99 put
GS<1>pstack
{1 99 add}
GS<1>exec
GS<1>==
100
```

L'opérateur **exec** force l'évaluation de chacun des membres du tableau exécutable qui se trouve en haut de la pile.

Les procédures anonymes sont utiles (notamment pour les structures de contrôle que nous explorerons dans très peu de lignes), mais les procédures nommées le sont encore plus. Pour créer une procédure nommée, il suffit de l'associer à un nom en utilisant l'opérateur **def**. Par exemple, cette procédure peut être utilisée pour convertir des millimètres en points pica :

```
GS>/mm
GS<1>{ 360 mul 127 div }
GS<2>def
GS>100 mm ==
283.464569
```

Nous l'utiliserons d'ailleurs de manière intensive.

4.3 Opérateurs standards

Les procédures standards (celles que je nomme « opérateur ») sont définies dans le dictionnaire **systemdict**, celui qui est tout en bas de la pile des dictionnaires. Il existe des procédures équivalentes aux opérateurs des autres langages :

- arithmétiques : **add**, **sub**, **mul**, **div**, **idiv** (division euclidienne), **mod** (reste de la division euclidienne), **abs**, **neg**, **ceiling**, **floor**, **round**, **truncate** et **sqrt** ;
- logarithmiques et trigonométriques : **exp**, **ln**, **log**, **sin**, **cos**, et **atan** ;
- pour la gestion des nombres aléatoires : **rand**, **srand** et **rrand** ;
- relationnels : **eq** (égalité), **ne** (inégalité), **gt** (supérieur), **ge** (supérieur ou égal), **lt** (inférieur), **le** (inférieur ou égal), **and** (et logique), **or** (ou logique), **xor** (ou exclusif) et **not** ;
- pour manipuler les types composés : **get**, **put**, **copy**, **length**, **forall**, **getinterval** et **putinterval** ;
- et bien d'autres…

Nous pouvons commencer à nous amuser avec tous ces nouveaux jouets. Par exemple, pour extraire un élément au hasard d'un tableau, nous pourrions écrire ceci :

```
Peace dup length rand exch mod get
```

Peace place sur la pile le tableau que nous lui avons associé précédemment et **dup** le duplique (plus exactement, il duplique la référence qui pointe vers ce tableau). **length** prend sur la pile un objet de type composé et y dépose sa taille. **rand** dépose sur la pile un nombre aléatoire compris entre 0 et 231-1, nous échangeons sa position sur la pile avec la taille du tableau en utilisant l'opérateur **exch** et nous obtenons le reste de leur division entière avec **mod**. Ce reste sera aléatoirement compris entre 0 et le nombre d'éléments contenus dans le tableau moins un. À ce stade, il ne reste plus sur la pile que le tableau et ce nombre aléatoire et nous utilisons simplement **get** qui consomme ces deux éléments et dépose sur la pile l'objet qui est à la position référencée dans le tableau.

4.4 Structures de contrôle

En PostScript, rappelons-le, il n'y a pas de mots réservés. Les structures de contrôle sont des procédures comme les autres. C'est pour cela que nous les abordons dans la section relative aux procédures.

4.4.1 Exécution conditionnelle

Deux opérateurs PostScript gèrent l'exécution conditionnelle : **if** et **ifelse**. Le premier prend deux opérandes sur la pile : un booléen et une procédure anonyme qui est évaluée uniquement si la valeur du booléen est **true**. Le second, étonnamment, consomme trois opérandes : un booléen et deux procédures anonymes, la première est exécutée si la valeur du booléen est **true** et la seconde l'est dans le cas contraire.

L'inconvénient de ces opérateurs, c'est qu'ils ne prennent comme argument que des procédures (en plus du booléen). Si j'ai besoin de choisir entre deux nombres en fonction du résultat d'un test, je serais obligé d'écrire quelque chose de semblable à cela :

```
{-15} {105} ifelse
```

Ce qui laissera **-15** en haut de la pile si celle-ci contenait la valeur **true** et **105** sinon.

4.4.2 Boucles

Nous trouverons en PostScript toutes les boucles classiques. Les deux différences importantes avec les autres langages,

vous les connaissez déjà : d'une part les instructions qui implémentent les boucles sont des procédures et non des mots clefs réservés et, d'autre part, ces procédures prennent leurs arguments sur la pile. Le corps des boucles est donc fourni sous la forme d'un tableau exécutable (une procédure anonyme) sur la pile. Exactement comme cela était le cas pour les structures de contrôle **if** et **ifelse**.

La boucle la plus simple est réalisée par l'opérateur **repeat** qui prend sur la pile un entier et une procédure. Cette dernière est exécutée autant de fois que spécifié par l'entier. Par exemple, pour appeler six fois la procédure associée au nom **petale** et effectuer après chaque exécution une rotation du plan de 60 degrés (nous y reviendrons dans la seconde partie de cet article) :

