N°169 MARS 2014

GNU LINUX MAGAZINE / FRANCE

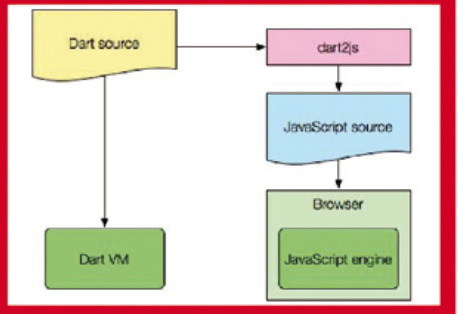
SGBD / ACTU

Découvrez les nouveautés de PostreSQL 9.3 pour les DBA, les utilisateurs et les développeurs p.04

ADMINISTRATION ET DÉVELOPPEMENT SUR SYSTÈMES OPEN SOURCE ET EMBARQUÉS

WEBDEV / JS

Apprenez à développer en Dart, le langage performant et productif de Google pour le Web p.67



SHELL / INSTALL

Créez vos scripts d'installation auto-extractibles avec le shell seul ou en reposant sur makeself p.22

PUPPET / RHEL

Réalisez une extension Puppet : étude de cas concret avec l'intégration du service tuned de RHEL/Fedora p.52

L 19275 - 169 - F: 7,90 € - RD

MAKE / BUILD

Remplacez Make par Redo en tant que système de build minimaliste avec gestion de dépendances p.78

RÉSEAU / DYNAMIQUE

Vous rêvez que votre réseau se configure tout seul ?

ROUTAGE OSPF ...avec BIRD p.30

1. Mise en place du réseau exemple
2. Architecture générale
3. Configuration des routeurs
4. Configuration des serveurs
5. Haute disponibilité en bonus

CLOUD / TUTO

Déployez pas à pas en toute simplicité votre infrastructure « cloud as a service » avec OpenStack p.44

HUMEUR / MATH

Dr KissCool ose le dire : « Non, les maths ne servent à rien en informatique et en programmation ! » p.42

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

Une infogérance de qualité grâce aux solutions Open Source

DBM TECHNOLOGIES

UNE SOCIÉTÉ À TAILLE HUMAINE VOUS OFFRE LA COMPÉTENCE DES GRANDS

Optimisez vos systèmes d'informations

Nous utilisons le meilleur de la technologie

- Virtualisation
- Messagerie / Groupware
- Sécurité réseaux (VPN, Proxy..)
- Téléphonie sur IP

Nous vous offrons des services personnalisables

- Adaptés à vos besoins
- Un suivi client maîtrisé
- Support téléphonique en France
- Développement spécifique

Vous avez un projet ?
Contactez-nous

www.dbmtechnologies.com +33 (0)4 72 83 52 28 contact@dbmtechnologies.com

SOMMAIRE N°169

GNU/Linux Magazine France
est édité par Les Éditions Diamond

LES ÉDITIONS DIAMOND

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

Directeur de publication : Arnaud Metzler
Rédacteur en chef : Denis Bodor
Réalisation graphique : Jérémy Gall

Responsable publicité : Valérie Fréchard, Tél. : 03 67 10 00 27
v.frechard@ed-diamond.com
Service abonnement : Tél. : 03 67 10 00 20
Impression : pva, Druck und Medien-Dienstleistungen GmbH, Landau, Allemagne

Distribution France : (uniquement pour les dépositaires de presse)

MLP Réassort : Plate-forme de Saint-Barthélemy-d'Anjou. Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier. Tél. : 04 74 82 63 04

IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : À parution, N° **ISSN** : 1291-78 34
Commission paritaire : K78 976

Périodicité : Mensuel
Prix de vente : 7,90 €

MIXTE Papier issu de sources responsables FSC® C015136

Membre April

ÉDITORIAL

Les forces de l'alliance risquent d'envahir le monde... de l'automobile !

J'adore ! Sérieusement, j'adore ce genre d'effet d'annonce, en particulier lorsqu'il est possible d'y glisser quelques jeux de mots. Les forces de l'alliance font ici référence à la *GENIVI Alliance*, un consortium à but non lucratif (sic) entre certains acteurs du secteur automobile (BMW, PSA Peugeot Citroën) et d'autres provenant de l'IT/multimédia (Intel, Wind Rivers, Delphi (l'équipementier, pas l'IDE)). L'objectif annoncé est, entre autres, de mettre en place un programme de certification autour de son projet FOSS (*Free and Open-Source Software*) basé sur GNU/Linux.

Récemment, une étude IHS, se basant sur les activités et mouvements tactiques/stratégiques des différents constructeurs américains et européens, annonce que ceux-ci « s'alignent derrière Linux » pour finalement conclure qu'en 2018 notre cher OS occupera 30% du marché, et plus de 40% en 2020. Le marché est celui du « *automotive infotainment operating systems* » soit, en bon français, « les trucs qui essaient de distraire le conducteur pendant qu'il roule ». Notez que de manière étonnante, c'est Microsoft qui actuellement, en compagnie de QNX, occupe massivement le terrain. Seulement voilà, les constructeurs automobiles souhaitent apparemment reprendre le contrôle et la maîtrise de leurs architectures logicielles, toujours selon IHS.

Je ne sais pas pour vous, mais ce genre d'annonce me laisse de plus en plus de marbre. En effet, il existe une différence importante entre les faits, la perception des faits et la manière dont ces faits s'établissent. Souvenez-nous, il n'y a pas si longtemps, l'arrivée d'un système d'exploitation pour smartphone basé sur GNU/Linux (ou vaguement basé sur quelques composants modifiés du système complet) était également une grande nouvelle... Android, qui maintenant fait aussi l'objet d'un intérêt tout particulier de la part de sociétés comme Audi, Hyundai et Honda, s'est finalement intégré au paysage mais n'a pas changé grand-chose à l'échelle de l'individu.

Certes, des projets comme Replicant ou CyanogenMod permettent à tout un chacun de construire son propre système en logiciel libre, mais à quel coût ?! L'utilisateur lambda, lui, n'y voit pas vraiment de différences. Veut-il d'ailleurs vraiment en voir une ? Certes, le logiciel libre avance, dans tous les domaines, mais les bienfaits pour les utilisateurs sont secondaires, voire difficiles à percevoir. Cela ne revient-il pas à encenser des évolutions dont les fruits ne seront jamais à notre portée, et donc à cautionner des comportements qui nous dépossèdent de l'intérêt primaire du logiciel libre : la maîtrise possible du système par l'utilisateur ? Mieux vaut réfléchir à deux fois avant d'abonder dans le sens de ces « progrès ».

Bon, je dis ça, je dis rien, hein... Je n'ai ni voiture, ni permis. Mais ça me dérangerait tout de même de me faire renverser par une voiture hors de contrôle du fait de l'utilisation brouillonne, car insuffisamment communautaire, de mon OS préféré à un niveau plus proche du matériel que l'*automotive infotainment*...

Je terminerai cet édito en profitant de l'espace qui me reste (Comment ça y'a plus d'espace ? Mais si, en corps 8 ça tient !) pour signaler l'ouverture officielle du service **connect.ed-diamond.com** regroupant la majorité des articles publiés dans les magazines des éditions Diamond, sous la forme d'une base documentaire indexée à destination des professionnels et entreprises. Ceci ayant consommé un temps non négligeable de la rédaction (d'un point de vue archéologique), je me permets ici de faire mention de cet accomplissement.

Sur ce, je vous souhaite une bonne lecture et vous donne rendez-vous le mois prochain...

Denis Bodor

NOUVEAUTÉS DE POSTGRESQL 9.3

par Guillaume Lelarge

À chaque année sa nouvelle version de PostgreSQL. Toutes les versions 9.x de PostgreSQL sont sorties à la rentrée, en septembre pour être précis. Cette version ne déroge pas à la règle. Elle est sortie le 4 septembre. Une première version corrective est sortie le 10 octobre, et une deuxième le 5 décembre. Il est temps de se pencher sur les nouveautés de cette version, qui commence à arriver en production chez de nombreux utilisateurs.

1 De quoi intéresser les DBA

1.1 À l'installation

Commençons par un problème qui a gêné une majorité de nouveaux utilisateurs. Lors de la création d'un répertoire de données avec la commande **initdb**, PostgreSQL essaie de trouver une bonne valeur pour le paramètre **shared_buffers**. Ce paramètre indique la taille de mémoire partagée utilisée pour le cache disque de PostgreSQL (pour les détails, voir le n° 107 de *GNU/Linux Magazine France*).

Historiquement, PostgreSQL passe par l'implémentation System V de la mémoire partagée, car cela lui permet de connaître le nombre de processus attachés à ce segment de mémoire partagée. Le problème de cette implémentation est que la taille maximale de mémoire partagée est configurable et configurée très bas par défaut (32 Mo, sauf pour les distributions Red Hat). Du coup, les DBA indiquant une valeur égale ou supérieure à 32 Mo pour le paramètre **shared_buffers** n'arrivaient plus à démarrer PostgreSQL, le système refusant de fournir autant de mémoire. Il fallait alors configurer le paramètre **SHMMAX** du noyau pour pouvoir démarrer PostgreSQL. Tout le monde est tombé dans ce piège, ce qui a bien aidé à promouvoir l'idée que PostgreSQL n'était pas une base de données pour débutants.

Il existe deux autres implémentations de la mémoire partagée sous Linux, mais personne n'avait encore suggéré un moyen intelligent de conserver la fonctionnalité de décompte de processus attachés à la mémoire partagée et de contourner le problème de la configuration système. Au tout début du développement de la version 9.3, Robert Haas a suggéré un moyen intelligent de le faire : il suffit de réclamer un peu de mémoire partagée sous l'implémentation System V et beaucoup de mémoire partagée sous l'implémentation **mmap**. On obtient ainsi le meilleur résultat possible, en profitant des bonnes fonctionnalités de chaque implémentation. Du coup, la version 9.3 peut utiliser beaucoup plus de mémoire partagée sans avoir à configurer le paramètre **SHMMAX** du noyau. De ce fait, les développeurs ont décidé que PostgreSQL utiliserait maintenant 128 Mo de mémoire cache (donc le paramètre **shared_buffers**) par défaut. Il est évidemment conseillé d'aller bien plus loin si votre mémoire le permet (la règle du un quart de la mémoire totale pour un serveur dédié restant inchangée). Un moyen de s'en rendre compte est de créer un répertoire de données pour une version antérieure (la 9.2 par exemple) et un autre pour la nouvelle version.

Commençons par initialiser les répertoires de données en renvoyant la sortie standard et celle des erreurs dans un fichier que nous regarderons plus tard :

```
$ export PATH=/opt/postgresql-9.2/bin:$PATH
$ initdb -D data92 >initdb92.log 2>&1
$ export PATH=/opt/postgresql-9.3/bin:$PATH
$ initdb -D data93 >initdb93.log 2>&1
```

Ensuite, démarrons la version 9.2 :

```
$ export PATH=/opt/postgresql-9.2/bin:$PATH
$ pg_ctl -D data92 start
server starting
LOG:  database system was shut down at 2013-12-01 15:19:01 CET
LOG:  database system is ready to accept connections
LOG:  autovacuum launcher started
```

Le fichier **postmaster.pid** est créé au démarrage. Il contient bien évidemment le PID du processus maître, mais aussi quelques autres informations comme l'identifiant de sémaphore System V demandé par PostgreSQL en dernière ligne du fichier :

```
$ tail -1 data92/postmaster.pid
  5432001  42008579
```

L'identifiant est le dernier nombre. Si on utilise ce nombre avec la commande **ipcs** pour connaître la quantité de mémoire partagée récupérée, on obtient ceci :

```
$ ipcs | grep 42008579
0x0052e2c1 42008579   guillaume  600        41222144   5
```

La quantité de mémoire prise pour ce segment de mémoire partagée est indiquée par le dernier nombre. Le processus PostgreSQL a donc réclamé 41222144 octets de mémoire partagée, soit 39,3 Mo (mon système a une configuration bien améliorée pour la mémoire partagée System V).

Arrêtons la version 9.2, et faisons le même test avec la version 9.3 :

```
$ pg_ctl -D data92 stop
LOG:  received smart shutdown request
LOG:  autovacuum launcher shutting down
waiting for server to shut down....
LOG:  shutting down
LOG:  database system is shut down
 done
server stopped
$ export PATH=/opt/postgresql-9.3/bin:$PATH
$ pg_ctl -D data93 start
server starting
LOG:  database system was shut down at 2013-12-01 15:19:15 CET
LOG:  database system is ready to accept connections
LOG:  autovacuum launcher started
$ tail -1 data93/postmaster.pid
  5432001  42041347
$ ipcs | grep 42041347
0x0052e2c1 42041347   guillaume  600        48         5
```

La version 9.3 n'a demandé que 48 octets de mémoire partagée System V, bien en-dessous des 32 Mo par défaut permis par le paramètre **SHMMAX**.

Si on regarde la différence dans les traces de **initdb** par rapport au paramètre **shared_buffers**, nous constatons immédiatement la différence sur la configuration par défaut :

```
$ diff initdb92.log initdb93.log | grep shared_buffers
< selecting default shared_buffers ... 32MB
> selecting default shared_buffers ... 128MB
```

On passe bien par défaut de 32 Mo à 128 Mo, même avec un paramètre **SHMMAX** configuré au minimum. Regardons maintenant les autres différences sur les traces de ces deux créations de répertoires de données :

```
$ diff initdb92.log initdb93.log | grep -v data9
8c8,10
---
> Data page checksums are disabled.
>
11c13
< selecting default shared_buffers ... 32MB
---
> selecting default shared_buffers ... 128MB
13c15
---
26a29
> syncing data to disk ... ok
```

Un nouveau message est apparu : **syncing data to disk**. Ce message indique simplement que la commande **initdb** demande au système d'exploitation de vider son cache sur disque immédiatement après la création du répertoire des données. Cette modification était nécessaire pour l'outil **pg_upgrade** (dont on abordera les évolutions plus tard).

On remarque aussi un autre message : **Data page checksums are disabled**. PostgreSQL dispose de sommes de contrôle sur les journaux de transactions, mais pas sur les fichiers de données. La version 9.3 apporte cela. Le but de cette fonctionnalité est d'éviter les corruptions silencieuses, qu'elles soient dues à une personne malintentionnée ou à un matériel défectueux. Cependant, elle est désactivée par défaut : le calcul des sommes de contrôle et leur stockage ont un coût au niveau de la charge CPU et disque, ce qui engendre des performances réduites. Voici ce que cela donne avec une installation sans somme de contrôle :

```
$ initdb -D data 2>&1 | grep checksums
Data page checksums are disabled.
$ pg_ctl -D data start
server starting
LOG:  redirecting log output to logging collector process
HINT:  Future log output will appear in directory "pg_log".

$ psql postgres
psql (9.3.1)
Type "help" for help.

postgres=# CREATE TABLE t1(id integer);
CREATE TABLE
postgres=# \timing
Timing is on.
postgres=# INSERT INTO t1 SELECT generate_series(1, 10000000);
INSERT 0 10000000
Time: 30215.729 ms
postgres=# INSERT INTO t1 SELECT generate_series(1, 10000000);
INSERT 0 10000000
Time: 30745.367 ms
postgres=# INSERT INTO t1 SELECT generate_series(1, 10000000);
INSERT 0 10000000
Time: 30366.370 ms
```

Autrement dit, il faut environ 30,3 secondes pour insérer 10 millions d'entiers. Voici le même test avec les sommes de contrôle :

```
$ pg_ctl -D data stop
waiting for server to shut down..... done
server stopped
$ rm -rf data
$ initdb --data-checksums -D data 2>&1 | grep checksums
Data page checksums are enabled.
$ pg_ctl -D data start
server starting
LOG:  redirecting log output to logging collector process
HINT:  Future log output will appear in directory "pg_log".

$ psql postgres
psql (9.3.1)
Type "help" for help.

postgres=# CREATE TABLE t1(id integer);
```

```
CREATE TABLE
postgres=# \timing
Timing is on.
postgres=# INSERT INTO t1 SELECT generate_series(1, 10000000);
Time: 31132.791 ms
postgres=# INSERT INTO t1 SELECT generate_series(1, 10000000);
INSERT 0 10000000
Time: 31581.761 ms
postgres=# INSERT INTO t1 SELECT generate_series(1, 10000000);
INSERT 0 10000000
Time: 31032.083 ms
```

Ce test n'a évidemment rien de scientifique, mais il montre la (petite) perte de performances suite à la mise en place des sommes de contrôle. Comme d'habitude avec PostgreSQL, toute nouvelle fonctionnalité qui pourrait entraîner des pertes de performances est désactivée par défaut pendant les premières versions où elle apparaît, puis est activée par défaut quand son intérêt se révèle important et que les potentiels problèmes de performances ont été réglés.

1.2 Pour la configuration

La modification importante dans la gestion de la configuration tient dans la directive d'import. Il existait déjà une directive **include** permettant de spécifier un fichier de configuration supplémentaire. Il est maintenant possible de spécifier l'import de fichiers appartenant à un répertoire grâce à la directive **include_dir**. En ajoutant cette directive et en précisant le répertoire de configuration, PostgreSQL ira lire les paramètres se trouvant dans les fichiers de ce répertoire. Pour rappel, si un paramètre est configuré plusieurs fois, que ce soit dans le même fichier ou dans différents fichiers inclus avec les directives **include** ou **include_dir**, seule la dernière configuration sera prise en compte.

Un nouveau paramètre apparaît dans la gestion des verrous. Prenons un exemple simple. Une session ouvre une transaction et lit une table :

```
postgres@session1=# CREATE TABLE t2(id integer);
CREATE TABLE
postgres@session1=# BEGIN;
BEGIN
postgres@session1=# SELECT * FROM t2;
 id
----
(0 rows)
```

La lecture de cette table a posé un verrou *AccessShare* sur la table pour éviter qu'elle soit supprimée le temps de sa lecture. Étant dans une transaction, ce verrou est conservé jusqu'à la fin de cette transaction. Du coup, toute autre session cherchant à récupérer un verrou exclusif sur la table sera bloquée le temps que la transaction de la première session se termine :

```
postgres@session2=# DROP TABLE t2;
```

Et cette commande ne rendra pas la main. Le seul moyen d'annuler automatiquement cette requête après un certain délai revient à utiliser le paramètre **statement_timeout**. Cependant, ce paramètre est assez dangereux : il annulera toute requête dont la durée d'exécution dépasse le délai imparti. Cela sera la cause d'annulation d'un bon nombre d'autres requêtes qui ne sont pourtant pas en attente d'un verrou. La version 9.3 ajoute un nouveau paramètre permettant justement de configurer spécifiquement le délai maximum d'attente d'un verrou. Ce paramètre s'appelle **lock_timeout**. Voici comment on pourrait l'utiliser :

```
postgres=# SET lock_timeout TO '5s';
SET
postgres=# DROP TABLE t2;
ERROR:  canceling statement due to lock timeout
```

Dernière amélioration intéressante sur la gestion de la configuration : nous avons enfin le numéro de la ligne du fichier **pg_hba.conf** impliquée dans l'échec d'une authentification. Voici un exemple :

```
$ psql postgres
Password:
psql: FATAL:  password authentication failed for user "guillaume"
$ tail -3 data/pg_log/postgresql-2013-12-01_160221.log
LOG:  received SIGHUP, reloading configuration files
FATAL:  password authentication failed for user "guillaume"
DETAIL:  Connection matched pg_hba.conf line 84: "local  all  all  md5"
  can listen on multiples unix socket
```

1.3 En ce qui concerne la réplication

En 9.2, quand les journaux changent de ligne de temps (*timeline*), il était impossible à l'esclave de suivre la timeline, sauf si l'esclave était configuré pour récupérer les journaux de transactions archivés en cas de problème. Voici l'exemple typique d'erreurs reçues dans ce genre de circonstances :

```
FATAL:  replication terminated by primary server
LOG:  record with zero length at 0/B000080
FATAL:  timeline 2 of the primary does not match recovery target timeline 1
```

Ce problème est corrigé avec la 9.3. Le protocole de réplication a été modifié pour permettre l'envoi du contenu des fichiers d'historique des lignes de temps. Voici un exemple complet montrant le bon fonctionnement de ce suivi. Nous allons pour cela créer un maître et deux esclaves en cascade. Autrement dit, le maître envoie les informations de réplication au premier esclave, qui les envoie au deuxième esclave.

Voici comment créer le maître :

- Création du répertoire des archives :

```
$ mkdir /home/guillaume/article/archives
```

- Création du répertoire des configurations personnalisées :

```
$ mkdir data/custom_conf
```

- Ajout de la directive **include_dir** pour indiquer le répertoire contenant les configurations personnalisées :

```
$ cat <<_EOF_ >>data/postgresql.conf
> include_dir custom_conf
> _EOF_
```

- Ajout d'un nouveau fichier de configuration sur la réplication :

```
$ cat <<_EOF_ >> data/custom_conf/replication.conf
> wal_level = hot_standby
> archive_mode = on
> archive_command = 'cp %p /home/guillaume/article/archives/%f'
> max_wal_senders = 5
> hot_standby = on
> _EOF_
```

- Ajout d'une autorisation pour la connexion en mode réplication :

```
$ cat <<_EOF_ >> data/pg_hba.conf
> host    replication     guillaume       127.0.0.1/32            trust
> _EOF_
```

- Redémarrage du serveur maître :

```
$ pg_ctl -D data restart
waiting for server to shut down.... done
server stopped
server starting
LOG:  redirecting log output to logging collector process
HINT:  Future log output will appear in directory "pg_log".
```

- Test du bon fonctionnement de l'archivage en forçant le changement (et donc l'archivage) du journal de transactions courant :

```
$ psql postgres -c "SELECT pg_switch_xlog()"
 pg_switch_xlog
---------------
 0/A985BAD0
(1 row)

$ ll archives/
total 16384
-rw-------. 1 guillaume guillaume 16777216 Dec  1 19:28 000000010000000000000A9
```

Maintenant que le maître est bien configuré et que l'archivage se passe bien, créons le premier esclave. Pour cela, nous allons utiliser l'outil **pg_basebackup** apparu en 9.1, mais avec une nouvelle option de la 9.3, **--write-recovery-conf**, qui crée automatiquement le fichier **recovery.conf** suivant les arguments donnés à la commande :

- Lancement de **pg_basebackup** :

```
$ pg_basebackup -h 127.0.0.1 -p 5432 -D data2 --write-recovery-conf -P
1436757/1436757 kB (100%), 1/1 tablespace
```

- Vérification du contenu du fichier **recovery.conf** :

```
$ cat data2/recovery.conf
standby_mode = 'on'
primary_conninfo = 'user=guillaume host=127.0.0.1 port=5432 sslmode=prefer
sslcompression=1'
```

- Il manque le paramètre de suivi de ligne de temps, ajoutons-le :

```
$ cat <<_EOF_ >>data2/recovery.conf
> recovery_target_timeline = 'latest'
> _EOF_
```

- Ajout d'un fichier de configuration pour indiquer un autre numéro de port (ceci est fait car tout est réalisé sur le même serveur et qu'il n'est pas possible d'avoir deux processus écoutant sur le même port TCP) :

```
$ cat <<_EOF_ >data2/custom_conf/port.conf
> port = 5433
> _EOF_
```

- Démarrage de l'esclave :

```
$ pg_ctl -D data2 start
server starting
LOG:  redirecting log output to logging collector process
HINT:  Future log output will appear in directory "pg_log".
```

Testons maintenant qu'une modification sur le maître est bien propagée sur l'esclave :

```
$ psql -p5432 postgres
psql (9.3.1)
Type "help" for help.

postgres=# CREATE TABLE t3(id integer);
CREATE TABLE
postgres=# \q
$ psql -p5433 postgres
psql (9.3.1)
Type "help" for help.

postgres=# \d
        List of relations
 Schema | Name | Type  |   Owner
--------+------+-------+-----------
 public | t1   | table | guillaume
 public | t2   | table | guillaume
 public | t3   | table | guillaume
(3 rows)
```

La nouvelle table est bien apparue immédiatement sur l'esclave. Créons maintenant l'esclave de l'esclave :

- Lancement de **pg_basebackup** :

```
$ pg_basebackup -h 127.0.0.1 -p 5433 -D data3 --write-recovery-conf -P
1436749/1436749 kB (100%), 1/1 tablespace
```

- Changement du numéro de port dans le fichier de configuration personnalisée :

```
$ cat <<_EOF_ >data3/custom_conf/port.conf
> port = 5434
> _EOF_
```

- Démarrage du deuxième esclave :

```
$ pg_ctl -D data3 start
server starting
LOG:  redirecting log output to logging collector process
HINT:  Future log output will appear in directory "pg_log".
```

Testons que le deuxième esclave reçoive bien les modifications faites sur le maître :

```
$ psql -p5432 postgres
psql (9.3.1)
Type "help" for help.

postgres=# INSERT INTO t3 VALUES (1);
INSERT 0 1
postgres=# \q
$ psql -p5434 postgres
psql (9.3.1)
Type "help" for help.

postgres=# SELECT * FROM t3;
 id
----
  1
(1 row)
```

La ligne insérée dans la table **t3** du maître se trouve bien sur le deuxième esclave. Il ne reste plus qu'à arrêter le maître, promouvoir le premier esclave, faire une modification sur cet ancien esclave et voir si la modification arrive sur le dernier esclave :

```
$ pg_ctl -D data stop
waiting for server to shut down..... done
server stopped
$ pg_ctl -D data2 promote
server promoting
$ psql -p5433 postgres
psql (9.3.1)
Type "help" for help.

postgres=# INSERT INTO t3 VALUES (2);
INSERT 0 1
postgres=# \q
$ psql -p5434 postgres
psql (9.3.1)
Type "help" for help.

postgres=# SELECT * FROM t3;
 id
----
  1
  2
(2 rows)
```

Il est à noter que ce suivi de changement de ligne de temps est aussi accepté par les outils **pg_basebackup** et **pg_receivexlog**.

Dans les améliorations diverses, notons l'arrivée de deux nouvelles fonctions, **pg_is_in_backup** et **pg_backup_start_time**, permettant respectivement de savoir si une sauvegarde en ligne des fichiers est en cours et depuis quand. Notons aussi un nouveau paramètre, **wal_receiver_timeout**, permettant de configurer le délai avant réveil du processus de réception sur l'esclave des enregistrements de journaux de transactions. Suivant le délai configuré, la déconnexion du maître sera détectée plus ou moins rapidement. Enfin, notons qu'il est possible d'utiliser un outil comme **pg_receivexlog** sur un serveur ayant une autre architecture matérielle que le maître. Toutefois, le rejeu ne peut se faire que sur un serveur de même architecture (32 bits/64 bits, *little endian*/*big endian*).

1.4 Et la supervision ?

Cette nouvelle version de PostgreSQL n'ajoute pas de nouvelles statistiques d'activité pour la supervision. Par contre, elle améliore les performances et la gestion du processus de récupération des statistiques (processus appelé *stats collector*).

Le fichier **pg_stat.stat** contenant les statistiques d'activité a été divisé en plusieurs parties : un fichier global (habilement nommé **global.stat**), un fichier pour les statistiques concernant toutes les bases (appelé **db_0.stat**) et un fichier par base (dont le nom commence par le préfixe **db_**, continue avec l'OID correspondant à la base de données, et se termine avec l'extension **.stat**). À l'arrêt, ces fichiers se trouvent dans le nouveau répertoire **pg_stat**, et non dans le répertoire global comme c'était le cas auparavant. Au démarrage, les fichiers sont copiés dans le répertoire pointé par le paramètre **stats_temp_directory** (par défaut, **pg_stat_tmp**) comme dans les versions précédentes.

```
[guillaume@laptop data]$ ll pg_stat
total 16
-rw-------. 1 guillaume guillaume 2202 Dec  1 19:42 db_0.stat
-rw-------. 1 guillaume guillaume 7948 Dec  1 19:42 db_12949.stat
-rw-------. 1 guillaume guillaume  471 Dec  1 19:42 global.stat
```

Cette répartition sur plusieurs fichiers permet d'améliorer fortement les performances relatives à l'enregistrement des statistiques d'activité. En effet, la majorité des écritures sur les statistiques ne sont que des écritures partielles. Quand il n'y avait qu'un fichier, on réécrivait le fichier entier. En divisant le fichier par base, il ne reste plus qu'à écrire le fichier de la base. Comme il est plus petit, on gagne en bande passante.

L'autre amélioration est plutôt un correctif. Le collecteur de statistiques se comporte enfin correctement quand l'horloge système va en arrière (suite à une remise à l'heure du système par exemple). Dans les versions précédentes, la récupération d'informations s'arrêtait jusqu'à ce que l'horloge système revienne à la valeur précédemment enregistrée par le collecteur.

SERVEURS DÉDIÉS
PREMIÈRE MONDIALE CHEZ 1&1

TOUT NOUVEAU ET DÉJÀ CHEZ 1&1 :

INTEL® ATOM™

1&1 SERVEUR DÉDIÉ A8i

À partir de

39,99 € HT/mois
47,99 € TTC*

NOUVEAU : 1&1 SERVEUR DÉDIÉ A8i AVEC 30 % DE PERFORMANCE EN PLUS

- Intel® Atom™ C2750
- 8 Cœurs et 8 Go de RAM
- 2 x 1 To SATA HDD
- Parallels® Plesk Panel 11
- Linux, Windows ou Clé-en-main
- Bande passante 100 Mbps
- Architecture 64 bits
- System-on-Chip (SoC) : 30 % de performance supplémentaire

MEMBRE DE united internet

0970 808 911
(appel non surtaxé)

1 MOIS POUR ESSAYER

1 CHOIX VOTRE DURÉE D'ENGAGEMENT

1 APPEL UN EXPERT VOUS RÉPOND

1&1

1and1.fr

* Le serveur dédié A8i est à partir de 39,99 € HT/mois (47,99 € TTC) pour un engagement de 24 mois. Egalement disponible avec une durée d'engagement de 12 mois ou sans durée minimale d'engagement. Frais de mise en service : 49 € HT (58,80 € TTC). Conditions détaillées sur 1and1.fr. Intel, le logo Intel, Intel Atom et Intel Inside sont des marques commerciales d'Intel Corporation aux États-Unis et/ou dans d'autres pays.

1.5 Lire le contenu des journaux de transactions

Depuis quelques temps, il existe une application nommée **xlogdump**, qui permet de décoder le contenu d'un journal de transactions. Cet outil est plus ou moins bien maintenu, et en fait, surtout moins. Les développeurs de PostgreSQL ont donc travaillé pour l'intégrer dans la distribution de PostgreSQL, ce qui permettra d'assurer sa maintenance au fil des versions mineures et majeures.

Cet outil permet de décoder le contenu d'un ou plusieurs journaux de transactions. Il est possible de lui demander de filtrer par rapport à un numéro de transaction spécifique, ce qui facilite la compréhension. Par exemple, avec cette transaction :

```
postgres=# BEGIN;
BEGIN
postgres=# SELECT txid_current();
 txid_current
--------------
         2008
(1 row)

postgres=# CREATE TABLE t6(id integer);
CREATE TABLE
postgres=# INSERT INTO t6 VALUES (1);
INSERT 0 1
postgres=# INSERT INTO t6 VALUES (2);
INSERT 0 1
postgres=# CREATE INDEX ON t6(id);
CREATE INDEX
postgres=# INSERT INTO t6 VALUES (3);
INSERT 0 1
postgres=# INSERT INTO t6 VALUES (4);
INSERT 0 1
postgres=# COMMIT;
COMMIT
```

Notez l'utilisation de la fonction **txid_current()** pour connaître l'identifiant de transaction à suivre. Avant d'utiliser **pg_xlogdump**, nous devons savoir sur quel journal travaille PostgreSQL :

```
postgres=# SELECT pg_xlogfile_name(pg_
current_xlog_location());
     pg_xlogfile_name
--------------------------
 000000010000000100000007
(1 row)
```

L'outil **pg_xlogdump** va nous donner cette sortie (éditée pour qu'elle soit plus simple à suivre), en lui demandant de ne décoder que les enregistrements concernant la transaction 2008 (option **--xid**) dans le journal **000000010000000100000007** (dernière option sur la ligne de commande) :

```
$ pg_xlogdump --xid 2008 000000010000000100000007
rmgr: Storage     len (rec/tot):     16/    48, tx:       2008, lsn: 1/071681B0, prev
1/07168180, bkp: 0000, desc: file create: base/12949/16558
```

Ce **file create** correspond au **CREATE TABLE t6**. L'identifiant associé est **16558** dans la base **12949**. Les enregistrements suivants correspondent aux modifications des catalogues système :

```
rmgr: Heap        len (rec/tot):     21/  2621, tx:       2008, lsn: 1/071681E0, prev
1/071681B0, bkp: 1000, desc: insert: rel 1663/12949/12688; tid 8/14
rmgr: Btree       len (rec/tot):     18/  7754, tx:       2008, lsn: 1/07168C20, prev
1/071681E0, bkp: 1000, desc: insert: rel 1663/12949/12690; tid 1/382
rmgr: Btree       len (rec/tot):     18/  4418, tx:       2008, lsn: 1/0716AA88, prev
1/07168C20, bkp: 1000, desc: insert: rel 1663/12949/12691; tid 2/93
rmgr: Heap        len (rec/tot):     21/  5877, tx:       2008, lsn: 1/0716BBD0, prev
1/0716AA88, bkp: 1000, desc: insert: rel 1663/12949/12817; tid 46/142
rmgr: Btree       len (rec/tot):     18/  4426, tx:       2008, lsn: 1/0716D2E0, prev
1/0716BBD0, bkp: 1000, desc: insert: rel 1663/12949/12819; tid 20/107
rmgr: Btree       len (rec/tot):     18/  5994, tx:       2008, lsn: 1/0716E448, prev
1/0716D2E0, bkp: 1000, desc: insert: rel 1663/12949/12820; tid 27/94
[...]
```

L'insertion a lieu ensuite :

```
rmgr: Standby     len (rec/tot):     16/    48, tx:       2008, lsn: 1/07180040, prev
1/0717E0D8, bkp: 0000, desc: AccessExclusive locks: xid 2008 db 12949 rel 16558
rmgr: Heap        len (rec/tot):     31/    63, tx:       2008, lsn: 1/07180070, prev
1/07180040, bkp: 0000, desc: insert(init): rel 1663/12949/16558; tid 0/1
rmgr: Heap        len (rec/tot):     31/    63, tx:       2008, lsn: 1/071800B0, prev
1/07180070, bkp: 0000, desc: insert: rel 1663/12949/16558; tid 0/2
```

La première ligne correspond à l'ajout d'un verrou à accès exclusif. La deuxième ligne correspond à la première insertion. Comme il s'agit d'une toute première insertion pour cette table, il y a initialisation du fichier. Le troisième enregistrement correspond à la deuxième ligne.

Maintenant, il y a création d'un nouveau fichier et ajout de tout plein d'informations dans les catalogues système :

```
rmgr: Storage     len (rec/tot):     16/    48, tx:       2008, lsn: 1/071800F0, prev
1/071800B0, bkp: 0000, desc: file create: base/12949/16561
rmgr: Standby     len (rec/tot):     16/    48, tx:       2008, lsn: 1/07180120, prev
1/071800F0, bkp: 0000, desc: AccessExclusive locks: xid 2008 db 12949 rel 16561
rmgr: Heap        len (rec/tot):    171/   203, tx:       2008, lsn: 1/07180150, prev
1/07180120, bkp: 0000, desc: insert: rel 1663/12949/12711; tid 7/45
rmgr: Btree       len (rec/tot):     34/    66, tx:       2008, lsn: 1/07180220, prev
1/07180150, bkp: 0000, desc: insert: rel 1663/12949/12713; tid 1/339
rmgr: Btree       len (rec/tot):     42/    74, tx:       2008, lsn: 1/07180268, prev
1/07180220, bkp: 0000, desc: insert: rel 1663/12949/12714; tid 2/133
rmgr: Heap        len (rec/tot):    143/   175, tx:       2008, lsn: 1/071802B8, prev
1/07180268, bkp: 0000, desc: insert: rel 1663/12949/12700; tid 43/97
```

Cela correspond à l'ajout de l'index. Après, nous avons deux nouvelles insertions, mais elles sont légèrement différentes de la fois précédente :

```
rmgr: Heap        len (rec/tot):     31/    63, tx:       2008, lsn: 1/07187118, prev
1/07187058, bkp: 0000, desc: insert: rel 1663/12949/16558; tid 0/3
rmgr: Btree       len (rec/tot):     34/    66, tx:       2008, lsn: 1/07187158, prev
1/07187118, bkp: 0000, desc: insert: rel 1663/12949/16561; tid 1/3
```

```
rmgr: Heap        len (rec/tot):     31/    63, tx:       2008, lsn: 1/071871A0, prev
1/07187158, bkp: 0000, desc: insert: rel 1663/12949/16558; tid 0/4
rmgr: Btree       len (rec/tot):     34/    66, tx:       2008, lsn: 1/071871E0, prev
1/071871A0, bkp: 0000, desc: insert: rel 1663/12949/16561; tid 1/4
```

En effet, maintenant qu'un index est présent, chaque insertion cause deux enregistrements : un pour la table (identifiant **16558**) et un pour l'index (identifiant **16561**).