```
6 { petale 60 rotate } repeat
```

Le second type de boucle, très classique, est celui qui est exécuté par l'opérateur **loop** tant que l'opérateur **exit** n'est pas rencontré dans le corps de la boucle (utile lorsque le nombre d'itérations n'est pas connu à l'avance). Par exemple, cette construction permet d'exécuter la boucle tant que le nombre en haut de la pile est strictement positif tout en lui retranchant la valeur de quatre millimètres à chaque itération :

```
{
  % bosse, bosse...
  4 mm sub
  dup 0 le { exit } if
  % bosse, bosse...
} loop
```

Une autre forme, tout aussi classique, est la boucle **for**. L'opérateur **for** prend sur la pile quatre opérandes : une valeur de début, un incrément, une valeur de fin et une procédure anonyme. Cette dernière sera exécutée autant de fois qu'il faut pour aller de la valeur de début à la valeur de fin par pas de l'incrément. De plus, la valeur courante sera déposée sur la pile avant chaque itération.

Le bout de code suivant exécute huit fois les instructions situées entre les caractères **{** et **}**. L'indice qui ira de **0** à **7** par incrément d'une unité sera déposé sur la pile avant chaque itération. La procédure anonyme utilisera cet indice pour exécuter la procédure **vert1** ou **bleu1** chacune à leur tour : **2 mod** réalise une division entière par deux du nombre en haut de la pile et retourne le reste qui est **0** si le nombre est pair et **1** sinon, **0 eq** déterminant cela. **{vert1}** est donc évalué par la procédure **ifelse** si l'indice de l'itération est pair, **{bleu1}** l'est dans le cas contraire.

```
0 1 7 {
  2 mod 0 eq {vert1} {bleu1} ifelse
  % bosse, bosse...
} for
```

Par souci d'exhaustivité, je cite brièvement l'opérateur **forall** qui permet d'exécuter une série d'instructions pour chacun des éléments d'un type composé (chaîne de caractères, tableau, dictionnaire).

Pour en finir avec les itérations, abordons le concept de récursivité (technique chère à mon cœur, s'il en est). Il ne s'agit pas à proprement parler de boucle, mais où en parler sinon ? Et pour l'illustrer, je ferais preuve d'un manque total d'imagination et utiliserais l'exemple éculé de la fonction factorielle dont voici une version PostScript :