Enfin, en dernier lieu se trouve l'enregistrement du **COMMIT** :

```
rmgr: Transaction len (rec/tot):    560/   592, tx:       2008, lsn: 1/07187228, prev 1/071871E0, bkp: 0000,
desc: commit: 2014-01-26 17:52:08.706046 CET; inval msgs: catcache 45 catcache 44 catcache 45 catcache 44
catcache 45 catcache 44 catcache 7 catcache 6 catcache 32 catcache 59 catcache 58 catcache 59 catcache 58
catcache 45 catcache 44 catcache 7 catcache 6 catcache 7 catcache 6 catcache 7 catcache 6 catcache 7 catcache
6 catcache 7 catcache 6 catcache 7 catcache 6 catcache 7 catcache 6 relcache 16558 relcache 16561 relcache
16561 relcache 16558
```

Il est à noter que vous pouvez obtenir ce message en fin d'exécution de **pg_xlogdump** :

```
pg_xlogdump: FATAL:  error in WAL record at 1/7187228: record with zero length at 1/7187478
```

Ce message est une erreur indiquant que **pg_xlogdump** n'a pas pu atteindre la fin du journal de transactions, car ce dernier n'est pas complètement écrit. L'erreur n'est pas grave en soi.

2 De quoi ravir les utilisateurs

Les utilisateurs ne sont pas oubliés. Quelques fonctionnalités manquantes font leur apparition.

2.1 Ajout de la clause LATERAL

PostgreSQL essaie de respecter au mieux les différentes versions de la norme SQL. Un manque était la clause **LATERAL**. Cette dernière est une amélioration des jointures, car elle permet de faire référence aux colonnes d'une table incluse précédemment dans la clause **FROM**. Cela peut nous donner ceci :

```
SELECT pg_ls_dir, size, access
FROM pg_ls_dir('.')
JOIN LATERAL pg_stat_file(pg_ls_dir) ON pg_ls_dir LIKE '%stat%';
```

Il est possible de ne pas utiliser la clause **JOIN** s'il n'est pas nécessaire de filtrer le résultat, mais on peut aussi transformer le **JOIN** en **LEFT JOIN**.

2.2 Vues modifiables... automatiquement

Exécuter des ordres DML (**INSERT**, **UPDATE**, **DELETE**) sur une vue est possible depuis bien longtemps. Cependant, ce n'est pas un comportement natif des vues. Il faut ajouter des règles ou des *triggers* (depuis la version 9.1 pour ces derniers) pour qu'un ordre DML soit exécutable sur une vue.

PostgreSQL 9.3 améliore grandement la situation, en rendant automatiquement modifiable les vues simples. Voici un exemple simple :

```
postgres=# CREATE TABLE t5 (c1 integer, c2 text);
CREATE TABLE
postgres=# INSERT INTO t5 SELECT i, 'Ligne '||i FROM generate_series(1, 10) AS i;
INSERT 0 10
postgres=# CREATE VIEW v5 AS SELECT * FROM t5 WHERE c1<5;
CREATE VIEW
postgres=# SELECT * FROM v5;
 c1 |   c2
----+--------
  1 | Ligne 1
  2 | Ligne 2
  3 | Ligne 3
  4 | Ligne 4
(4 rows)

postgres=# UPDATE v5 SET c2=upper(c2) WHERE c1=3;
UPDATE 1
postgres=# INSERT INTO v5 VALUES (0);
INSERT 0 1
postgres=# SELECT * FROM v5;
 c1 |   c2
----+--------
  1 | Ligne 1
  2 | Ligne 2
  4 | Ligne 4
  3 | LIGNE 3
  0 |
(5 rows)
```

La mise à jour s'est bien passée, ainsi que l'insertion. Ce fonctionnement n'est possible que pour les vues simples. Une vue contenant une colonne calculée ou une jointure ne sera pas modifiable automatiquement.

```
postgres=# CREATE VIEW v6 AS SELECT c1, c1+3, c2 FROM
t5 WHERE c1<5;
CREATE VIEW
postgres=# UPDATE v6 SET c2=upper(c2) WHERE c1=1;
ERROR:  cannot update view "v6"
DETAIL:  Views that return columns that are not columns
of their base relation are not automatically updatable.
HINT:  To enable updating the view, provide an INSTEAD
OF UPDATE trigger or an unconditional ON UPDATE DO
INSTEAD rule.
```

Dans ce cas, comme l'indique la ligne **HINT**, il faut toujours passer par une règle ou un trigger.

Ce n'est pas le seul inconvénient de cette nouvelle fonctionnalité. Voyons un nouvel exemple :

```
postgres=# INSERT INTO v5 VALUES (10);
INSERT 0 1
postgres=# UPDATE v5 SET c1=11 WHERE c1=1;
UPDATE 1
postgres=# SELECT * FROM v5;
 c1 |   c2
----+--------
  2 | Ligne 2
  4 | Ligne 4
  3 | LIGNE 3
  0 |
(4 rows)
```

L'insertion s'est apparemment bien passée. Par contre, elle n'est pas visible dans la vue. C'est normal, vu que la donnée insérée ne remplit pas la condition de la clause **WHERE**, mais c'est un peu étonnant. Encore plus étonnant, l'**UPDATE** s'est comporté comme un **DELETE**. En effet, la nouvelle ligne ne remplit plus la condition de la clause **WHERE**, elle n'est donc plus visible après l'**UPDATE**.

Ce comportement est logique, mais vraiment contre-intuitif. Il manque en fait que PostgreSQL respecte la clause **WITH CHECK OPTION** des vues.

2.3 Ajout des vues matérialisées

PostgreSQL dispose de vues depuis très longtemps. Ces vues sont des requêtes SQL pré-enregistrées. Lorsqu'elles sont utilisées dans une requête, le planificateur remplace la vue par la requête et optimise l'ensemble pour une exécution globale. L'avantage est que l'exécution est optimisée avec la requête englobante, l'inconvénient est que la requête de la vue doit être exécutée. Parfois, on aimerait bien ne pas avoir besoin d'exécuter la requête de la vue, et de simplement lire des données pré-calculées. Ce sont des vues matérialisées. Ce type de vue n'existait pas dans PostgreSQL avant la 9.3. Il était bien possible de les simuler avec une table qui contenait les données, la table étant alimentée par des triggers sur la table ou les tables de la requête de la vue. Cela pouvait se révéler bien difficile à réaliser, et quelques erreurs pouvaient toujours se glisser dans ce système.

PostgreSQL 9.3 dispose donc du concept de vue matérialisée. C'est un premier jet, donc toutes les possibilités offertes par d'autres moteurs ne sont pas encore présentes. Mais c'est un début intéressant.

```
postgres=# \timing
Timing is on.
postgres=# CREATE VIEW v1std AS SELECT * FROM t1 WHERE id<10000;
CREATE VIEW
Time: 54.281 ms
postgres=# CREATE MATERIALIZED VIEW v1mat AS SELECT * FROM t1 WHERE id<10000;
SELECT 39996
Time: 4199.586 ms
```

Dans le cas d'une vue normale, seules quelques informations système sont enregistrées, ce qui rend l'ajout d'une vue très rapide. Dans le cas d'une vue matérialisée, quelques informations système sont aussi enregistrées, mais la requête est exécutée et son résultat est stocké dans un fichier sur disque. D'où le fait que la création d'une vue matérialisée est bien plus longue que celle d'une vue standard.

Cherchons maintenant à récupérer les données de chaque vue (le **\o /dev/null** permet d'éviter l'affichage du résultat, qui ne nous intéresse pas ici) :

```
postgres=# \o /dev/null
postgres=# SELECT * FROM v1std;
Time: 4459.520 ms
postgres=# SELECT * FROM v1mat;
Time: 42.659 ms
```

Le résultat est inversé. La lecture de la vue normale est lente, car la requête de la vue doit être exécutée. Dans le cas de la lecture de la vue matérialisée, le résultat est déjà disponible dans un fichier qu'il suffit de lire. Les plans d'exécution le montrent clairement :

```
postgres=# EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM v1std;
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on t1  (cost=0.00..676992.00 rows=35758 width=4)
                 (actual time=0.044..4294.637 rows=39996 loops=1)
   Filter: (id < 10000)
   Rows Removed by Filter: 39960004
   Buffers: shared read=176992
 Total runtime: 4297.286 ms
(5 rows)

postgres=# EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM v1mat;
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on v1mat  (cost=0.00..576.96 rows=39996 width=4)
                    (actual time=0.048..9.882 rows=39996 loops=1)
   Buffers: shared read=177
 Total runtime: 14.019 ms
(3 rows)
```

L'intérêt d'une vue matérialisée est donc de stocker le résultat d'une requête SQL et de pouvoir accéder à ce résultat rapidement. Cependant, que se passe-t-il quand les données changent sur les tables sources :

```
postgres=# CREATE TABLE t4 (c1 integer, c2 text);
postgres=# INSERT INTO t4 SELECT i, 'Ligne '||i FROM generate_series(1, 10) AS i;
postgres=# CREATE VIEW t4std AS SELECT * FROM t4 WHERE c1<5;
postgres=# CREATE MATERIALIZED VIEW t4mat AS SELECT * FROM t4 WHERE c1<5;
postgres=# \o
postgres=# SELECT * FROM t4std WHERE c1<1;
 c1 | c2
----+----
(0 rows)

Time: 0.518 ms
postgres=# SELECT * FROM t4mat WHERE c1<1;
 c1 | c2
----+----
(0 rows)

Time: 0.476 ms
```

Les vues n'affichent aucune donnée. Insérons une ligne qui devrait satisfaire la clause **WHERE** :

```
postgres=# INSERT INTO t4 VALUES (0);
INSERT 0 1
postgres=# SELECT * FROM t4std WHERE c1<1;
 c1 | c2
----+----
  0 |
(1 row)

Time: 0.555 ms
postgres=# SELECT * FROM t4mat WHERE c1<1;
 c1 | c2
----+----
(0 rows)

Time: 0.502 ms
```

La nouvelle ligne apparaît dans la vue normale, mais pas dans la vue matérialisée. En effet, comme la requête est à chaque fois exécutée pour la première vue, elle récupère les modifications des tables sous-jacentes. Quant à la vue matérialisée, elle ne fait que ressortir son ancien résultat, elle n'est pas rafraîchie. Pour la rafraîchir, il faut utiliser une nouvelle instruction :

```
postgres=# REFRESH MATERIALIZED VIEW t4mat;
REFRESH MATERIALIZED VIEW
Time: 98.361 ms
postgres=# SELECT * FROM t4mat WHERE c1<1;
 c1 | c2
----+----
  Ø |
(1 row)

Time: Ø.756 ms
```

Et voilà !

Pour en revenir au thème des performances, un moyen d'accélérer encore l'accès à une vue matérialisée est de lui ajouter un index :

```
postgres=# CREATE INDEX ON t4mat(c2);
CREATE INDEX
postgres=# INSERT INTO t4 SELECT i, 'Ligne '||i FROM generate_series(-1000000,
Ø) AS i;
INSERT Ø 1000001
Time: 2998.253 ms
postgres=# REFRESH MATERIALIZED VIEW t4mat;
REFRESH MATERIALIZED VIEW
Time: 20732.083 ms
postgres=# EXPLAIN SELECT * FROM t4mat WHERE c2='Ligne -4';
                            QUERY PLAN
-------------------------------------------------------------------
 Index Scan using t4mat_c2_idx on t4mat  (cost=Ø.42..8.44 rows=1 width=12)
   Index Cond: (c2 = 'Ligne -4'::text)
(2 rows)
```

L'index est bien utilisé.

Enfin, notez aussi qu'une vue matérialisée n'est pas modifiable automatiquement :

```
postgres=# INSERT INTO t4mat VALUES (-1);
ERROR:  cannot change materialized view "t4mat"
Time: Ø.334 ms
```

2.4 Autres améliorations concernant la norme SQL

Comme à chaque nouvelle version, de nouvelles clauses ou de nouvelles instructions apparaissent, soit pour compléter le support de la norme, soit pour assurer une meilleure compatibilité des instructions SQL entre moteurs de bases de données. La syntaxe **CREATE RECURSIVE VIEW** appartient clairement à la première raison, car si cette syntaxe n'existait pas, sa fonctionnalité était déjà offerte avec un **CREATE VIEW** sur une requête CTE récursive.

Ça peut aussi être dans le but de faciliter la vue des administrateurs de bases de données. Dans ce cadre, l'instruction **CREATE SCHEMA** se voit ajouter la clause **IF NOT EXISTS**. Il est ainsi plus facile d'écrire un script, notamment de mise à jour de structure, sans avoir d'erreur due à un schéma pré-existant.

```
base_locale=# CREATE SCHEMA public;
ERROR:  schema "public" already exists
base_locale=# CREATE SCHEMA IF NOT EXISTS public;
NOTICE:  schema "public" already exists, skipping
CREATE SCHEMA
```

Un message apparaît toujours, pour informer du fait que le schéma existe bien déjà et que la requête SQL est donc ignorée. Cela permet d'éviter d'annuler une transaction pour une erreur qui, suivant le contexte, n'est pas forcément grave au point de devoir annuler la transaction courante.

Enfin, ça peut aussi être pour ajouter des fonctionnalités. L'instruction **REASSIGN OWNED** réaffectait le propriétaire d'objets pour les objets d'une certaine base de données. Maintenant, elle est aussi capable de réaffecter les objets partagés (donc bases de données et *tablespaces*). En voici un exemple :

```
base_locale=# CREATE TABLESPACE grosdisque OWNER u1 LOCATION '/home/
guillaume/ici';
CREATE TABLESPACE
base_locale=# \db
            List of tablespaces
    Name    |   Owner   |      Location
------------+-----------+--------------------
 grosdisque | u1        | /home/guillaume/ici
 pg_default | guillaume |
 pg_global  | guillaume |
(3 rows)

base_locale=# REASSIGN OWNED BY u1 TO guillaume;
REASSIGN OWNED
base_locale=# \db
            List of tablespaces
    Name    |   Owner   |      Location
------------+-----------+--------------------
 grosdisque | guillaume | /home/guillaume/ici
 pg_default | guillaume |
 pg_global  | guillaume |
(3 rows)
```

Enfin, il est bon de noter qu'une instruction **CREATE TABLE**, qui ajoute implicitement des objets (index et/ou séquence), n'envoie plus de messages sur ces créations. Il est nécessaire de passer au niveau **DEBUG1** pour les récupérer.

2.5 Nouveautés sur la norme SQL/MED

SQL/MED est la norme SQL permettant l'accès à des données distantes, à partir de l'interface SQL de PostgreSQL. Son implémentation a commencé avec la version 8.4, où sont apparus les *Foreign Data Wrappers* (sorte de connecteurs vers d'autres sources de données), les *Foreign Servers* (pour déclarer

R & D, Enseignants, voici une offre qui vous est spécialement destinée !

NOUVEAUTÉ 2014 !

L'accès à la base documentaire en ligne totale des Éditions Diamond vous permettra d'effectuer des recherches dans la majorité des articles parus dans nos magazines.

Vous pourrez ainsi trouver l'article indexé qu'il vous faut, puis, par exemple copier/coller les codes etc.

Ces articles seront disponibles avec un décalage de 6 mois après leur parution dans l'ensemble de nos titres.

UNE SOLUTION !

Il y a certainement la réponse dans

LA BASE DOCUMENTAIRE !

...ET ACCÉDEZ À + DE 3500 ARTICLES DE TOUS NOS MAGAZINES !

diamond.com

DES OFFRES BASE DOCUMENTAIRE PAR MAGAZINE SONT DISPONIBLES SUR : **boutique.ed-diamond.com**

le moyen d'atteindre une source de données) et les *User Mappings* (pour préciser la correspondance entre utilisateur local et utilisateur distant). La 9.1 allait plus loin en offrant le concept de *Foreign Table* (une table locale ne contenant qu'un pointeur vers les données externes). L'implémentation était déjà très poussée, au niveau des fonctionnalités comme au niveau des performances. Cependant, il manquait quand même une fonctionnalité de base : la possibilité d'écrire sur les Foreign Tables. La version 9.3 comble ce manque. Il faudra évidemment voir une mise à jour des différents Foreign Data Wrappers pour pouvoir en profiter. Néanmoins, plusieurs Foreign Data Wrappers le proposent déjà : **oracle_fdw** et **postgres_fdw**..., car en effet, la version 9.3 apporte aussi dans les modules **contrib** un Foreign Data Wrapper pour PostgreSQL. Il était un peu temps, son arrivée s'est laissée désirer. Voici un exemple d'utilisation de ce Foreign Data Wrapper, avec un exemple d'écriture.

Tout d'abord, il faut ajouter le Foreign Data Wrapper PostgreSQL sur la base locale (qu'on créera au préalable) :

```
$ psql postgres
psql (9.3.2)
Type "help" for help.

postgres=# CREATE DATABASE base_locale;
CREATE DATABASE
postgres=# \c base_locale
You are now connected to database "base_locale" as user "guillaume".
base_locale=# CREATE EXTENSION postgres_fdw;
CREATE EXTENSION
base_locale=# CREATE SERVER serveur_distant FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (DBNAME 'postgres');
CREATE SERVER
base_locale=# CREATE USER MAPPING FOR guillaume SERVER serveur_distant OPTIONS
(user 'guillaume');
CREATE USER MAPPING
base_locale=# CREATE FOREIGN TABLE t4 (c1 integer, c2 text) SERVER serveur_
distant;
CREATE FOREIGN TABLE
base_locale=# SELECT * FROM t4 LIMIT 10;
 c1 |    c2
----+---------
  1 | Ligne 1
  2 | Ligne 2
  3 | Ligne 3
  4 | Ligne 4
  5 | Ligne 5
  6 | Ligne 6
  7 | Ligne 7
  8 | Ligne 8
  9 | Ligne 9
 10 | Ligne 10
(10 rows)
```

On arrive bien à récupérer les données de la table distante. On va maintenant récupérer un élément :

```
base_locale=# EXPLAIN (ANALYZE, BUFFERS, VERBOSE) SELECT * FROM t4 WHERE c1=1;
                              QUERY PLAN
-----------------------------------------------------------------------
 Foreign Scan on public.t4  (cost=100.00..127.20 rows=7 width=36)
                           (actual time=128.053..128.055 rows=1 loops=1)
   Output: c1, c2
   Remote SQL: SELECT c1, c2 FROM public.t4 WHERE ((c1 = 1))
 Total runtime: 128.320 ms
(4 rows)
```

La requête envoyée au serveur distant contient le filtre. Autrement dit, on ne récupère pas toutes les données, seulement celles qui ont été filtrées. Cela va même plus loin :

```
base_locale=# EXPLAIN (ANALYZE, BUFFERS, VERBOSE) SELECT c2 FROM t4 where c1=1;
                              QUERY PLAN
-----------------------------------------------------------------------
 Foreign Scan on public.t4  (cost=100.00..128.41 rows=7 width=32)
                           (actual time=127.614..127.615 rows=1 loops=1)
   Output: c2
   Remote SQL: SELECT c2 FROM public.t4 WHERE ((c1 = 1))
 Total runtime: 127.884 ms
(4 rows)
```

Le Foreign Data Wrapper a su qu'on ne voulait que la colonne **c2** et n'a donc réclamé que cette colonne. Testons maintenant la modification d'une ligne :

```
base_locale=# UPDATE t4 SET c2=upper(c2) WHERE c1=2;
UPDATE 1
base_locale=# SELECT * FROM t4 WHERE c1 BETWEEN 0 AND 5;
 c1 |   c2
----+---------
  1 | Ligne 1
  3 | Ligne 3
  4 | Ligne 4
  5 | Ligne 5
  0 |
  2 | LIGNE 2
  0 | Ligne 0
(7 rows)
```

La valeur a bien été mise à jour. Maintenant, testons une insertion :

```
base_locale=# INSERT INTO t4 VALUES (3, 'trois');
INSERT 0 1
base_locale=# SELECT * FROM t4 WHERE c1=3;
 c1 |   c2
----+---------
  3 | Ligne 3
  3 | trois
(2 rows)
```

La ligne a bien été insérée.

Les modifications sur table distante fonctionnent bien. Elles sont performantes. L'implémentation de SQL/MED a franchi une nouvelle étape significative. Évidemment, il ne s'agit que de l'implémentation au niveau de PostgreSQL. Il faudra voir maintenant si les différents Foreign Data Wrappers vont saisir l'opportunité et proposer ce type de fonctionnalités. Néanmoins, c'est déjà le cas pour le FDW sur Oracle et sur PostgreSQL.

2.6 Détecter la présence d'un serveur

Un nouvel outil fait son apparition en 9.3. Il s'agit de **pg_isready**. Le but de cet outil est de pouvoir dire si un serveur est démarré ou non, sans faire de connexion réellement. Cet outil se base sur la fonction **PQping** de la bibliothèque C libpq. Cette fonction existe depuis la version 9.2, mais aucun outil n'en profitait encore. C'est chose faite avec cette version 9.3. Voici un exemple, sur deux ports différents :

```
$ pg_isready -h localhost -p 5432
localhost:5432 - accepting connections
$ pg_isready -h localhost -p 5433
localhost:5433 - no response
```

Cet outil ne fait pas grand-chose directement, mais il permet de tester la disponibilité d'un serveur. Il est intéressant dans le cadre d'un script qui doit exécuter des requêtes mais veut vérifier avant que le serveur est bien disponible.

2.7 Sauvegarde parallélisée

Depuis la version 8.4, PostgreSQL dispose d'un outil de restauration parallélisée. Il fonctionne en ouvrant plusieurs connexions à la base de données, et en restaurant chaque table et chaque index sur une des connexions ouvertes. Cela permet de contourner le manque de parallélisation du moteur. Mais, autant cette technique est efficace avec la restauration (car il n'est pas nécessaire de voir la même image de la base de données quand on ne fait qu'écrire), autant elle n'est pas simple à implémenter pour la sauvegarde. En effet, il faudrait que chaque connexion puisse partager la même vision de la base de données pour que la sauvegarde soit cohérente. Il est aussi difficile d'écrire dans le même fichier à partir de plusieurs processus, il faudrait donc un format de sauvegarde multi-fichier.

Pour aboutir à la sauvegarde parallélisée, les développeurs de PostgreSQL se sont donc attaqués à ces différents problèmes. Ils ont tout d'abord proposé une fonction d'export du *snapshot* de la base de données. Cette fonction est disponible depuis la version 9.2, sous le nom de **pg_export_snapshot()**. Le texte en sortie est à fournir à l'instruction **SET TRANSACTION SNAPSHOT** sur une autre session pour que cette session voie la base de données de la même façon que la session qui a exécuté la fonction.

Les développeurs de PostgreSQL ont ensuite travaillé sur un autre format de sauvegarde, appelé *directory*. Ce format enregistre les données des tables et des vues matérialisées dans des fichiers avec extension **.dat** dans le répertoire indiqué avec l'option **-f**.

L'infrastructure étant disponible, ils ont pu travailler sur la parallélisation de **pg_dump** pour la version 9.3. On active ce mode en utilisant l'option **-j**, à qui on indique le nombre de connexions à réaliser.

Pour surveiller ce que fait **pg_dump**, nous allons tracer ses actions grâce à une modification de la configuration :

- **log_min_duration_statement = 0**
- **log_connections = on**
- **log_disconnections = on**
- **log_line_prefix = '%t [%p,%l] '**

Ainsi, après rechargement de la configuration, PostgreSQL va tracer toute connexion et déconnexion, ainsi que toutes les requêtes exécutées.

Voici la commande de sauvegarde parallélisée :

```
$ pg_dump -Fd -f sauvegarde -j 2 postgres
```

L'option **-Fd** permet de sélectionner le format directory, l'option **-f sauvegarde** indique le répertoire **sauvegarde** à **pg_dump** (il n'a pas besoin d'exister, **pg_dump** le crée si nécessaire), l'option **-j 2** indique le nombre de connexions (donc le niveau de parallélisation) et enfin, **postgres** est la base à sauvegarder.

Après exécution, le répertoire **sauvegarde** contient bien les données de la sauvegarde :

```
$ ll sauvegarde
total 87268
-rw-rw-r--. 1 guillaume guillaume 84408879 Jan 26 19:58 2907.dat.gz
-rw-rw-r--. 1 guillaume guillaume       25 Jan 26 19:57 2908.dat.gz
-rw-rw-r--. 1 guillaume guillaume       27 Jan 26 19:57 2909.dat.gz
-rw-rw-r--. 1 guillaume guillaume  4921879 Jan 26 19:57 2911.dat.gz
-rw-rw-r--. 1 guillaume guillaume       91 Jan 26 19:57 2913.dat.gz
-rw-rw-r--. 1 guillaume guillaume       33 Jan 26 19:57 2914.dat.gz
-rw-rw-r--. 1 guillaume guillaume     6427 Jan 26 19:57 toc.dat
```

En regardant les traces, on aperçoit quelques informations intéressantes. Le premier processus a ouvert une transaction en lecture seule en mode *repeatable read*, puis il a utilisé la fonction **pg_export_snapshot()** :

```
2014-01-26 19:57:46 CET [11869,9] LOG:  duration: 0.027 ms  statement: BEGIN
2014-01-26 19:57:46 CET [11869,10] LOG:  duration: 0.038 ms  statement: SET TRANSACTION ISOLATION LEVEL
REPEATABLE READ, READ ONLY
2014-01-26 19:57:46 CET [11869,11] LOG:  duration: 1.045 ms  statement: SELECT pg_export_snapshot()
```

Après cela, il récupère un grand nombre d'informations sur le catalogue système pour recréer les ordres SQL qui permettront de recréer le schéma. Il en profite aussi pour verrouiller les tables :

```
2014-01-26 19:57:46 CET [11869,17] LOG:  duration: 0.133 ms  statement: LOCK TABLE public.t1 IN ACCESS SHARE MODE
2014-01-26 19:57:46 CET [11869,18] LOG:  duration: 0.067 ms  statement: LOCK TABLE public.t2 IN ACCESS SHARE MODE
2014-01-26 19:57:46 CET [11869,19] LOG:  duration: 0.068 ms  statement: LOCK TABLE public.t3 IN ACCESS SHARE MODE
2014-01-26 19:57:46 CET [11869,20] LOG:  duration: 0.069 ms  statement: LOCK TABLE public.t4 IN ACCESS SHARE MODE
2014-01-26 19:57:46 CET [11869,21] LOG:  duration: 0.070 ms  statement: LOCK TABLE public.t5 IN ACCESS SHARE MODE
2014-01-26 19:57:46 CET [11869,22] LOG:  duration: 0.078 ms  statement: LOCK TABLE public.t6 IN ACCESS SHARE MODE
```

Le verrou est léger, il interdit juste la suppression des tables, ainsi que la modification de la plupart de leurs propriétés (impossible d'ajouter une colonne par exemple). Cependant, la lecture et l'écriture de données sont toujours autorisées.

On voit après l'arrivée de deux connexions :

```
2014-01-26 19:57:46 CET [11872,1] LOG:  connection received: host=[local]
2014-01-26 19:57:46 CET [11872,2] LOG:  connection authorized: user=guillaume database=postgres
2014-01-26 19:57:46 CET [11873,1] LOG:  connection received: host=[local]
2014-01-26 19:57:46 CET [11873,2] LOG:  connection authorized: user=guillaume database=postgres
```

Ces connexions entrent en transaction et récupèrent l'image de la base du processus père :

```
2014-01-26 19:57:46 CET [11872,10] LOG:  duration: 0.017 ms  statement: BEGIN
2014-01-26 19:57:46 CET [11872,11] LOG:  duration: 0.026 ms  statement: SET TRANSACTION ISOLATION LEVEL REPEATABLE
READ, READ ONLY
2014-01-26 19:57:46 CET [11872,12] LOG:  duration: 0.073 ms  statement: SET TRANSACTION SNAPSHOT '00000D9-1'
```

Les tables sont ensuite sauvegardées, soit par un processus, soit par l'autre.

Cette parallélisation est très fonctionnelle, et elle repose sur une infrastructure mise en place depuis les dernières versions. Évidemment, elle n'est pas parfaite. Notamment, si vous avez une table qui prend une majeure partie de la base, cette parallélisation aura moins d'effet. Néanmoins, c'est une avancée importante pour PostgreSQL.

2.8 Quelques améliorations diverses des clients

L'outil de sauvegarde **pg_dump** dispose depuis longtemps de la possibilité de spécifier plusieurs tables avec l'option **--table**. Cette possibilité est maintenant offerte pour les outils **pg_restore**, **clusterdb**, **reindexdb** et **vacuumdb**.

De même, les outils **pg_dumpall**, **pg_basebackup** et **pg_receivexlog** ont maintenant l'option **--dbname**, qui permet de spécifier une chaîne de connexion. Par exemple :

```
$ pg_basebackup --dbname "host=master_server user=guillaume"
```

Et enfin, une modification d'importance a eu lieu sur l'option **--single-transaction** de l'outil **psql**. Cette option permet d'ajouter automatiquement une instruction **BEGIN** en début d'exécution d'un script SQL et un **END** en fin. Mais ceci ne fonctionnait qu'avec un script SQL contenu dans un fichier (donc avec l'option **-f**). Cela ne fonctionnait pas avec un script SQL envoyé sur l'entrée standard de **psql**. Ce manque est comblé.

2.9 Plus de performances

Dans le domaine des performances, l'amélioration majeure se base sur les recherches par expressions rationnelles qui peuvent enfin passer par les index. Pour cela, il faut passer par l'extension **pg_trgm** et ses capacités d'indexation avec les méthodes GIN et GiST :

```
base_locale=# CREATE EXTENSION pg_trgm;
CREATE EXTENSION
```

Cette extension ajoute de nombreux objets (visibles avec la méta-commande **\dx+ pg_trgm**), dont la classe d'opérateur **gist_trgm_ops**. Cette classe permet de créer l'index qui sera utilisable dans une recherche par expression rationnelle. Commençons par créer une table avec quelques données :

```
base_locale=# SELECT name, short_desc INTO
settings FROM pg_settings;
SELECT 227
```

Il suffit d'exécuter l'instruction **INSERT INTO settings SELECT name, short_desc FROM pg_settings;** plusieurs fois pour avoir un volume de données intéressant.

Nous pouvons enfin créer l'index :

```
base_locale=# CREATE INDEX ON settings USING
GIST (short_desc gist_trgm_ops);
CREATE INDEX
```

Et mettre à jour les statistiques du planificateur :

```
base_locale=# ANALYZE settings;
ANALYZE
```

Voici le plan d'exécution d'une recherche par expression rationnelle, une fois l'index en place et les statistiques à jour :

```
base_locale=# EXPLAIN SELECT name, short_desc FROM
settings WHERE short_desc ~ '(keepalive|restransmits)';
                                  QUERY PLAN
------------------------------------------------------------
--------------------
 Bitmap Heap Scan on settings  (cost=4.32..29.42 rows=23
width=72)
   Recheck Cond: (short_desc ~ '(keepalive|restransmits
)'::text)
   ->  Bitmap Index Scan on settings_short_desc_idx
       (cost=0.00..4.32 rows=23 width=0)
         Index Cond: (short_desc ~ '(keepalive|restrans
mits)'::text)
(4 rows)
```

La recherche passe bien par l'index.

Dans les autres améliorations, on peut noter que l'indexation SP-GiST est enfin utilisable pour les recherches sur des types de données intervalles (comme **int4range** et **daterange**). De plus, les

index GiST sont utilisables sur des tables non journalisées. Dans ce cas, ils sont non journalisés eux aussi.

Une autre amélioration notable apportée par la 9.3 concerne la commande **COPY**. Cette commande permet d'insérer un grand nombre de lignes d'un seul coup. Cependant, les *hints bits* ne sont pas immédiatement stockés dans la table, pour gagner en performance d'insertion. Le problème est que cette surcharge est payée après coup, généralement par le premier **SELECT** qui intervient après. Pour le montrer, commençons par créer un fichier de données que nous utiliserons pour peupler la table :

```
postgres=# COPY (select i, 'Ligne '||i from
generate_series(1, 10000000) as i) to '/tmp/datas';
COPY 10000000
```

Ensuite, configurons **psql** pour qu'il n'affiche pas le résultat du **SELECT** (seule sa durée nous importe) et pour qu'il affiche la durée d'exécution des requêtes suivantes :

```
postgres=# \o /dev/null
postgres=# \timing
Timing is on.
```

Ce qui suit est un exemple du problème habituel du **COPY** :

```
postgres=# CREATE TABLE t11(c1 integer, c2 text);
Time: 86.017 ms
postgres=# COPY t11 FROM '/tmp/datas';
Time: 18041.963 ms
postgres=# SELECT * FROM t11;
Time: 28682.092 ms
postgres=# SELECT * FROM t11;
Time: 3766.665 ms
postgres=# SELECT * FROM t11;
Time: 3625.188 ms
```

Comme nous pouvons le voir, le **COPY** est plutôt rapide, mais le premier **SELECT** est extrêmement lent, notamment par rapport au suivant. Ceci n'est pas dû au cache, vu que les données y sont, tout du moins partiellement, suite au **COPY**. Essayons maintenant la nouvelle option **FREEZE** de la commande **COPY** :

```
postgres=# DROP TABLE t11;
Time: 102.232 ms
postgres=# BEGIN;
Time: 0.228 ms
postgres=# CREATE TABLE t11(c1 integer, c2 text);
Time: 74.919 ms
postgres=# COPY t11 FROM '/tmp/datas' WITH
(FREEZE);
Time: 18422.317 ms
postgres=# SELECT * FROM t11;
Time: 3628.545 ms
postgres=# SELECT * FROM t11;
Time: 3630.089 ms
postgres=# SELECT * FROM t11;
Time: 3615.051 ms
postgres=# ROLLBACK;
Time: 86.377 ms
```

Le **COPY** est un peu plus lent (de deux secondes), par contre le premier **SELECT** est beaucoup plus rapide que celui du premier test. En fait, il est exécuté aussi rapidement que les **SELECT** suivants. Ceci est dû au fait que l'option **FREEZE** a forcé la commande **COPY** à faire ce que le premier **SELECT** aurait fait. Comme **COPY** écrit de toute façon toutes les lignes, elle met en plus à jour les informations des hint bits, ce qui ralentit assez peu le **COPY**. Cette amélioration va intéresser principalement ceux qui font des batchs d'insertion, et qui déclenchent immédiatement après des requêtes de *reporting*.

3 De quoi inspirer les développeurs

3.1 Améliorations sur PL/pgsql

Le langage PL/pgsql a été un peu amélioré pour la version 9.3, sans que ce soit non plus révolutionnaire.

Lorsqu'une requête tente de violer une contrainte dans une procédure stockée, la procédure stockée peut récupérer un grand nombre d'informations sur cette erreur, ce qui permet à une exception de mieux gérer l'erreur. Commençons par créer une table contenant une contrainte (dans ce cas, il s'agit de la clé primaire) :

```
postgres=# CREATE TABLE t10(id integer primary key);
CREATE TABLE
```

Ensuite, ajoutons la procédure stockée :

```
CREATE OR REPLACE FUNCTION public.f1(p_id integer)
 RETURNS boolean
 LANGUAGE plpgsql
AS $function$
declare
  message text;
  detail text;
  astuce text;
BEGIN
  INSERT INTO t10 VALUES(p_id);
  RETURN true;

  EXCEPTION WHEN OTHERS WHEN
    GET STACKED DIAGNOSTICS message = MESSAGE_TEXT,
                            detail = PG_EXCEPTION_DETAIL,
                            astuce = PG_EXCEPTION_HINT;
    RAISE LOG 'exception f1, message: %', message;
    RAISE LOG 'exception f1, detail:  %', detail;
    RAISE LOG 'exception f1, astuce:  %', astuce;
    RETURN false;
END
$function$
```

Cette procédure stockée insère une ligne dans la nouvelle table. Le bloc d'exception permet de récupérer toute erreur intervenant lors de l'insertion. Il ne fera qu'une trace détaillée de l'erreur dans les traces, et renverra **false** pour indiquer que l'insertion n'a pas eu lieu.

Si on insère une ligne correcte, tout se passe bien :

```
postgres=# SELECT f1(1);
 f1
----
 t
(1 row)
```

Si on essaie d'insérer la même ligne, on récupère bien la valeur **false** et la requête ne génère pas d'erreurs :

```
postgres=# SELECT f1(1);
 f1
----
 f
(1 row)
```

Par contre, les traces donnent les détails sur l'insertion :

```
$ grep "LOG:  exception" postgresql-2014-01-31_115314.log
2014-01-31 12:01:49 CET [5469,53] LOG:  exception f1,
message: duplicate key value violates unique constraint
"t10_pkey"
2014-01-31 12:01:49 CET [5469,55] LOG:  exception f1,
detail:  Key (id)=(1) already exists.
2014-01-31 12:01:49 CET [5469,57] LOG:  exception f1,
astuce:
```

Une autre amélioration du langage est la possibilité de récupérer le nombre de lignes insérées par une requête **COPY**.