```
/fact {
  dup 1 gt
  { dup 1 sub fact } { 1 } ifelse
  mul
} def
```

La première ligne de la fonction duplique le nombre en haut de la pile pour tester s'il est strictement plus grand que **1**. Si c'est le cas, la procédure **fact** est appelée avec comme argument une copie du nombre diminué de **1**, sinon le nombre **1** est déposé sur la pile. Ensuite, les deux nombres de la pile sont multipliés entre eux.

Conclusion

Il est évident qu'il était impossible d'aborder tous les aspects du langage dans ces quelques pages. Néanmoins, je crois avoir présenté les aspects les plus caractéristiques et intéressants. Nous verrons dans un prochain article les techniques pour dessiner et écrire sur des pages, ce qui est bien là l'objet principal de PostScript.

En attendant, pour aller plus loin, je vous recommande la lecture d'une introduction à PostScript, le livre bleu d'Adobe **[3]**. ■

Références

[1] Adobe Systems Incorporated, « *PostScript Language Reference, Third Edition* », http://www.adobe.com/products/postscript/pdfs/PLRM.pdf

[2] Adobe Systems Incorporated, « *PostScript Language Reference Supplement for Version 3010 and 3011* », http://partners.adobe.com/public/developer/en/ps/PS3010and3011.Supplement.pdf

[3] Adobe Systems Incorporated, « *The PostScript Language Tutorial and Cookbook* » (le « Livre bleu »), http://partners.adobe.com/public/developer/en/ps/sdk/sample/BlueBook.zip

PLANIFICATEUR – LES OUTILS

par **Guillaume Lelarge** *[Consultant technique chez Dalibo - Contributeur à pgAdmin]*

Maintenant que nous avons vu tous les types de nœuds disponibles pour le travail du planificateur de requêtes, et que la commande EXPLAIN et sa sortie n'ont plus de secrets pour nous, il nous reste à voir les outils intéressants à connaître dans le contexte des plans d'exécution. Ils ne sont pas nombreux. Il y a pgAdmin, le site explain.depesz.com et l'extension explanation.

Ces trois outils ont des buts différents. L'outil d'administration **pgAdmin** nous permet de mieux comprendre le plan d'exécution, le site explain.depesz.com nous facilite la détection des nœuds réellement problématiques et l'extension **explanation** décode pour nous les différents éléments d'un plan pour pouvoir ne récupérer que les informations qui nous intéressent.

1 Le plan graphique proposé par pgAdmin

pgAdmin est un outil d'administration très connu et très utilisé dans le monde des administrateurs de bases de données PostgreSQL. Une de ses spécificités vient d'une fonctionnalité assez unique dans ce type d'outils pour PostgreSQL : le plan graphique d'exécution d'une requête.

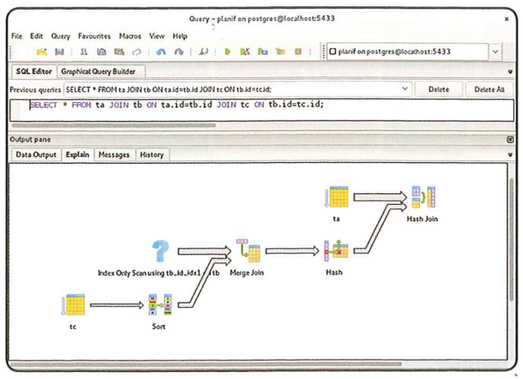


Fig. 1 : Fenêtre d'exécution de requêtes de pgAdmin affichant graphiquement l'exécution d'un plan d'exécution.

L'éditeur de requêtes permet de demander à pgAdmin de décoder le retour de la commande `EXPLAIN` pour afficher le plan sous une forme graphique. Pour cela, il faut aller dans l'éditeur de requêtes, saisir une requête à exécuter (sans l'instruction `EXPLAIN`, car pgAdmin la rajoute de lui-même), puis cliquer sur le bouton d'EXPLAIN graphique.

À partir de là, pgAdmin ajoute l'instruction `EXPLAIN` à la requête, avec les options sélectionnées dans le menu Query. Il décode le résultat et affiche le plan sous une forme graphique équivalente au graphe montré en figure 1.

Le résultat ne s'affiche pas dans l'onglet *« Data Output »* comme habituellement, mais dans le deuxième onglet, nommé *« Explain »*. Cependant, il est à noter que le premier onglet contient malgré tout le résultat de la commande `EXPLAIN`, mais au format texte.

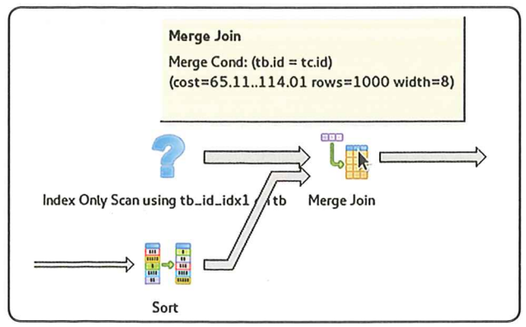


Fig. 2 : Pop-up d'information sur un nœud spécifique du plan d'exécution.