Mais la grosse amélioration concerne sa meilleure gestion de la modification du paramètre **search_path**. Ce paramètre indique à PostgreSQL dans quels schémas rechercher les objets dont on n'a pas spécifié explicitement le schéma associé. Prenons l'exemple suivant :

```
postgres=# CREATE SCHEMA s1;
CREATE SCHEMA
postgres=# CREATE SCHEMA s2;
CREATE SCHEMA
postgres=# CREATE TABLE s1.t1 (id integer);
CREATE TABLE
postgres=# CREATE TABLE s2.t1 (id integer);
CREATE TABLE
postgres=# INSERT INTO s1.t1 VALUES (1);
INSERT 0 1
postgres=# INSERT INTO s2.t1 VALUES (2);
INSERT 0 1
```

Donc, nous avons deux tables de même nom, mais dans des schémas différents. Une requête qui explicite le schéma ne pose pas de soucis :

```
postgres=# SELECT * FROM s2.t1;
 id
----
  2
(1 row)
```

Par contre, si le schéma n'est pas indiqué, tout dépendra de la valeur du paramètre **search_path** :

```
postgres=# SELECT * FROM t1 LIMIT 5;
   id
---------
 5376993
 5376994
 5376995
 5376996
 5376997
(5 rows)
```

Comme nous n'avons pas modifié le paramètre **search_path**, PostgreSQL utilise le schéma public et il s'avère qu'une table **t1** existe dans ce schéma. Si on change la valeur du paramètre **search_path**, le résultat ne sera pas identique :

```
postgres=# SET search_path TO s1;
SET
postgres=# SELECT * FROM t1 LIMIT 5;
 id
----
  1
(1 row)
```

C'est une astuce assez intéressante dans quelques cas d'utilisation. Mais cela se révèle un problème quand un plan d'exécution est mis en cache, car le plan ne contient pas le nom de l'objet mais sa référence (son OID). Prenons un exemple de procédure stockée en 9.2 :

```
$ psql postgres
psql (9.2.6)
Type "help" for help.

postgres=# SELECT version();
                                                 version
-----------------------------------------------------------------------------------------------------
 PostgreSQL 9.2.6 on x86_64-unknown-linux-gnu, compiled by gcc (GCC) 4.8.2 20131212 (Red Hat 4.8.2-7), 64-bit
(1 row)

postgres=# CREATE SCHEMA s1 CREATE TABLE t1(id integer);
CREATE SCHEMA
postgres=# CREATE SCHEMA s2 CREATE TABLE t1(id integer);
CREATE SCHEMA
postgres=# INSERT INTO s1.t1 VALUES (1);
INSERT 0 1
postgres=# INSERT INTO s2.t1 VALUES (2);
INSERT 0 1
postgres=# CREATE FUNCTION f3(p_id integer) RETURNS integer LANGUAGE plpgsql
AS $$
BEGIN
  RETURN count(*) FROM t1 WHERE id=p_id;
END
$$;
CREATE FUNCTION
```

Cette procédure stockée ne fait que renvoyer le nombre de lignes correspondant à la valeur de l'identifiant donné en argument pour la table **t1**, sans spécifier son schéma.

```
postgres=# SET search_path TO s1;
SET
postgres=# SELECT public.f3(1);
 f3
----
  1
(1 row)
```

Jusqu'ici, tout va bien. La valeur renvoyée par la fonction est bonne. Cependant, là, PostgreSQL a mis en cache le plan d'exécution. Changeons maintenant la valeur du paramètre **search_path** :

```
postgres=# SET search_path TO s2;
SET
postgres=# SELECT public.f3(1);
 f3
----
  1
(1 row)
```

On a le même résultat alors que les données sont différentes. Si on vérifie avec la requête de la procédure stockée :

```
postgres=# SELECT count(*) FROM t1 WHERE id=1;
 count
-------
     0
(1 row)
```

Le résultat est différent. En fait, le plan étant en cache, PostgreSQL ne cherche plus l'identifiant de la table. Il faudrait que PostgreSQL vide les plans en cache lorsque ce paramètre est modifié. C'est déjà le cas pour les requêtes préparées, mais pas pour les procédures stockées... jusqu'en version 9.3. Voyons ce que cela donne avec cette version :

```
postgres=# SET search_path TO s1;
SET
postgres=# SELECT public.f3(1);
 f3
----
  1
(1 row)

postgres=# SET search_path TO s2;
SET
postgres=# SELECT public.f3(1);
 f3
----
  0
(1 row)

postgres=# SELECT count(*) FROM t1 WHERE id=1;
 count
-------
     0
(1 row)
```

Cette modification risque de faire perdre un peu en performances à chaque changement du paramètre **search_path**, mais au moins, la planification se fera avec les bons objets.

3.2 Background workers

PostgreSQL est un système multi-processus. Dès que PostgreSQL est lancé, de nombreux processus apparaissent :

```
$ pg_ctl start
server starting
2014-01-31 18:33:58 CET [21469,1] LOG:  redirecting log output to logging collector process
2014-01-31 18:33:58 CET [21469,2] HINT:  Future log output will appear in directory "pg_log".
$ ps -ef | grep postgres
guillau+ 21469     1  0 18:33 pts/0    00:00:00 /opt/postgresql-9.3/bin/postgres
guillau+ 21476 21469  0 18:33 ?        00:00:00 postgres: logger process
guillau+ 21478 21469  0 18:33 ?        00:00:00 postgres: checkpointer process
guillau+ 21479 21469  0 18:33 ?        00:00:00 postgres: writer process
guillau+ 21480 21469  0 18:33 ?        00:00:00 postgres: wal writer process
guillau+ 21481 21469  0 18:33 ?        00:00:00 postgres: autovacuum launcher process
guillau+ 21482 21469  0 18:33 ?        00:00:00 postgres: archiver process
guillau+ 21483 21469  0 18:33 ?        00:00:00 postgres: stats collector process
```

Ces différents processus sont des processus en tâches de fond, chargés de différents traitements. Par exemple, le processus *autovacuum launcher* s'occupe de vérifier l'état des tables, et d'exécuter une opération de maintenance si elle s'avère nécessaire.

L'infrastructure de lancement de processus en arrière-plan est donc présent, il ne restait plus grand-chose à faire pour permettre de lancer des processus supplémentaires. Ceci arrive en 9.3. Ils sont appelés des *background workers*, autrement dit des travailleurs en arrière-plan. Ces programmes doivent être codés en C et ont accès, soit à la mémoire partagée, soit à une base de données, via une connexion SPI.

Différents exemples sont apparus, pour enregistrer les statistiques d'activité de PostgreSQL (https://github.com/gleu/stats_recorder) ou pour détecter automatiquement des modifications de configuration (https://github.com/ibarwick/config_log), ou encore pour simuler un serveur MongoDB dont les données sont enregistrées dans le serveur PostgreSQL (https://github.com/umitanuki/mongres). À ce niveau-là, tout dépend de l'imagination des développeurs qui voudront profiter de l'API de PostgreSQL.

Néanmoins, avant d'en installer un sur un serveur en production, il sera nécessaire d'être extrêmement prudent. En effet, ces démons ont accès à la mémoire partagée et au cœur de PostgreSQL. Il est donc assez aisé de rendre l'ensemble très instable si le processus supplémentaire est mal codé.

4 Régressions

Cette nouvelle version ne contient pas réellement de régressions. Cependant, comme à chaque fois, le fichier de configuration ne peut pas être repris tel quel. Par exemple, le paramètre **`replication_timeout`** a été renommé en **`wal_sender_timeout`**, suite à l'ajout du paramètre **`wal_receiver_timeout`**.

De même, le nommage des journaux de transactions n'utilisait pas la numérotation FF. Ceci est corrigé.

Autrement dit, il n'y a pas de régressions réellement problématiques pour les applications, ce qui est une excellente nouvelle.

5 Et le futur ?

Le développement de la version 9.4 est pratiquement terminé. Nous allons bientôt entrer dans la phase de stabilisation, qui devrait se terminer en mai. À ce moment-là sortira la première version bêta. Après quelques versions bêta, puis une ou deux *Release Candidate*, devrait sortir la version finale pour septembre 2014.

Cette nouvelle version apportera une amélioration des fonctionnalités déjà présentes (notamment celles introduites avec la 9.3), ainsi que la mise en place d'une infrastructure importante pour des fonctionnalités très attendues.

Dans les améliorations, on peut déjà remarquer le rafraîchissement non bloquant des vues matérialisées, la disponibilité de la clause **`WITH CHECK OPTION`** pour les vues modifiables automatiquement, ainsi que la gestion des clauses **`FILTER`** et **`WITHIN GROUP`** pour les fonctions d'agrégats. Une nouvelle instruction (**`ALTER SYSTEM`**) fera son apparition pour modifier à distance le fichier de configuration principal de PostgreSQL. Un nouveau module *contrib* sera ajouté pour faciliter la pré-chauffe du cache après redémarrage du serveur. Les background workers vont aussi gagner en fonctionnalités (possibilité d'utiliser un cache dynamique, etc.).

Quant à l'infrastructure, beaucoup de discussions ont lieu sur l'ajout d'une réplication logique interne à PostgreSQL. Ce travail est surtout mené par les équipes de la société 2ndQuadrant. Leur but est de faciliter la mise en place d'une réplication permettant de répliquer des données entre différentes versions majeures, de disposer d'une réplication multi-maître, ainsi que de choisir les éléments à répliquer (réplication partielle d'une instance, voire d'une base). Une autre infrastructure commence à être mise en place pour faciliter l'utilisation de plusieurs processeurs lors de l'exécution d'une seule requête. La société EnterpriseDB est très fortement impliquée dans ce développement. Malheureusement, tout ceci ne concernera pas la 9.4.

Conclusion

Cet article n'a évidemment abordé que les fonctionnalités les plus intéressantes. Il en existe beaucoup d'autres, comme l'ajout de nombreux opérateurs et fonctions pour le type de données JSON, l'implémentation de l'infrastructure pour des triggers sur événement, et bien d'autres encore.

PostgreSQL 9.3 apporte de nombreuses améliorations à un système de bases de données déjà très complet. Comme le début de la série 9, elle se focalise sur la réplication et l'extensibilité du moteur. Cependant, les développeurs ne se reposent pas sur leurs lauriers et continuent à améliorer le moteur pour la prochaine version (en toute logique la 9.4), mais réfléchissent aussi à la prochaine série (les 10.X), dont les thèmes seront plutôt l'exécution parallélisée de requêtes, la réplication logique des données et la gestion de très gros volumes de données. ■

CRÉER UN SCRIPT D'INSTALLATION AUTO-EXTRACTIBLE

par Thierry Gayet

Qui n'a jamais été intéressé par la mise à disposition d'un script bash auto-extractible pour la livraison d'un logiciel donné ? Nous allons voir comment cela fonctionne et quels outils peuvent être utilisés.

Introduction

Nombre solutions commerciales sont livrées via des scripts auto-extractibles « clefs en main ». Leur usage s'est quelque peu démocratisé et reste très pratique, car ils demeurent autonomes. Nous allons étudier leur principe de fonctionnement avant de présenter **makeself**, un script reprenant ces concepts et qui a été développé à l'origine par LOKI, un éditeur de jeux pour la plateforme GNU/Linux.

1 Fonctionnement

Pour développer un système de script auto-extractible, il nous faut en général trois scripts :

- un script **extract.sh**, qui sera inclus au script auto-extractible et qui aura pour fonction de séparer les données du script lui-même ;

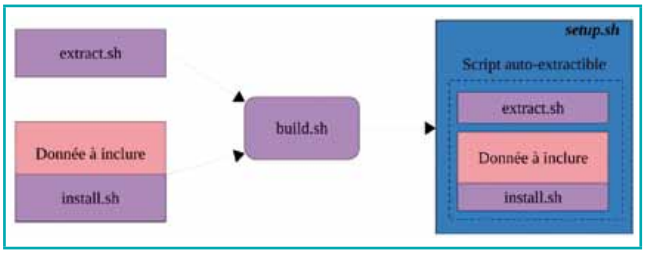


Fig. 1 : Résumé du principe utilisé lors de la génération du script « setup.sh »

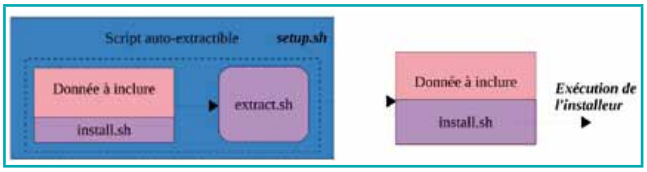


Fig. 2 : Résumé du principe utilisé lors de l'extraction des données du script « setup.sh »

- un script **install.sh**, qui sera exécuté une fois l'extraction des données effectuée ;
- un script **build.sh**, qui a pour rôle la génération du script auto-extractible concaténant le script **extract.sh** avec les données du logiciel.

Le script **build.sh** aura pour rôle de concaténer le script **extract.sh** avec les données. Ainsi, le fichier **setup.sh** par exemple aura la structure suivante :

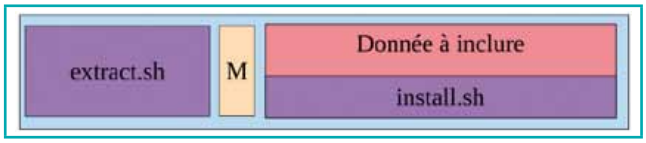


Fig. 3 : Détail du format du script auto-extractible

On remarquera le « marqueur », qui a pour rôle de délimiter la zone du script **extract.sh** et celle des données (le script **install.sh** étant inclus à la génération dans cette seconde, avant d'être retiré lors de l'installation).

1.1 Script de génération « build.sh »

Supposons que les trois scripts **build.sh**, **extract.sh** et **install.sh** soient localisés dans le répertoire **/opt/autoshell/** :

```
$ tree /opt/autorun
/opt/autorun
├── build.sh
├── extract.sh
└── install.sh
0 directories, 3 files
```

Associons le chemin vers ce répertoire dans une variable :

```
$ AUTORUN_PATH="/opt/autorun"
```

Le répertoire des données sera aussi défini dans une autre variable que l'on nommera **DATADIR** :

```
$ DATADIR="/home/tgayet/temp/data"
```

Pour commencer, copions le script **install.sh**, que l'on détaillera plus tard, dans le répertoire contenant les données à installer :

```
$ cp ${AUTORUN_PATH]/install.sh ${DATADIR}/
```

Créons une archive temporaire dans le répertoire **/tmp**, qui sera ensuite concaténée au script **extract.sh** de façon à former le script final **setup.sh** :

```
$ cd ${DATADIR}/
$ tar pczvf /tmp/tmp_compress.tar.gz ./*
```

Sortons du répertoire et revenons à notre dernier emplacement :

```
$ cd -
```

Concaténons donc le script d'extraction **extract.sh** avec l'archive que nous venons de générer :

```
$ cat ${AUTORUN_PATH]/extract.sh /tmp/tmp_compress.tar.gz > setup.sh
```

Supprimons l'archive créée temporairement :

```
$ rm -f /tmp/tmp_compress.tar.gz
```

Rendons notre script exécutable :

```
$ chmod u+x setup.sh
```

Cela nous donne au final le script suivant :

```
#!/bin/bash
#  --- build.sh ---
# ----------------------------------------------
# $1 : chemin vers les données
# $2 : path + nom du script final (défaut : setup.sh)
# $3 : chemin vers les scripts    (défaut : /opt/autorun)
# ----------------------------------------------

DEFAULT_AUTORUN_PATH=/opt/autorun
DEFAULT_FINAL_SCRIPTNAME=setup.sh
TMPDIR=/tmp
TEMP_ARCHIVE=${TMPDIR}/tmp_compress.$$.tar.gz
CURDIR=$(pwd)

# Test du nombre d'arguments (seul le premier l'est)
if [ "$#" -ne 1 ] || [ "$#" -ne 2 ] || [ "$#" -ne 3 ]; then
  echo "[BUILD] --> Nom du script final : $2 "
else
  echo "[BUILD] Nombre incorrect de paramètres. "
  echo "[BUILD] Usage: build.sh <chemin-donnees> <chemin-script-final>
<chemin-script-autorun> "
  exit 1
fi

# Test des paramètres et affectation des valeurs par défaut
DATADIR=$1
if [ "$2" == "" ]; then
  FINALSCRIPT="${DEFAULT_FINAL_SCRIPTNAME}"
else
  FINALSCRIPT="$2"
fi
if [ "$3" == "" ]; then
  AUTORUN_PATH="${DEFAULT_AUTORUN_PATH}"
else
  AUTORUN_PATH="$3"
fi

# Test de l'existence des chemins
echo -n "[BUILD] --> Vérification du chemin vers les scripts
contenant les données (${DATADIR}) : "
if [ -d ${DATADIR} ]; then
  echo "OK"
else
  echo "NOK !! Sortie..."
  exit 1
fi
echo -n "[BUILD] --> Vérification du chemin vers les scripts autorun
(${AUTORUN_PATH}) : "
if [ -d ${AUTORUN_PATH} ]; then
  echo "OK"
else
  echo "NOK !! Sortie..."
  exit 1
fi

# Test de l'existence du répertoire des données
echo -n "[BUILD] --> Generation de l'archive : "
cp ${AUTORUN_PATH}/install.sh ${DATADIR}/
cd ${DATADIR}
tar pczvf ${TEMP_ARCHIVE} ./* > /dev/null 2>&1
cd ${CURDIR}
cat ${AUTORUN_PATH}/extract.sh ${TEMP_ARCHIVE} > ${FINALSCRIPT}
#rm -f ${TEMP_ARCHIVE}
chmod +x ${FINALSCRIPT}
if [ -e ${FINALSCRIPT} ]; then
  echo "OK"
else
  echo "NOK !! Sortie..."
  exit 1
fi

exit 0

#
# --- END ---
#
```

L'utilisation est assez simple. Seul le premier paramètre est obligatoire ; les deux autres sont optionnels, car des valeurs par défaut leur sont associées.

Nous allons créer un répertoire de données et y copier un binaire **monbinaire.bin** à installer :

```
$ mkdir -p archive01/
$ cp  monbinaire.bin  archive01/
$ tree archive01/
archive01/
└── monbinaire.bin

0 directories, 1 file
```

Voici trois exemples illustrant les usages des paramètres :

```
$ export PATH="${PATH} :/opt/autorun"
$ build.sh /home/tgayet/tmp/archive01/
$ build.sh /home/tgayet/tmp/archive01/ install.bin
$ build.sh /home/tgayet/tmp/archive01/ install.bin /media/tools/autorun/
```

Le script en action :

```
$ build.sh /home/tgayet/tmp/archive01/
Archive temporaire : /tmp/tmp_compress.23604.tar.gz
--> Nom du script final :
--> Vérification du chemin vers les scripts contenant les données (/home/tgayet/
archive01/) : OK
```

```
--> Vérification du chemin vers les scripts autorun (/opt/autorun) : OK
--> Generation de l'archive
--> Vérification de l'archive (setup.sh) : OK
```

Une fois le script auto-extractible généré, on peut vérifier que l'on obtient bien une concaténation du script d'extraction avec l'archive contenant les données à installer :

```
$ ls -al setup.sh
-rwxrwxr-x 1 tgayet tgayet 39175989 janv. 26 13:58 setup.sh
tgayet@PCL131017:~/Documents/articles_lm/script_auto_extractible$ file setup.sh
setup.sh: data
```

La vérification peut aussi se constater via un éditeur.