Pour en revenir au graphe, chaque nœud a sa propre icône. Les derniers types de nœuds ne disposent pas forcément d'icônes spécifiques, auquel cas elles sont affichées avec un gros point d'interrogation (c'est le cas du nœud *« Index Only Scan »* en figure 1). Les informations de chaque nœud sont affichées dans une pop-up qui apparaît une fois que la souris est placée sur le nœud, comme le montre la figure 2.

À ce niveau-là, les informations ne sont plus réellement décodées. Tout ce qui concerne un nœud est affiché pour ce nœud. Ainsi, les informations supplémentaires des versions suivantes ne demandent pas une modification de pgAdmin pour être visibles.

La taille des flèches a une signification : elle correspond à la quantité de données envoyées d'un nœud à un autre, autrement dit le nombre de lignes ainsi que la largeur des lignes. Dans le plan en figure 1, on voit bien que le parcours de la table `ta` renvoie plus de données que le parcours de la table `tb` et beaucoup plus que le parcours de la table `tc`. L'avantage de cet affichage est de montrer rapidement par quels nœuds l'exécution commence, par quel nœud elle finit, et le lien entre chaque nœud. Cependant, il est difficile à partir d'une telle représentation de savoir quoi faire pour améliorer les performances.

2 Le site explain.depesz.com

Hubert « depesz » Lubaczewski a été confronté à ce problème. Comme aucune autre solution n'existait, il a codé sa propre solution. Elle est composée de deux parties :

- un module Perl de décodage d'un plan d'exécution tel qu'il est donné par la commande `EXPLAIN` ;
- un site Web, basé sur `mojolicious` (donc aussi en Perl).

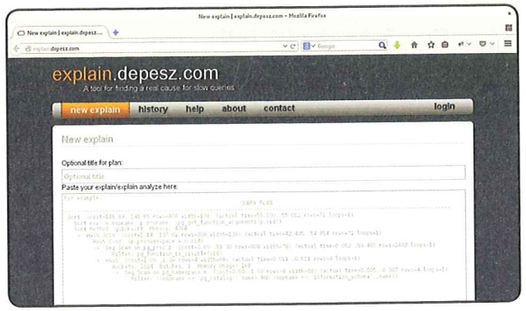


Fig. 3 : Page d'accueil du site explain.depesz.com.

Il propose ces deux parties sur des dépôts GitHub. Cependant, pour faciliter leur utilisation, il héberge le site Web http://explain.depesz.com qui utilise le code en question.

Pour l'utiliser, il suffit d'aller sur cette adresse. La page affichée ressemble à celle de la figure 3.

La première action à réaliser est d'ajouter un compte utilisateur. Ce n'est pas obligatoire, mais cela permettra d'avoir accès plus facilement à son historique de plans. Pour ajouter un compte, il faut cliquer sur le bouton ***Login***, ce qui amène à la page indiquée en figure 4.

Après avoir cliqué sur la case à cocher *« I want to register new account »*, il suffit de remplir les informations très standards : nom (*« username »*) et deux fois le mot de passe (*« password »*) pour confirmation. Après un clic sur le bouton ***Register***, le compte est créé et utilisable. Le menu change avec deux nouveaux éléments : *« user:nom_utilisateur »* et *« plans »*. Le premier permet de changer son mot de passe et de se déconnecter alors que le second est un historique des plans de l'utilisateur connecté.

Commençons par demander l'analyse d'un plan. L'écran d'accueil permet de copier le plan d'exécution et de configurer certaines options. Il faut donc avoir exécuté la commande `EXPLAIN` avec au minimum l'option `ANALYZE` activée et sans désactiver les options `COSTS` et `TIMING`, et copier le résultat de cette commande sur le gros champ de texte. En option, il est possible :

- de donner un titre au plan ;
- de rendre visible ou non le plan sur la page d'historique (activé, donc visible par défaut, autrement dit l'accès au plan est public) ;
- d'anonymiser les objets indiqués dans le plan (désactivé par défaut).

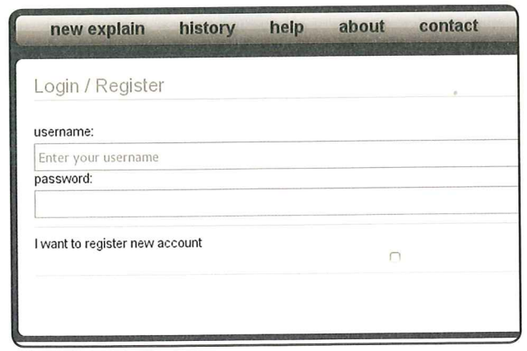


Fig. 4 : Formulaire de connexion/création de comptes.

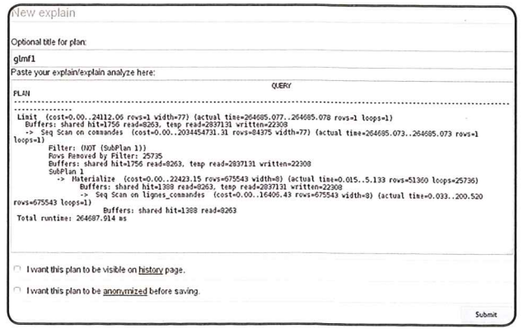


Fig. 5 : Page d'accueil avec les différentes informations saisies.

Pour donner un avis personnel, je ne saisis jamais de titre, je rends le plan toujours invisible (ce sont souvent des plans de requêtes de clients, et, même si le risque est très léger, je ne conçois pas de prendre ce risque sans en avertir mes clients), et je ne demande jamais l'anonymisation.