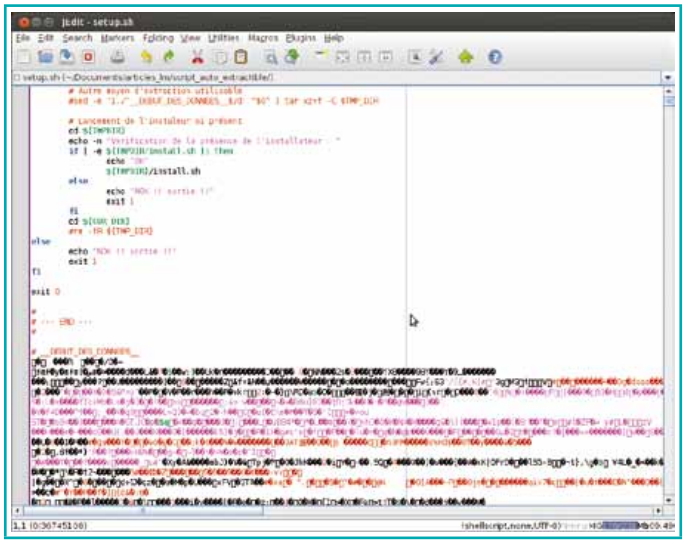


Fig. 4

1.2 Script d'extraction « extract.sh »

Dans le principe, le script d'extraction sera exécuté lors du lancement du script auto-extractible et aura la fonction de retrouver un marqueur dans le script servant à délimiter la frontière entre le script d'extraction et les données à extraire. Un fois les données extraites dans un répertoire temporaire, s'il est présent, un script d'installation sera lancé.

Concernant l'extraction, il faut bien entendu que l'algorithme de compression soit similaire à celui utilisé lors de la génération.

On commence par créer un répertoire temporaire dans **/tmp** pour l'extraction, en s'assurant que le nommage soit unique :

```
$ TMPDIR=$(mktemp -d tmp_extract_XXXXXX --tmpdir=/tmp)
```

Ensuite, grâce à la commande **awk**, on va repérer la position d'un marqueur dans le script de façon à s'en servir ensuite dans la commande **tail** pour extraire les lignes allant de l'offset à la fin du fichier. Le marqueur utilisé **__DEBUT_DES_DONNEES__** sera placé à la fin du script d'extraction, car il servira ensuite comme délimiteur des deux zones.

La sortie issue de l'extraction sera redirigée dans un fichier temporaire, puis extraite dans le répertoire précédemment créé :

```
$ ARCHIVE_OFFSET="`awk '/^__DEBUT_DES_DONNEES__/ { print NR+1; exit 0; }' ${SCRIPTNAME}`"
$ TEMP_ARCHIVE=/tmp/tmp_compress.$$.tar.gz
$ tail -n+${ARCHIVE_OFFSET} ${SCRIPTNAME} >> ${TEMP_ARCHIVE}
$ tar zxvf ${TEMP_ARCHIVE} -C ${TMPDIR}
```

Une fois les données extraites, on lance le script d'installation :

```
$ ${TMPDIR}/install.sh
```

Cela nous donne au final le script suivant :

```
#!/bin/bash
#  --- extract.sh ---
# ------------------------------------------

CURDIR=$(pwd)
SCRIPTNAME=$0
MARQUEUR="__DEBUT_DES_DONNEES__"
TEMP_ARCHIVE=/tmp/tmp_compress.$$.tar.gz

# Création d'un répertoire temporaire
TMPDIR=$(mktemp -d tmp_extract_XXXXXX --tmpdir=/tmp)

# Repérage de l'offset à partir du marqueur délimitant le script des données
ARCHIVE_OFFSET="`awk '/^__DEBUT_DES_DONNEES__/ { print NR+1; exit 0; }'
${SCRIPTNAME}`"
NBLINES="`wc -l ${SCRIPTNAME} |awk -F ' ' '{print $1}'`"
echo "[EXTRACT] --> Marqueur ${MARQUEUR} trouvé à l'offset : ${ARCHIVE_OFFSET} /
${NBLINES} "

# Extraction de l'archive contenant les données en utilisant le marqueur
echo "[EXTRACT] --> Extraction des données "
tail -n+${ARCHIVE_OFFSET} ${SCRIPTNAME} >> ${TEMP_ARCHIVE}
tar zxvf ${TEMP_ARCHIVE} -C ${TMPDIR} > /dev/null 2>&1

# Autres moyens d'extraction utilisables
#tail -n+${ARCHIVE_OFFSET} ${SCRIPTNAME} | tar xzvf -C ${TMPDIR}
#sed -e '1,/^__DEBUT_DES_DONNEES__$/d' "${SCRIPTNAME}" | tar xzvf -C ${TMPDIR}

# Lancement de l'installateur si présent
cd ${TMPDIR}
echo -n "[EXTRACT] --> Verification de la présence de l'installateur : "
if [ -e ${TMPDIR}/install.sh ]; then
  echo "OK"
  echo "[EXTRACT] --> Lancement du script d'installation..."
  ${TMPDIR}/install.sh
else
  echo "NOK !! sortie !!"
  exit 1
fi
cd ${CUR_DIR}

rm -fR ${TMP_DIR}
rm -f  ${TEMP_ARCHIVE}

exit 0

#
# --- END ---
#

__DEBUT_DES_DONNEES__
```

Deux autres méthodes d'extraction étaient également possibles :

```
#tail -n+${ARCHIVE_OFFSET} ${SCRIPTNAME} | tar xzvf -C ${TMPDIR}
#sed -e '1,/^__DEBUT_DES_DONNEES__$/d' "${SCRIPTNAME}" | tar xzvf -C ${TMPDIR}
```

Le script n'inclut pas de paramètres au lancement, mais pourrait en disposer.

Le script en action :

```
./setup.sh
--> Marqueur __DEBUT_DES_DONNEES__ trouvé à l'offset : 54 / 155574
--> Extraction des données
--> Verification de la présence de l'installateur : OK
--> Lancement du script d'installation...
```

1.3 Script d'installation « install.sh »

Le script d'installation sert simplement à créer un répertoire d'installation et à recopier les fichiers extraits :

```
$ NOMPACKAGE="monpackage"
$ SRCDIR=$(dirname $0)
$ DSTDIR="/usr/local/${NOMPACKAGE}"
$ mkdir -p ${DSTDIR}
$ cp -par ${SRCDIR}/* ${DSTDIR}/
```

On supprime au final ce script d'installation :

```
$ rm -fR ${DSTDIR}/install.sh
```

Cela nous donne le script suivant :

```
#!/bin/bash
#  --- install.sh ---
# ------------------------------------------------

CURDIR=$(pwd)
SRCDIR=$(dirname $0)
NOMPACKAGE="monpackage"
DSTDIR="/usr/local/${NOMPACKAGE}"

# Test et/ou création du répertoire de destination
echo -n "[INSTALL] --> Testing destination directory (${DSTDIR}) : "
if [ ! -d ${DSTDIR} ]; then
  echo "NOK ... Creation au préalable "
  mkdir -p ${DSTDIR}
  if [ ! -d ${DSTDIR} ]; then
    exit 1
  fi
else
  echo "OK"
fi

# Copie des données dans le répertoire
echo -n "[INSTALL] --> Recopie récursive des données du package : "
cp -par ${SRCDIR}/* ${DSTDIR}/
echo "OK"

rm -fR ${DSTDIR}/install.sh

exit 0

#
# --- END ---
#
```

Pour l'exécution du script d'installation via le script d'extraction, les droits d'administration sont requis et donc, un lancement via la commande **sudo** ou **fakeroot** :

```
sudo ./setup.sh
[sudo] password for tgayet:
[EXTRACT] --> Marqueur __DEBUT_DES_DONNEES__ trouvé à l'offset : 54 / 155135
[EXTRACT] --> Extraction des données
[EXTRACT] --> Verification de la présence de l'installateur : OK
[EXTRACT] --> Lancement du script d'installation...
[INSTALL] --> Testing destination directory (/usr/local/monpackage) : NOK ...
Creation au préalable
[INSTALL] --> Recopie récursive des données du package : OK
--------------------------------------------------------------------------------
```

Suite à l'installation, nous pouvons constater que tout s'est bien passé :

```
$ ls -al /usr/local/monpackage
total 38860
drwxr-xr-x  2 root   root       4096 janv. 26 13:44 .
drwxr-xr-x 12 root   root       4096 janv. 26 13:44 ..
-rw-r--r--  1 tgayet tgayet 39777435 janv. 26 09:47 monbinaire.bin
```

En effet, sans les permissions requises, cela bloque dès la création du répertoire :

```
./setup.sh
[EXTRACT] --> Marqueur __DEBUT_DES_DONNEES__ trouvé à l'offset : 54 / 155135
[EXTRACT] --> Extraction des données
[EXTRACT] --> Verification de la présence de l'installateur : OK
[EXTRACT] --> Lancement du script d'installation...
[INSTALL] --> Testing destination directory (/usr/local/monpackage) : NOK ...
Creation au préalable
mkdir: impossible de créer le répertoire "/usr/local/monpackage": Permission non
accordée
```

Le script d'installation est la partie à personnaliser, car pouvant être spécifique à chaque installation de package. Au lieu de le réécrire à chaque fois, l'approche la plus intéressante serait de passer des paramètres spécifiques à chaque package.

2 makeself

Maintenant que l'on a détaillé le fonctionnement des scripts shell auto-extractibles, je vais présenter **makeself**, un petit script qui rend pas mal de services dans le monde des installateurs.

2.1 Présentation

makeself.sh est un petit script shell générant une archive shell auto-extractible à partir d'un répertoire source. Le procédé est similaire aux archives générées avec WinZip Self-Extractor dans le monde Windows. **makeself** gère également des sommes de contrôle d'intégrité (CRC et/ou de contrôle MD5). Le script est développé de façon à rester le plus portable possible, c'est-à-dire sans trop se fonder sur des fonctionnalités spécifiques au système. Il est associé à la licence GPLv2.

Le script **makeself**, ainsi que les archives qu'il génère, doivent pouvoir s'exécuter sur n'importe quel serveur Unix

compatible Bourne shell. En plus du shell, le programme utilisé pour la compression est aussi nécessaire.

Depuis la version 2.1, **makeself** a été réécrit et testé sur les plateformes suivantes :

- Linux (toutes les distributions)
- Sun Solaris (8 et plus)
- HP-UX (testé sur 11.0 et 11i sur APPS RISC)
- SCO OpenServer et OpenUnix
- IBM AIX 5.1L
- Mac OS X (Darwin)
- SGI IRIX 6.5
- FreeBSD
- Unicos / Cray
- Cygwin (Windows)

Parmi les usages courants de **makeself** dans les logiciels accessibles au grand public, il est possible de citer :

- Les patchs et les installateurs pour les jeux id Software (*Quake 3* pour Linux, ou *Return To Castle Wolfenstein*) ;
- Tous les patchs de jeux publiés par Loki Software pour la version Linux de jeux populaires ;
- Les pilotes Nvidia pour Linux ;
- L'installation de la version Linux de Google Earth ;
- Les installateurs de VirtualBox pour Linux ;
- Le package **makeself** lui-même ;
- Et de nombreux autres...

2.2 Récupération du script

Pour récupérer la dernière version du script **makeself**, on peut soit télécharger la dernière archive :

```
$ wget http://megastep.org/makeself/makeself-2.1.5.run
requête HTTP transmise, en attente de la réponse... 200 OK
Taille : 38210 (37K)
Enregistre : "makeself-2.1.5.run"
2014-01-26 18:11:43 (53,1 KB/s) - "makeself-2.1.5.run" enregistré [38210/38210]
```

Soit passer par le repo GIT de GitHub :

```
$ git clone https://github.com/megastep/makeself.git
Cloning into 'makeself'...
remote: Reusing existing pack: 472, done.
remote: Total 472 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (472/472), 154.34 KiB | 264 KiB/s, done.
Resolving deltas: 100% (281/281), done.
```

On peut vérifier la version :

```
$ cd makeself/
$ ./makeself.sh --version
Makeself version 2.2.0
```

2.3 Détail des paramètres

La syntaxe de **makeself** est la suivante :

```
makeself.sh [ args ] archive_rep description script_final install_
script [ install_script_args ]
```

args sont des options facultatives pour **makeself**. Celles qui sont disponibles sont les suivantes :

- **--version** : affiche le numéro de version de **makeself** sur la sortie standard ;
- **--gzip** : utilise **gzip** pour la compression (la valeur par défaut sur les plateformes sur lesquelles **gzip** est généralement disponible, notamment Linux) ;
- **--bzip2** : utilise **bzip2** au lieu de **gzip** pour une meilleure compression ; la commande **bzip2** doit être disponible dans le **path** courant ;
- **--pbzip2** : utilise **pbzip2** au lieu de **gzip** pour une meilleure compression et plus rapide sur des machines possédant plusieurs processeurs ou cœurs ;
- **--xz** : utilise **xz** au lieu de **gzip** pour une meilleure compression ;
- **--base64** : encode l'archive au format ASCII suivant le format base64 (nécessite la disponibilité de la commande **base64** dans le PATH) ;
- **--compresse** : utilise la commande Unix **compress** pour compresser les données ; c'est la valeur par défaut sur toutes les plateformes qui n'ont pas **gzip** disponible ;
- **--nocomp** : n'active pas de compression de l'archive, qui sera alors un simple fichier **tar** non-compressé ;
- **--complevel** : spécifie le niveau de compression pour **gzip**, **bzip2**, **pbzip2** et **xz**. (la valeur par défaut est 9) ;
- **--notemp** : l'archive générée n'utilisera pas de répertoire temporaire pour l'extraction des fichiers. Cependant, un nouveau répertoire sera créé dans le chemin courant. C'est mieux pour distribuer des logiciels qui peuvent s'extraire et se compiler par eux-mêmes (par exemple, lancer la compilation via le post-script intégré ; c'est intéressant pour des applications ou des drivers devant se re-compiler avec les *headers* du noyau courant) ;
- **--current** : les fichiers seront extraits dans le répertoire courant plutôt que dans un sous-répertoire ; cette option implique **--notemp** ci-dessus ;
- **--follow** : suit les liens symboliques à l'intérieur du répertoire d'archives, c'est-à-dire stocke les fichiers qui sont pointés à la place des liens eux-mêmes ;
- **--append** (nouveau depuis la version 2.1.x) : permet l'ajout de données à une archive existante au lieu d'en

créer une nouvelle. Dans ce mode, les paramètres de l'archive d'origine sont réutilisés (type de compression, étiquette, script intégré), et n'ont donc pas besoin d'être spécifiés à nouveau sur la ligne de commandes ;

- **--header** : makeself 2.0 utilise un fichier distinct pour stocker le stub d'entête, appelé **makeself-header.sh**. Par défaut, il est supposé qu'il est stocké dans le même emplacement que **makeself.sh**. Cette option peut être utilisée pour spécifier son emplacement actuel si elle est stockée ailleurs ;
- **--copy** : après extraction, l'archive s'extraira d'abord dans un répertoire temporaire. L'application principale est de permettre aux installateurs autonomes stockés dans une archive **makeself** sur un CD par exemple, d'éviter des problèmes d'occupation de ressources. Cela est aussi utile sur des installations multi-CD en démontant le premier support et en montant le suivant ;
- **--nox11** : désactive le « spawning » automatique d'un nouveau terminal sous X11 ;
- **--nowait** : lorsqu'il est exécuté à partir d'un nouveau terminal X11, désactive l'invite de l'utilisateur à la fin de l'exécution du script ;
- **--nomd5** et **--nocrc** : désactive la création d'un *checksum* MD5/CRC pour l'archive. Cela accélère le processus d'extraction si la vérification d'intégrité n'est pas nécessaire ;
- **--lsm fichier** : fournit un fichier LSM à **makeself**. Il sera intégré dans l'archive générée. Les fichiers LSM décrivent un logiciel d'une manière qui est facilement analysable. L'entrée LSM peut alors être récupérée plus tard, en utilisant l'argument **--lsm** du script auto-extractible. Un exemple d'un fichier de LSM est fourni avec le package **makeself** lui-même ;
- **--tar-extra opt** : ajoute plus d'options à la ligne de commandes **tar**.

Pour les autres éléments de la syntaxe :

- **archive_rep** : nom du répertoire qui contient les fichiers à archiver ;
- **script_final** : nom de l'archive auto-extractible à créer ;
- **description** : chaîne de texte arbitraire décrivant le paquet ; elle sera affichée lors de l'extraction des fichiers ;
- **install_script** : commande à exécuter à partir du répertoire d'extraction une fois celle-ci terminée. Ainsi, si vous souhaitez exécuter un programme contenu dans le répertoire d'installation, vous devez faire précéder votre commande de **./**. Les **install_script_args** sont des arguments additionnels pour le script d'installation.

Il est recommandé que le préfixe de l'archive soit associé à l'algorithme de compression, comme **.bz2.run** pour **bzip2**, de sorte que les utilisateurs potentiels sachent qu'ils auront besoin de **bzip2** pour l'extraction.