Ce dernier point mérite peut-être une explication. L'anonymisation semble une bonne idée. Cependant, une fois le plan anonymisé, il devient très difficile de savoir quelle table ou quel index est concerné par tel ou tel nœud. Cela devient donc plus un frein qu'une aide. Et dans le cas où un strict respect de la confidentialité du schéma est nécessaire, autant installer une version du site en local (nous expliquerons plus tard comment faire).

Enfin, concernant la forme du plan, ce site est assez permissif. Il est possible de donner le plan dans les différents formats supportés par la commande `EXPLAIN`, à savoir texte, XML, JSON et YAML. Il est même possible de copier directement le texte provenant d'un `EXPLAIN` exécuté dans pgAdmin (la différence tient dans le fait que pgAdmin ajoute des guillemets doubles à chaque ligne).

Donc une fois toutes les informations saisies, l'écran ressemble à celui de la figure 5.

Après avoir cliqué sur le bouton ***Submit***, nous obtenons l'écran de la figure 6.

Tout en haut se trouve le hachage du nom du plan (ici **hRwa**). Ce dernier sert de lien partageable avec qui l'on souhaite. Le nom du plan est rappelé après le hachage.

Le gros cadre vert fournit un lien qui permet de supprimer ce plan de la base du site. Il n'est fourni qu'une seule fois, et il est donc nécessaire de l'enregistrer pour pouvoir supprimer le plan plus tard. Cependant, si vous êtes connecté via votre utilisateur, il sera toujours possible de le supprimer dans votre page d'historique des plans. Sous le cadre vert se trouve un bouton ***options*** dont on parlera plus tard. Enfin, nous avons un cadre avec trois onglets : *« HTML »*, *« TEXT »* et *« STATS »*.

Le premier onglet est le plus intéressant. Il montre un tableau contenant le plan d'exécution décodé. La colonne

Result: hRwa : glmf1

To delete this plan, you can use this link.

This link will not be shown any more, so you might want to bookmark it, just in case.

options

HTML TEXT STATS

Did it help? Consider supporting us - Bitcoin address: 12v2hUztAk2LgzQ9H9LMwuU32urHMjZQnq

#	exclusive	inclusive	rows x	rows	loops	node
1.	0.005	264685.078	↑ 1.0	1	1	→ Limit (cost=0.00..24112.06 rows=1 width=77) (actual time=264685.077..264685.078 rows=1 loops=1) Buffers: shared hit=1756 read=8263, temp read=2837131 written=22308
2.	132582.185	264685.073	↑ 84375.0	1	1	→ Seq Scan on commandes (cost=0.00..2034454731.31 rows=84375 width=77) (actual time=264685.073..264685.073 rows=1 loops=1) Filter: (NOT (SubPlan 1)) Rows Removed by Filter: 25735 Buffers: shared hit=1756 read=8263, temp read=2837131 written=22308
3.						SubPlan (forSeq Scan)
4.	131902.368	132102.888	↑ 13.2	51360	25736	→ Materialize (cost=0.00..22423.15 rows=675543 width=8) (actual time=0.015..5.133 rows=51360 loops=25736) Buffers: shared hit=1388 read=8263, temp read=2837131 written=22308
5.	200.520	200.520	↑ 1.0	675543	1	→ Seq Scan on lignes_commandes (cost=0.00..16406.43 rows=675543 width=8) (actual time=0.033..200.520 rows=675543 loops=1) Buffers: shared hit=1388 read=8263

Fig. 6 : Plan d'exécution affiché par le site explain.depesz.com.

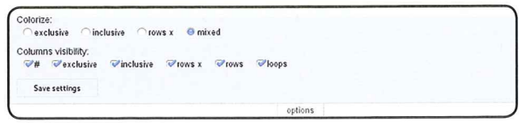


Fig. 7 : Formulaire des options disponibles.

« node » contient un nœud du plan d'exécution par ligne du tableau. Les informations proviennent directement du décodage du plan, sans ajout ou modification. Les colonnes *« inclusive »*, *« rows »* et *« loops »* contiennent aussi des informations provenant du nœud en question, à savoir, respectivement, la durée d'exécution totale du nœud (et de ses enfants), le nombre de lignes et le nombre de boucles. Encore une fois, ce ne sont pas des informations traitées d'une façon ou d'une autre, elles ont juste été lues à partir du plan d'exécution.

Les deux dernières colonnes, *« rows x »* et *« exclusive »*, sont calculées.

La colonne *« rows x »* est le facteur d'échelle entre le nombre de lignes estimées et le nombre de lignes réelles pour un nœud. Par exemple, au niveau du parcours séquentiel de la table `lignes_commandes`, le planificateur a estimé le nombre de lignes à 675543 et l'exécuteur a bien récupéré 675543 lignes. Le facteur est donc de 1. Par contre, au niveau du nœud *« Materialize »*, le planificateur s'attendait à récupérer 675543 lignes alors que l'exécuteur n'en a lu que 51360. Le facteur est donc de 13,2 (675543 divisé par 51360). Cette information est intéressante pour savoir si les statistiques sont à jour ou non. Cela peut aussi être trompeur. Prenons le cas du parcours séquentiel de la table `commandes`. Le planificateur estime l'ensemble

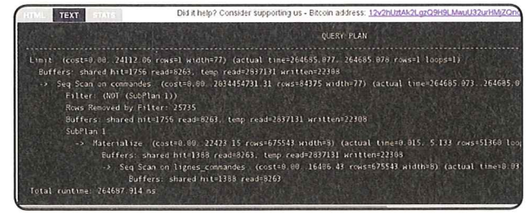


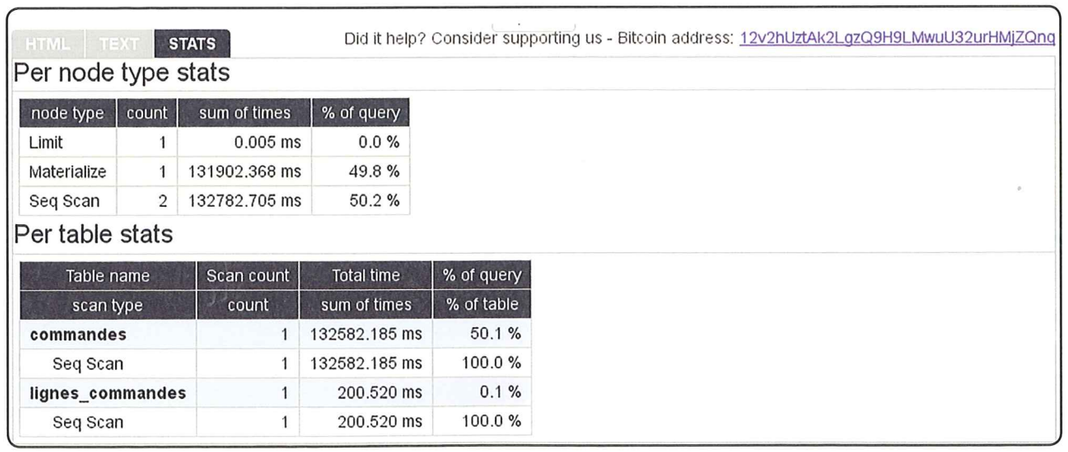
Fig.8 : Contenu de l'onglet « TEXT ».

de résultats à 84375 lignes (ce qui est vrai). Mais l'exécuteur n'en lit qu'une. Ceci est un effet du nœud *« Limit »* dû à la clause `LIMIT 1` de la requête. Le facteur est donc de 84375, mais ce n'est pas une erreur au niveau des statistiques. Il s'agit plutôt d'une limitation du planificateur.

La colonne *« exclusive »* ajoute une information très intéressante. Elle indique la durée d'exécution propre du nœud. Autrement dit, elle ne tient pas compte de la durée des nœuds fils. Par contre, elle tient compte du nombre de boucles. C'est ainsi qu'on s'aperçoit que l'opération *« Materialize »* prend pratiquement autant de temps que le parcours séquentiel de la table `commandes`, alors que celui de la table `lignes_commandes` a une durée d'exécution très inférieure.

À ces informations, le site ajoute une aide sous la forme de couleurs :

- un fond blanc, tout va bien ;
- un fond jaune, c'est inquiétant ;
- un fond orange, c'est très inquiétant ;
- un fond rouge, c'est extrêmement inquiétant.


HTML TEXT STATS

Did it help? Consider supporting us - Bitcoin address: 12v2hUztAk2LgzQ9H9LMwuU32urHMjZQnq

Per node type stats

node type	count	sum of times	% of query
Limit	1	0.005 ms	0.0 %
Materialize	1	131902.368 ms	49.8 %
Seq Scan	2	132782.705 ms	50.2 %

Per table stats

Table name	Scan count	Total time	% of query
scan type	count	sum of times	% of table
commandes	1	132582.185 ms	50.1 %
Seq Scan	1	132582.185 ms	100.0 %
lignes_commandes	1	200.520 ms	0.1 %
Seq Scan	1	200.520 ms	100.0 %

Fig. 9 : Contenu de l'onglet « STATS ».

Grâce à ce jeu de couleurs, on peut immédiatement repérer les nœuds problématiques, notamment sur un plan de plusieurs pages. Les seules options fournies par le site sont disponibles via le bouton ***Options*** qui, une fois cliqué, affiche le formulaire de la figure 7.

Il est donc possible de choisir les colonnes à coloriser et/ou à afficher. Les deux autres onglets peuvent être intéressants :

- l'onglet « *TEXT* » reprend le plan d'exécution tel qu'il a été fourni (voir la figure 8) ;
- l'onglet « *STATS* » donne des statistiques par type de nœud et par objet (voir la figure 9).

Cependant, j'avoue que je n'ai jamais rencontré de cas où ces deux onglets étaient déterminants dans la correction d'une requête lente.

La question la plus fréquente concernant ce site touche à la confidentialité des informations fournies. Ce site enregistre le plan d'exécution, ce qui en soi n'est pas alarmant. Ça ne peut que donner une idée partielle du schéma et des types d'informations stockées. Si cela pose un vrai problème de partager ces informations, il est toujours possible d'installer ce site en local. Pour cela, il faut récupérer les deux dépôts Git avec la commande **`git clone`** :