Pour permettre le téléchargement des archives shell auto-extractibles depuis un serveur web comme Apache 2, sous forme binaire et non sous forme texte, il est nécessaire de définir un nouveau type MIME dans **/etc/apache2/httpd.conf** :

```
AddType application / x-makeself. Terme
```

Pour les archives créées avec **makeself** avant la version 2.1.2, sous certaines distributions GNU/Linux, de vieilles syntaxes pour les commandes **tail** et **head** vont devenir progressivement obsolètes. Par conséquent, certains problèmes peuvent apparaître au moment de la décompression des archives. La solution, pour cela, est de définir la variable d'environnement **POSIX2VERSION**, afin de permettre à l'ancienne syntaxe de toujours fonctionner :

```
$ export _POSIX2_VERSION = 199209
```

2.4 Utilisation du script « makeself.run »

En considérant l'explication donnée plus haut, nous allons repartir du répertoire **archive01/** et du script d'installation **install.sh** ; nous allons demander au script **makeself** de générer un shell auto-extractible. Remémorons-nous le contenu du répertoire **archive01/** :

```
$ tree archive01/
archive01/
└── monbinaire.bin

0 directories, 1 file
```

Puis, demandons la génération de notre script auto-extractible :

```
$ makeself.sh ./archive01/ setup.sh "ma première
application" install.sh
Header is 499 lines long

About to compress 38852 KB of data...
Adding files to archive named "setup.sh"...
./
./monbinaire.bin
CRC: 3792604588
MD5: 338093afec6e6a0f1ad2e9dca69028d4

Self-extractable archive "setup.sh" successfully created.
```

Le résultat donne un fichier **setup.sh**, comme nous l'avions également généré :

```
$ ls -al setup.sh
-rwxrwxr-x 1 tgayet tgayet 39184403 janv. 27 18:07 setup.sh
$ file setup.sh
setup.sh: data
```

L'archive **setup.sh** peut fournir des infos :

```
$ ./setup.sh --info
Identification: ma première applicatio n
Target directory: archive01
Uncompressed size: 38852 KB
Compression: gzip
Date of packaging: Mon Jan 27 18:07:43 CET 2014
Built with Makeself version 2.2.0 on
Build command was: ./makeself/makeself.sh \
    "./archive01" \
    "setup.sh" \
    "ma première application" \
    "/opt/autorun/install.sh"
Script run after extraction:
     /opt/autorun/install.sh
archive01 will be removed after extraction
```

Nous pouvons également demander à lister les fichiers :

```
$ ./setup.sh --list
Target directory: archive01
drwxrwxr-x tgayet/tgayet     0 2014-01-26 18:03 ./
-rw-r--r-- tgayet/tgayet 39777435 2014-01-26 09:47 ./monbinaire.bin
```

L'intégrité des données peut être vérifiée :

```
$ ./setup.sh --check
Verifying archive integrity... MD5 checksums are OK. All good.
```

Bon, maintenant, essayons de lancer une installation :

```
$ sudo ./setup.sh
[sudo] password for tgayet:
Verifying archive integrity... All good.
Uncompressing ma première applicatio  100%
[INSTALL] --> Testing destination directory (/usr/local/monpackage) : OK
[INSTALL] --> Recopie récursive des données du package : OK
```

Voici un second exemple détaillant comment l'archive **makeself.run** elle-même a été créée :

```
$ makeself.sh - notemp makeself makeself.run "makeself par Stéphane
Peter" echo "makeself a extrait itsel "
```

2.5 Utilisation des shells auto-extractibles

Les archives générées avec **makeself** (au moins pour la version 2.1) utilisent les arguments suivants :

- **--keep** : informe que les fichiers qui seront extraits après la décompression du script shell auto-extractible seront conservés après l'installation jusqu'à suppression par l'utilisateur ;
- **--verbose** : invite l'utilisateur avant d'exécuter la commande intégrée utilisée pour l'installation ;
- **--target répertoire** : permet de préciser un répertoire d'extraction ;
- **--nox11** : ne « spawn » pas les terminaux X11 ;
- **--confirm** : demande confirmation à l'utilisateur avant d'exécuter la commande intégrée utilisée pour l'installation ;
- **--info** : affiche les informations générales sur l'archive sans l'extraire ;
- **--lsm** : affiche l'entrée LSM si elle est présente ;
- **--list** : précise la liste des fichiers de l'archive ;
- **--check** : vérifie l'intégrité de l'archive en utilisant les sommes de contrôle embarquées ; n'effectue pas l'extraction de l'archive ;
- **--nochown** : par défaut, la commande **chown -R** est exécutée sur le répertoire cible après extraction, de sorte que tous les fichiers appartiennent à l'utilisateur courant. Ceci est surtout nécessaire si vous exécutez en tant que **root**, car la commande **tar** essayera ensuite de réassigner les droits de l'utilisateur initial. Ce comportement peut donc être désactivé avec ce paramètre ;
- **--tar** : exécute la commande **tar** sur le contenu de l'archive, en utilisant les arguments suivants comme paramètres de la commande ;
- **--noexec** : n'exécute pas le script intégré après l'extraction ;

Tous les arguments suivant l'archive seront passés comme arguments supplémentaires à la commande intégrée. Vous devez explicitement utiliser **-** avant de telles options pour s'assurer que **makeself** n'essaye pas de les interpréter.

2.6 Exemple de fichier de description d'archive LSM

makeself gère un système de description des paquets, qui peut être inclus aux shells auto-extractibles.

Un exemple de description LSM utilisée pour le paquet **makeself** est fourni avec le script même. Cela donne une idée pour la rédaction.

```
Begin3
Title:          makeself.sh
Version:        2.2.0
Description:    makeself.sh is a shell script that generates a self-extractable
                tar.gz archive from a directory. The resulting file appears as a shell
                script, and can be launched as is. The archive will then uncompress
                itself to a temporary directory and an arbitrary command will be
                executed (for example an installation script). This is pretty similar
                to archives generated with WinZip Self-Extractor in the Windows world.
Keywords:       Installation archive tar winzip
Author:         Stephane Peter (megastep@megastep.org)
Maintained-by:  Stephane Peter (megastep@megastep.org)
Original-site:  http://github.com/megastep/makeself
Platform:       Unix
Copying-policy: GPL
End
```

Conclusion

Vous voilà éclairé sur l'usage des scripts auto-extractibles. Vous pouvez désormais écrire vos propres scripts d'installation, ou bien utiliser **makeself**, voire d'autres solutions alternatives comme GNU/Sharutils. ■

Liens

- http://megastep.org/makeself/
- http://megastep.org/makeself/makeself-2.1.5.run
- https://github.com/megastep/makeself
- http://www.linuxjournal.com/content/add-binary-payload-your-shell-scripts
- http://tvaira.free.fr/dev/methodologie/deploiement_script_autoextractible.pdf
- http://www.linuxjournal.com/node/1005818
- https://bugzilla.icculus.org/
- http://www.gnu.org/software/sharutils/

22ème édition / *22nd Edition*

rts EMBEDDED SYSTEMS

Systèmes Temps Réel & Embarqués
Affichage et Visualisation - Conception et test de systèmes électroniques
Real-time and Embedded Systems
Digital display and viewing
Electronic System design and Testing

Les Salons Solutions ELECTRONIQUES

EXPOSITION - CONFERENCES - ATELIERS
TRADE SHOW - CONFERENCES - WORKSHOPS

- Cartes, composants et modules
  Cards, components and modules
- OS temps réel et embarqués
  RT and Embedded OS
- Environnements de développement
  Development Environment Tools
- Outils de test et de validation
  Software testing and validation tools
- Conception et test de systèmes électroniques
  Electronic System Design & Testing
- PC Industriels / *Industrial PCs*
- Ecrans / *Screens*

9ème édition / *9th Edition*

toMachine to Machine

MtoM le salon des professionnels des objets communicant
The MtoM show for professionals in communicating objects

- Opérateurs telecoms ou MVNO
  TelCos or MVNO
- Fournisseurs de solutions réseaux
  Network solutions providers
- Constructeurs ou distributeurs
  Manufacturers or distributors
- SSII et cabinets de consulting /
  Consulting and services companies
- Intégrateurs de solutions
  Solutions integrators

19 & 20 mars 2014
March 19th & 20th, 2014

CNIT PARIS LA DEFENSE

BIRP

Partenaire officiel : ELECTRONIQUE S

www.salons-solutions-electroniques.fr

ROUTAGE DES IP DE SERVICE SUR UN RÉSEAU LOCAL

par Arnaud Gomes-do-Vale [utilisateur de GNU/Linux depuis 1997, administrateur systèmes et réseaux depuis 2003]

J'ai, comme pas mal de gens, un petit réseau domestique. J'ai, comme probablement un peu moins de gens, une petite poignée d'adresses IPv4 publiques. J'ai donc naturellement essayé d'en tirer le meilleur parti.

Pour fixer les idées, disons que je dispose de la plage 192.0.2.72/29, c'est-à-dire les adresses de 192.0.2.72 à 192.0.2.79.

Le réflexe naturel serait de les utiliser directement sur le réseau local en se passant d'adresses RFC 1918. Malheureusement, dans mon cas, j'ai tout simplement trop de machines. J'ai donc commencé par séparer mon réseau local en deux segments Ethernet (en fait deux VLAN, mais ça n'a pas d'importance pour ce qui nous occupe), l'un en adressage privé, et l'autre pour mes adresses publiques.

C'est bon alors, l'article est fini ? Hé non. Si cette solution fonctionne à peu près, elle a l'immense inconvénient de bloquer deux de nos huit adresses pour le réseau et la diffusion. Ça fait 25% de gaspillage, et ça heurte ma fibre écolo. Du coup, on va essayer de faire mieux, et ça aura en plus le bon goût d'être plus simple, même si ce n'est pas évident à première vue.

1 Notre réseau

Nous allons monter une petite maquette avec trois machines, la généralisation est laissée en exercice au lecteur.

La première de ces machines fait office de routeur et s'appelle donc, fort logiquement, **`routeur`**. Elle dispose d'une interface Ethernet, **`eth0`**, sur le réseau local, et d'une autre interface raccordée à Internet d'une façon ou d'une autre. Je ne vais pas m'étendre sur la configuration du routage ou de la traduction d'adresses, il y a plein de docs là-dessus et ce n'est de toute façon pas le sujet ; on va juste supposer que le routeur est configuré « comme il faut » pour remplir sa fonction.

Les deux autres machines s'appelleront **`serveur1`** et **`serveur2`**, et je vais leur affecter les adresses IP 192.0.2.72 et 192.0.2.73, respectivement.

« *Diantre !* », vous exclamerez-vous, « *Mais il me prend pour une bille ! 192.0.2.72, c'est l'adresse du réseau* !». Oui, mais non. Ce serait certes le cas si nous utilisions un réseau à diffusion, typiquement Ethernet, mais pas dans le cas présent. Nos huit adresses IP seront rigoureusement interchangeables, ce qui est tout de même le but de la manip'.

À ce sujet, vous noterez par la suite que le routeur lui-même n'a pas besoin d'utiliser une adresse publique. Et en voilà encore une de gagnée !

Pour fixer les idées, notre réseau local utilisera la plage 10.0.5.0/24, le routeur aura l'adresse 10.0.5.1 et les serveurs seront en 10.0.5.11 et 10.0.5.12. Pour les besoins de l'exemple les trois machines tourneront sous Debian Wheezy ; à vous de transposer pour votre système d'exploitation préféré.

2 L'interface dummy

Le noyau Linux permet de monter une interface Ethernet bidon, qui est essentiellement une variante de l'interface de boucle locale (*loopback*) que vous connaissez probablement déjà. Sans entrer dans les détails, la boucle locale de Linux a parfois des comportements un peu inhabituels, alors que l'interface bidon ressemble plus à une interface Ethernet ; c'est donc elle qu'on utilise habituellement pour monter une adresse qu'on ne veut pas rendre directement accessible à l'extérieur de la machine. Toujours sans entrer dans les détails, si vous utilisez un système d'exploitation différent pour vos serveurs, par exemple un BSD, vous pouvez parfaitement utiliser l'interface **`lo0`**.

On va donc commencer par ajouter une interface bidon à nos serveurs pour y affecter leurs adresses publiques. Je sais, monter une adresse publique sur une interface inaccessible de l'extérieur peut sembler étrange ; pas d'inquiétude, on parlera routage d'ici trois paragraphes.

Voici donc la configuration de l'interface bidon de **`serveur1`** (dans le fichier **`/etc/network/interfaces`**, je vous laisse trouver l'équivalent pour votre distribution) ; il faudra bien sûr écrire l'équivalent sur **`serveur2`**, que je ne vais pas détailler ici.

```
auto dummy0
iface dummy0 inet static
        address 192.0.2.72
        netmask 255.255.255.255
```

On peut maintenant activer notre interface :

```
root@serveur1:~# ifup dummy0
root@serveur1:~# ip a sh dev dummy0
3: dummy0: <BROADCAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN
    link/ether 06:27:e7:f8:df:59 brd ff:ff:ff:ff:ff:ff
    inet 192.0.2.72/32 brd 192.0.2.72 scope global dummy0
    inet6 fe80::427:e7ff:fef8:df59/64 scope link
       valid_lft forever preferred_lft forever
```

Voilà, notre serveur connaît son adresse publique ! Par contre, le routeur ne sait bien entendu pas comment la joindre, il va donc falloir lui indiquer :

```
root@routeur:~# ip r add 192.0.2.72/32 via 10.0.5.11
root@routeur:~# fping 192.0.2.72
192.0.2.72 is alive
```

Voilà, je vous avais dit qu'on allait parler routage, c'est fait. Enfin, presque, reste à rendre la configuration persistante ; sur une Debian, ça se fait en installant le paquet **ifupdown-extra** et en ajoutant les routes au fichier **/etc/network/routes** :

```
root@routeur:~# cat >> /etc/network/routes
192.0.2.72 255.255.255.255 10.0.5.11 any
192.0.2.73 255.255.255.255 10.0.5.12 any
root@routeur:~# /etc/init.d/networking-routes start
[ ok ] Configuring network routes...done.
root@routeur:~# ip r sh dev eth0
10.0.5.0/24  proto kernel  scope link  src 10.0.5.1
192.0.2.72 via 10.0.5.11
192.0.2.73 via 10.0.5.12
```

C'est fini alors, on peut s'arrêter là ? Non, toujours pas. Pour comprendre pourquoi, on va aller faire un tour sur **serveur2** et voir s'il peut joindre l'adresse publique de **serveur1**.

```
root@serveur2:~# fping 192.0.2.72
192.0.2.72 is alive
```

Jusqu'ici tout va bien. Et si on creuse un peu ?

```
root@serveur2:~# traceroute -I 192.0.2.72
traceroute to 192.0.2.72 (192.0.2.72), 30 hops max, 60 byte packets
 1  10.0.5.1 (10.0.5.1)  1.556 ms  1.526 ms  1.350 ms
 2  192.0.2.72 (192.0.2.72)  2.126 ms  2.118 ms  2.104 ms
```

Les paquets passent par le routeur, alors que les deux serveurs sont raccordés au même segment Ethernet. Ça marche, mais ce n'est pas optimal.

On peut bien sûr ajouter une route statique de **serveur2** vers **serveur1**, et réciproquement. Mais si le nombre de machines augmente, ça va finir par faire beaucoup de routes, et du boulot pour tout mettre à jour à chaque fois qu'on voudra déplacer une adresse publique. Or, je suis administrateur système, du coup je suis fainéant, c'est dans la fiche de poste. Alors on va arrêter de gérer le routage à la main, on va configurer les machines pour qu'elles le fassent elles-mêmes et on va les regarder bosser, non mais !

3 OSPF

Il existe un certain nombre de protocoles de routage dynamiques qui peuvent faire le boulot à notre place. Vous me direz que ces protocoles sont conçus pour synchroniser la configuration de routeurs, pas de serveurs. Il se trouve que dans le cas présent, chaque serveur joue aussi le rôle de routeur entre notre réseau local et le /32 qui contient son adresse publique. On est dans les clous, on peut continuer, bonne nouvelle !

J'ai choisi d'utiliser le protocole OSPF, qui est bien adapté à un réseau local avec un nombre fluctuant de routeurs. Il s'agit d'un protocole de couche 4, qui fonctionne donc directement sur IP à côté de TCP et UDP. Plus précisément, nous allons utiliser sa version 2, qui fonctionne sur IPv4 et qui est définie dans la RFC 2328. Je mentionnerai la version 3 un peu plus tard, quand j'aborderai la question d'IPv6, mais pour le moment, ne nous dispersons pas et restons sur OSPFv2.

Avant d'entrer dans le vif du sujet, une description rapide du protocole. Je me contenterai du cas qui nous occupe, c'est-à-dire un réseau à diffusion tout simple ; OSPF peut aussi fonctionner sur des liens point à point, ou sur des réseaux à topologie plus compliquée, mais nous n'utiliserons pas cette possibilité ici.

Le principe de base est relativement simple. Chaque routeur commence par faire l'inventaire des réseaux auxquels il est directement connecté. Il s'annonce ensuite en multicast sur chacun de ces réseaux pour découvrir ses voisins et établir une relation d'adjacence avec eux. Chaque routeur échange ensuite les informations de topologie dont il dispose avec les routeurs adjacents. Pour finir, une fois que tous les routeurs ont une vue d'ensemble de la topologie du réseau, chacun génère sa table de routage en utilisant l'algorithme de Dijkstra.

La découverte des voisins et l'établissement des relations d'adjacence mérite un peu d'approfondissement. Chaque routeur émet régulièrement des paquets OSPF **hello** sur le groupe multicast 224.0.0.5. Les routeurs d'un même domaine de diffusion (donc, dans notre cas, toutes nos machines) élisent un routeur désigné (DR) et un routeur désigné de secours (BDR). Ensuite, tous les routeurs du domaine établissent un lien d'adjacence avec le DR, et le BDR se tient prêt à prendre le relais en cas de disparition du DR. Cette optimisation permet de réduire le nombre de liens d'adjacence et donc, le trafic OSPF global ; elle n'a par contre aucun impact sur le calcul de la table de routage.

Pour finir, quelques notions techniques que nous verrons passer un peu plus loin.

Un réseau important peut être segmenté en plusieurs zones, numérotées sur 32 bits. L'usage veut qu'on note les numéros de zone sous la forme de quatre octets séparés par des points, comme une adresse IPv4. La zone 0.0.0.0 est privilégiée ; c'est la dorsale à laquelle toutes les autres zones sont directement reliées. Cette segmentation permet de rendre les zones autonomes, typiquement pour éviter qu'un lien qui bagotte force tous les routeurs du réseau à recalculer leurs routes ; un routeur qui relie deux zones entre elles peut agréger ses routes pour cacher la topologie interne de chaque zone à sa voisine. Dans le cas de notre petit réseau tout à plat, nous n'utiliserons que la zone 0.0.0.0.

D'autre part, chaque routeur dispose d'un identifiant unique sur le réseau, là encore un entier sur 32 bits noté comme une adresse IPv4. On utilise traditionnellement une des adresses IP du routeur comme identifiant, ce qui permet d'assurer cette unicité (ou presque, après tout, rien n'interdit de monter plusieurs fois la même adresse IP sur des réseaux différents).

4 BIRD

Il existe au moins quatre implémentations libres d'OSPF, à savoir Quagga, BIRD, XORP et OpenOSPFD. Mon choix s'est porté sur BIRD, qui est à la fois léger, stable et sans surprises pour un admin système. Ce que je veux dire avec ce dernier critère, c'est que BIRD se configure comme un démon Unix et pas comme un routeur : on édite un fichier de config, on envoie le signal **SIGHUP** au démon **bird** et ce dernier relit le fichier.