```
$ git clone https://github.com/depesz/Pg--Explain.git
$ git clone https://github.com/depesz/explain.depesz.com.git
```

Ensuite, nous allons installer le module Perl **`Pg--Explain`**. Cela se fait de façon traditionnelle :

- création du script de construction :

```
$ perl Build.PL
Created MYMETA.yml and MYMETA.json
Creating new 'Build' script for 'Pg-Explain' version '0.68'
```

- construction du module :

```
$ ./Build
Building Pg-Explain
```

- test du nouveau module construit :

```
$ ./Build test
t/00-load.t ............................. 1/8 # Testing
Pg::Explain 0.68, Perl 5.018002, /usr/bin/perl
t/00-load.t ............................. ok
...
t/99-manifest.t ......................... skipped: Author
tests not required for installation
All tests successful.
Files=26, Tests=354,  5 wallclock secs ( 0.21 usr  0.04 sys +
3.40 cusr  0.32 csys =  3.97 CPU)
Result: PASS
```

- installation du module construit et testé :

```
$ sudo ./Build install
```

Maintenant, nous devons nous occuper du site web. Nous allons utiliser le serveur web interne de **`mojolicious`** et laissons une installation définitive avec Apache, par exemple, en exercice pour le lecteur.

Cette application Web a besoin d'une base avec un schéma prédéfini. Voici les étapes nécessaires pour la création de cette base et l'ajout de son schéma :

- création de la base :

```
$ createdb explaindb
```

- ajout du schéma :

```
$ psql -f sql/create.sql explaindb
SET
...
REVOKE
psql:sql/create.sql:172: ERROR:  role "pgdba" does not exist
psql:sql/create.sql:173: ERROR:  role "pgdba" does not exist
GRANT
```

- correction du schéma avec les deux fichiers de patch fournis :

```
$ psql -f sql/patch-001.sql explaindb
$ psql -f sql/patch-002.sql explaindb
```

Ensuite, il faut configurer l'application Web. Cela se fait avec le fichier **`explain.json`**. Voici les informations à modifier :

- le DSN (*Data Source Name*) de connexion pour qu'il attaque la base créée (**`explaindb`** pour cet exemple, mais vous pouvez aussi modifier les autres informations suivant votre besoin) ;
- le nom d'utilisateur ;
- les informations de mail (suivant votre cas... ce mail est utilisé en cas de soumission d'un message dans le formulaire de contact).

Cela nous donne le fichier suivant :

```
{
   "title" : "explain.depesz.com",

   "secret" : "|Erp--Wjgb)+eiB/|H=|V7!#+M|L{a8=J2|pd+N1=M|&pJWq
|M&,f3q^XS",

   "database" : {
       "dsn"      : "dbi:Pg:database=explaindb;host=127.0.0.1;p
ort=5436",
       "username" : "postgres",
       "options"  : {
           "auto_commit" : 1,
           "pg_server_prepare" : 0,
           "RaiseError" : 1,
           "PrintError" : 0
       }
   },

   "mail_sender" : {
       "from"    : "explaindb <guillaume.lelarge@dalibo.com>",
       "to"      : "Guillaume Lelarge <guillaume.lelarge@dalibo.
com>",
       "subject" : "Message from: explaindb/contact"
   },

   "hypnotoad"   : {
       "listen"  : ["http://*:12004"],
       "workers" : 2,
       "pid_file" : "/home/depesz/sites/explain.depesz.com.pid"
   }
}
```

Il ne reste plus qu'à lancer l'application Web en mode démon :

```
$ ./explain.pl daemon
Smartmatch is experimental at /usr/local/share/perl5/Mojo/
Collection.pm line 31.
Smartmatch is experimental at /usr/local/share/perl5/Mojo/
Collection.pm line 36.
[Sun May 18 14:37:00 2014] [debug] Reading config file "/home/
guillaume/article/explain.depesz.com/explain.json".
[Sun May 18 14:37:00 2014] [debug] Database connection args: [
  'dbi:Pg:database=explaindb;host=127.0.0.1;port=5436',
  'postgres',
  undef,
  {
    'pg_server_prepare' => 0,
    'PrintError' => 0,
    'auto_commit' => 1,
    'RaiseError' => 1
  }
]

[Sun May 18 14:37:00 2014] [info] Listening at "http://*:3000".
Server available at http://127.0.0.1:3000.
```

À partir de là, il est possible d'accéder à l'application Web via l'URL http://127.0.0.1:3000. Elle fonctionnera exactement comme le site http://explain.depesz.com, mais à partir d'une base locale, donc sans risquer de divulguer des informations sur le schéma et les requêtes.

3 L'extension explanation

L'extension explanation a été écrite par David Wheeler. Elle profite du format XML de la sortie du **`EXPLAIN`** pour décomposer les informations de chaque nœud. L'extension est disponible sur le site http://pgxn.org (créé par le même David Wheeler), ainsi que sur GitHub (https://github.com/pgexperts/explanation/.

Le mieux est actuellement de récupérer le dépôt Git via la commande **`git clone`** standard :

```
$ git clone https://github.com/pgexperts/explanation.git
```

Après être entré dans le répertoire du clone, un simple **`make install`** va installer l'extension dans le bon répertoire (à condition que la commande **`pg_config`** soit accessible pour récupérer le répertoire d'installation des extensions) :

```
$ cd explanation/
$ make install
```

Il ne reste plus qu'à activer l'extension dans les bases de données où elle sera utilisée :

```
tpc=# CREATE EXTENSION explanation;
CREATE EXTENSION
tpc=# \dx+ explanation
            Objects in extension "explanation"
                      Object Description
-------------------------------------------------------------
 function explanation(text,boolean,text[])
 function parse_node(text[],xml,text,interval,trigger_plan[])
 function parse_node(xml,text,interval,trigger_plan[])
 function parse_triggers(xml[])
 type trigger_plan
(5 rows)
```

L'objet qui nous intéresse principalement est la fonction **`explanation`**. Cette dernière accepte trois arguments : la requête pour laquelle on souhaite récupérer le plan d'exécution, un booléen pour activer si nécessaire l'option **`ANALYZE`** (l'option **`BUFFERS`** est automatiquement activée dans ce cas), et enfin un tableau précisant les colonnes à renvoyer. Seul le premier argument est obligatoire. Cela nous donne par exemple ceci :

```
tpc=# SELECT * FROM explanation('SELECT * FROM commandes');
-[ RECORD 1 ]--------+-----------------------------------
----
planned_at           | 2014-05-18 16:22:48.467632+02
node_id              | 04c9f9b30fa0279f20e4d4538d1d7826
parent_id            |
node_type            | Seq Scan
total_runtime        |
strategy             |
operation            |
startup_cost         | 0
total_cost           | 4098.5
plan_rows            | 168750
plan_width           | 77
actual_startup_time  |
actual_total_time    |
actual_rows          |
actual_loops         |
parent_relationship  |
sort_key             | {}
sort_method          | {}
sort_space_used      |
sort_space_type      |
join_type            |
join_filter          |
hash_cond            |
relation_name        | commandes
alias                | commandes
scan_direction       |
index_name           |
index_cond           |
recheck_cond         |
tid_cond             |
merge_cond           |
subplan_name         |
function_name        |
function_call        |
filter               |
one_time_filter      |
command              |
shared_hit_blocks    |
shared_read_blocks   |
shared_written_blocks |
local_hit_blocks     |
local_read_blocks    |
local_written_blocks |
temp_read_blocks     |
temp_written_blocks  |
output               | {numero_commande,client_
id,etat_commande,prix_total,date_commande,priorite_
commande,vendeur,priorite_expedition,commentaire}
hash_buckets         |
hash_batches         |
original_hash_batches |
peak_memory_usage    |
schema               | public
cte_name             |
triggers             | {}
```

Cette fonction décode chaque information comprise dans chaque ligne du plan d'exécution et les renvoie sous la forme d'un ensemble de lignes et de colonnes, chaque colonne correspondant à une information du nœud, chaque ligne correspondant à un nœud. D'où cette ligne immense indiquant que le plan est composé d'un simple parcours séquentiel sur la table **commandes** qui devrait ramener 168750 lignes, etc.

Le gros intérêt de cette fonction est de pouvoir afficher seulement les informations qui nous intéressent. Voici un exemple sur une requête plus complexe :

```
tpc=# SELECT node_type, relation_name, total_cost, plan_rows
FROM
       explanation('SELECT * FROM commandes WHERE numero_
commande NOT IN
                   (SELECT numero_commande FROM lignes_
commandes) LIMIT 1
                   ');

  node_type  |  relation_name   |  total_cost   | plan_rows
-------------+------------------+---------------+-----------
 Limit       |                  |      24112.06 |         1
 Seq Scan    | commandes        | 2034454731.31 |     84375
 Materialize |                  |      22423.15 |    675543
 Seq Scan    | lignes_commandes |      16406.43 |    675543
(4 rows)
```

Il existe néanmoins un gros souci sur cette extension. Elle fonctionne parfaitement jusqu'à la version 9.1 de PostgreSQL incluse. À partir de la 9.2, le plan avec les options **ANALYZE** et **BUFFERS** dispose de balises dont le nom contient un **/** et cela génère une erreur d'analyse du plan au format XML. Ce bug fait l'objet d'un ticket (https://github.com/pgexperts/explanation/issues/1). Espérons qu'il soit vite corrigé.

Conclusion

Ce quatrième et dernier article sur le planificateur nous a permis de faire le tour des outils facilitant la compréhension des plans d'exécution de PostgreSQL. Le plus utile des trois reste néanmoins le site http://explain.depesz.com. Apprenez à bien l'utiliser et vous pourrez plus facilement optimiser vos requêtes SQL. ■