Debian distribue BIRD et une version presque à jour est disponible dans les backports ; c'est celle-ci que nous allons installer. Si votre système ne fournit pas de paquet BIRD, ou si vous voulez absolument installer la dernière version, vous pouvez aller la chercher sur le site de référence (http://bird.network.cz/) ; ici, nous nous contenterons du paquet Debian.

```
root@routeur:~# apt-get install -t wheezy-
backports bird
```

Deux petites observations. Tout d'abord, si votre **apt-get** est configuré pour installer les paquets recommandés (ce qu'il fait par défaut sous Debian Wheezy), il va aussi vous installer un paquet **bird6** ; il s'agit du pendant IPv6 de BIRD, dont nous parlerons vers la fin de l'article. D'autre part, Debian étant toujours Debian, le démon **bird** est lancé sitôt après l'installation ; toutefois, sa configuration par défaut fait très exactement rien. Nous allons donc nous empresser de la remplacer par la nôtre.

4.1. Configuration du routeur

Le démon **bird** se configure dans le fichier **/etc/bird.conf** (encore une fois, ça peut varier d'un système d'exploitation à l'autre). Voici le contenu de ce fichier sur **routeur** :

```
router id 10.0.5.1;

protocol device {
  scan time 10;
}

protocol kernel {
  learn;
  persist;
  scan time 1;
  import all;
  export all;
}

protocol ospf PubIP {
  area 0.0.0.0 {
    interface "eth0" { };
  };
  import all;
  export none;
}
```

Nous avons déjà vu la signification de la première ligne, c'est l'identifiant du routeur. Nous avons ensuite trois blocs qui servent à configurer trois protocoles de routage. Pourquoi trois, alors que jusqu'ici je n'ai parlé que d'OSPF ? Voyons ça plus en détails.

Le protocole **device** sert à faire l'inventaire des interfaces réseau de la machine. Ici, nous le configurons pour rafraîchir cette liste toutes les 10 secondes ; sous Linux, le noyau est de toute façon capable d'aviser BIRD des changements en temps réel, cet inventaire périodique sert juste au cas où un message du noyau se perdrait en route. Sur nos machines à la configuration réseau à peu près statique, on peut sans risque choisir un intervalle de rafraîchissement très élevé.

Le protocole **kernel** permet à BIRD de synchroniser sa table de routage avec celle du noyau. Nous avons déjà vu la signification de l'option **scan time**. L'option **learn** indique à BIRD qu'il doit tenir compte des routes de toute origine (par exemple, celles qui auraient été établies par un autre démon de routage) ; quant à l'option **persist**, elle indique à BIRD de laisser ses routes en place lorsqu'il s'arrête au lieu de nettoyer la table de routage. Enfin, les options **import** et **export** permettent de contrôler, respectivement, la liste des routes que BIRD apprend du noyau et celles qu'il lui communique. Dans le cas présent, on choisit de communiquer toutes les routes dans les deux sens ; on peut remplacer le mot-clé **all** par un filtre, je vous laisse consulter la documentation de BIRD pour vous faire une idée des possibilités.

Une petite note au passage : si nous n'avons ici qu'une seule instance du protocole **kernel**, on peut tout à fait en définir plusieurs, qui correspondront à différentes tables de routage du noyau. Cette possibilité n'existe bien entendu que sur les systèmes dont le noyau supporte plusieurs tables de routage, ce qui est le cas de Linux.

Enfin, nous entrons dans le vif du sujet avec le protocole **ospf**. Nous pouvons en définir plusieurs instances ; ici, nous nous contenterons d'une seule, identifiée par l'étiquette **PubIP**. Nous définissons une seule zone OSPF, la dorsale 0.0.0.0, à laquelle nous raccordons l'interface **eth0**. Nous connaissons déjà les

options **import** et **export** ; ici, l'idée est que le routeur écoute les annonces des serveurs, mais n'annonce rien lui-même, puisque les routes par défaut des serveurs lui enverront de toute façon tout le trafic.

4.2. Configuration des serveurs

La configuration des serveurs est très proche de celle du routeur ; examinons rapidement les différences. Je passe sur l'identifiant ; nous prendrons les adresses IP privées des machines, histoire de ne pas devoir en changer le jour où on déplacera les adresses publiques. La configuration des protocoles **device** et **kernel** est identique ; reste à voir le protocole **ospf**.

```
protocol ospf PubIP {
  area 0.0.0.0 {
    interface "eth0" { };
    interface "dummy0" { stub yes; };
  };
  import all;
  export all;
}
```

La directive **export** prend la valeur **all** : on veut annoncer notre adresse publique, on doit donc exporter nos routes en OSPF. Notez que la directive **import** reste à **all** ; en effet, un de nos objectifs est que chaque serveur puisse apprendre les routes vers les adresses publiques de ses voisins. Et puisqu'il est question d'exporter des routes, on configure notre instance d'**ospf** pour ajouter l'interface **dummy0** à la dorsale, afin d'y annoncer les adresses configurées pour cette interface. Le drapeau **stub** indique qu'aucun trafic OSPF ne passera sur cette interface, on se contente de l'annoncer sur le reste du réseau.

Une fois **bird** configuré et lancé, on peut accéder à son interface de contrôle via la commande **birdc**. Une fois dans l'interface de contrôle, la touche **?** permet d'accéder à l'aide en ligne, je vous laisse la découvrir par vous-même. Dans notre cas, on peut au moins s'en servir pour examiner les relations d'adjacence ; si les démons tournent depuis un petit moment (plus de quelques secondes, le temps que les instances OSPF négocient entre elles), le résultat devrait ressembler à ça :

```
root@serveur2:~# birdc
BIRD 1.3.10 ready.
bird> sh ospf neighbors
PubIP:
Router ID  Pri    State    DTime  Interface  Router IP
10.0.5.1     1   full/bdr  00:39  eth0       10.0.5.1
10.0.5.11    1   full/dr   00:39  eth0       10.0.5.11
```

Notez les états **full/dr** et **full/bdr**, qui indiquent respectivement le DR et le BDR ; les autres routeurs OSPF du réseau sont normalement dans l'état **full/other**.

On peut vérifier que chaque serveur a bien une route vers l'IP publique de son voisin :

```
root@serveur2:~# ip r sh
default via 10.0.5.1 dev eth0
10.0.5.0/24 dev eth0  proto kernel  scope link  src 10.0.5.12
192.0.2.72 via 10.0.5.11 dev eth0  proto bird
```

Et sur le routeur :

```
root@routeur:~# ip r sh dev eth0
10.0.5.0/24  proto kernel  scope link  src 10.0.5.1
192.0.2.72 via 10.0.5.11  proto bird
192.0.2.73 via 10.0.5.12  proto bird
```

Mission accomplie ! Avant de terminer cet article, je voudrais juste vous donner quelques pistes pour aller un peu plus loin.

5 Pour aller plus loin

5.1. Un peu de haute dispo

Et si je monte la même adresse IP sur les deux serveurs, il se passe quoi ? Essayons pour voir, en ajoutant l'adresse 192.0.2.72 sur l'interface **dummy0** de **serveur2**.

```
root@routeur:~# ip r sh dev eth0
10.0.5.0/24  proto kernel  scope link  src 10.0.5.1
192.0.2.72 via 10.0.5.12  proto bird
192.0.2.73 via 10.0.5.12  proto bird
```

Arrêtons maintenant **serveur2** brutalement ; après une trentaine ou une quarantaine de secondes le temps qu'OSPF converge, on retrouve notre route vers **serveur1**.

```
root@routeur:~# ip r sh dev eth0
10.0.5.0/24  proto kernel  scope link  src 10.0.5.1
192.0.2.72 via 10.0.5.11  proto bird
```

Dans cette configuration, on a une forme limitée de haute disponibilité. Les possibilités sont moindres qu'avec des outils spécialisés genre **keepalived**, mais une fois notre routage OSPF en place, c'est gratuit.

5.2. Et la sécurité dans tout ça ?

Le protocole OSPF, tel que défini dans la RFC 2328, permet d'authentifier les paquets OSPF au moyen d'une empreinte MD5. L'empreinte en question est générée à partir du paquet lui-même et d'une clé partagée par les routeurs OSPF adjacents. La RFC 5709 prévoit l'utilisation des algorithmes de la famille SHA à la place de MD5, mais à ma connaissance, cette extension n'est pas implantée dans BIRD.

Pour activer ce mode d'authentification, il suffit d'indiquer la clé partagée dans la définition de l'interface concernée dans le fichier **bird.conf** ; l'instance OSPF utilisera alors cette clé partagée pour signer tout le trafic OSPF sur cette interface, et refusera les paquets qui ne sont pas signés avec la même clé. Notez que cette signature n'interdit pas à un attaquant d'écouter le trafic OSPF, elle empêche juste l'injection de routes pirates.

```
interface "eth0" {
  authentication cryptographic;
  password "maclepartageequejai";
};
```

BIRD permet d'affiner un peu cette configuration, notamment de prévoir un changement de mot de passe à une date donnée ; je vous laisse lire la documentation pour en savoir plus.

5.3. Et IPv6 alors ?

C'est vrai, quoi, on est au XXI[ème] siècle et pas un mot sur IPv6 ? C'est simplement pour une raison pratique : les adresses IPv6 sont tellement abondantes que je n'ai jamais rencontré le problème de base de l'article (à savoir « *j'ai moins d'adresses publiques que de machines* »).

BIRD est parfaitement utilisable en IPv6. Par contre, le même binaire **bird** parle IPv4 ou IPv6, mais pas les deux. Le choix se fait à la compilation. La plupart des distributions fournissent donc deux binaires, **bird** pour IPv4 et **bird6** pour IPv6 (Debian comporte deux paquets séparés, **bird** et **bird6**).

La configuration de **bird6** est très similaire à ce qu'on a vu en IPv4 ; il faut juste faire attention à deux petites choses.

D'une part, j'insiste sur le fait que les numéros de zones et les identifiants de routeurs sont des entiers sur 32 bits et pas des adresses IP, la notation ne doit pas vous induire en erreur. Ils restent exprimés sur 32 bits même en IPv6. Ça a un petit effet de bord en pratique : si en IPv4 BIRD est capable de générer un identifiant de routeur lui-même (il utilise la plus petite adresse IPv4 portée par une interface autre que la boucle locale), en IPv6 l'option **router id** est obligatoire.

D'autre part, l'authentification cryptographique n'existe pas en IPv6. Mais comment donc, pas de sécurité ? Les concepteurs d'OSPFv3 sont partis du principe qu'une pile IPv6 comportait obligatoirement une implantation d'IPSEC et donc, qu'on n'avait qu'à s'en servir pour chiffrer le trafic OSPF ; non mais c'est vrai, c'est pas parce que ça fait mal à la tête qu'on ne doit pas le faire. La réalité de la vraie vie étant ce qu'elle est (surtout depuis la RFC 6434, qui supprime l'obligation d'implanter IPSEC), j'ai bien peur que la sécurité de beaucoup de déploiements OSPFv3 se limite à du filtrage par adresse d'émetteur. ■

Références

- Algorithme de Dijkstra : http://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra
- BIRD : http://bird.network.cz/
- OSPFv2 : http://tools.ietf.org/html/rfc2328
- OSPFv3 : http://tools.ietf.org/html/rfc5340
- Authentification cryptographique HMAC-SHA pour OSPFv2 : http://tools.ietf.org/html/rfc5340

ABONNEZ-VOUS !

CONSULTEZ L'ENSEMBLE DE NOS OFFRES SUR : boutique.ed-diamond.com !

11 Numéros de GNU/Linux Magazine

60€*

au lieu de 86,90 €* en kiosque

Économie 26,90€

Économisez plus de 30%*

* Sur le prix de vente unitaire France Métropolitaine

Nouveauté 2014
TOUS LES HORS-SÉRIES PASSENT EN GUIDE !

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

Découvrez tous les Guides déjà parus sur

boutique.ed-diamond.com

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

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

boutique.ed-diamond.com

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

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

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

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

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

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

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

GNU LINUX MAGAZINE / FRANCE

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

Vos remarques :

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

Tournez SVP pour découvrir toutes les offres d'abonnement

ABONNEMENT GLMF

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

offre LM

INCLUS : GNU/Linux Magazine (11nos)

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

Économie 26,90€

offre LM+

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

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

Économie 49,30€

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

offre T

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

Économie 79,10€

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

offre T+

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

Économie 112,20€

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

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

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

boutique.ed-diamond.com

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

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

➡ Nos Tarifs s'entendent TTC et en euros

		F	OM1	OM2	E	RM
		France Métro.	Outre-Mer		Europe	Reste du Monde
LM	Abonnement GLMF	60 €	75 €	96 €	83 €	90 €
LM+	Abonnement GLMF + GLMF HS (6 Guides)	115 €	147 €	190 €	160 €	173 €
T	Abonnement GLMF + MISC + OS + LP + LE	198 €	253 €	325 €	276 €	300 €
T+	Abonnement GLMF + GLMF HS (6 Guides) + MISC + MISC HS + OS + LP + LP HS (3 Guides) + LE	299 €	382 €	491 €	415 €	448 €

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

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

MA FORMULE D'ABONNEMENT :

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

J'indique la somme due : (Total) ..... €

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

Je choisis de régler par :

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

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

Date et signature obligatoires

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

par Guillaume Saupin, Phd

Dans notre précédent article, nous avons vu comment créer simplement un applicatif Android en Python. Nous allons maintenant détailler comment cross compiler simplement du C++ vers la plateforme Android pour booster les performances.

1 Introduction

1.1 Optimiser les sections critiques de code

Dans tout applicatif, il y a des sections de code critiques, dont la complexité algorithmique, ou la dimension des données à traiter nécessitent une implémentation efficace. Dans ce cas, il faut coller au plus près à l'architecture de la machine, en optimisant les accès au cache, en utilisant des instructions vectorielles, en réduisant la taille des structures de données, ... Le C/C++ s'impose alors pour obtenir les meilleurs résultats.

Il n'est pas nécessaire, cependant, d'écrire toute l'application en C/C++, ce qui peut être fastidieux. Une simple réécriture locale des quelques fonctions les plus gourmandes en C/C++ suffit. Reste alors à établir un lien entre C/C++ et le langage de l'application pour pouvoir invoquer le code optimisé depuis l'application.

1.2 Cross compiler du C/C++ pour Android

Dans le cadre du langage Python, une solution est offerte par Cython. Dans cet article, nous allons donc utiliser Cython, les `setuptools`, et `python-for-android`, pour optimiser une section de code critique de notre application exemple.

2 PitchDetector

Notre application exemple est un détecteur de note simplissime, dont l'interface graphique sera particulièrement épurée. Son fonctionnement repose sur l'utilisation du micro de votre appareil Android, ce qui va nous permettre de voir comment Kivy permet l'accès au hardware.

Vous retrouverez l'ensemble du code de cette application en ligne sur le repository git suivant : https://github.com/kayhman/kivyLM.

2.1 Détection de la note

2.1.1 Les données en entrée

Les appareils Android nous permettent, comme la quasi totalité des appareils numériques offrant des fonctionnalités audio, de récupérer un signal sonore échantillonné à une fréquence de 44100Hz. C'est-à-dire que l'intensité sonore mesurée par le micro de votre appareil est stockée 44100 fois par seconde.

Les données que l'on récupère se présentent donc sous la forme d'un tableau de valeurs d'intensité sonore. Ces données peuvent être encodées de manière plus ou moins précise. Par exemple, dans le format WAV, elles peuvent être encodées sur 8, 16, ou 24 bits. Dans le cas des échantillons fournis par Android, la possibilité nous est offerte de les encoder sur 8 ou 16 bits.

Cependant, seul l'encodage sur 16 bits est garanti de fonctionner sur tous les appareils. Nous choisirons donc ce dernier.

2.1.2 Le principe

Pour détecter la note enregistrée par votre appareil, nous allons utiliser une méthode basique : l'autocorrelation.

L'idée est la suivante : une note a une fréquence principale qui la caractérise. Ainsi, le *fa* a une fréquence f de

440Hz. Si l'on décale ce signal de sa période t = 1/f, et qu'on le compare avec le signal d'origine, alors les deux signaux doivent être très similaires, voire identiques dans le cas d'un son purement sinusoïdal. La valeur quantifiant la comparaison avec le signal d'origine est appelée autocorrelation.

Pour caractériser un son, cette méthode est répétée avec plusieurs périodes différentes. La période qui génère la plus grande valeur d'autocorrelation donne la fréquence probable du signal.

Il n'est bien sûr pas possible d'appliquer cette méthode avec toutes les fréquences audibles, leur nombre étant infini. Nous allons donc nous contenter d'un sous-ensemble discret et fini en prenant simplement les 88 notes d'un piano. Nous utiliserons pour cela la formule suivante, qui donne la fréquence d'une touche :

```
def keyFreq(key):
   return  pow(2, (key-49.0) / 12.0) * 440.0
```

Nous retrouverons cette fonction plus tard dans notre exemple Python. La variable **key** correspond à une touche de piano, et prend comme valeur un entier entre 1 et 88.

Arrêtons-nous un instant pour analyser cette fonction, et voir ce qu'elle, et donc finalement la physique d'un son, implique sur la conception de notre programme.

Tout d'abord, la quarante-neuvième touche étant le *la*, on retombe bien sur la fameuse fréquence de 440Hz. Ensuite, sachant que la première note, la plus grave, a pour indice de touche 1, il s'ensuit que la fréquence la plus basse est de 27.5Hz. De même, la fréquence la plus haute, pour la touche d'indice 88, est de 3520Hz.

Cela implique donc que la période la plus grande à laquelle nous aurons affaire sera atteinte pour la note la plus grave, et vaudra 1/27.5 Hz, soit 0.0363 seconde. Notre échantillonnage étant effectué à 44000Hz, cela représente donc un enregistrement de 1/27.5 * 44 100 = 1600 échantillons.

Il faut encore doubler ce chiffre, puisque la méthode de l'autocorrelation implique de comparer un signal à lui-même, avec un décalage d'une période t. Il nous faudra donc travailler sur une base de 3200 échantillons pour pouvoir traiter la fréquence la plus basse.

Par conséquent, il nous faudra faire nos calculs en moins de deux fois la période max de 1/27.5 Hz pour rester en « temps réel », c'est-à-dire pour ne pas avoir trop de décalage entre le son réel et le résultat de notre calcul. Concrètement, nous avons à peu près 72 millisecondes. Par contre, nous aurons nécessairement un décalage minimal de 72 millisecondes, puisqu'il nous faudra bien attendre d'avoir au moins enregistré ces 72 ms de signal.

2.1.3 Le code Python

L'implémentation en Python est immédiate :

```
def computeCorrelation(samples, size, offset):
   dist = 0.0
   for idx in range(0, size):
       s  = float(samples[2*idx] + 255 * samples[2*idx+1] )
       st = float(samples[int(2*(idx))%size] + 255 * samples[int(2*(idx+offset)+1)%size])
       dist += s * st
   return dist
```

Notre fonction prend en entrée un tableau contenant les échantillons de notre signal sonore. Les données étant présentées sous la forme d'un tableau d'octets, et le signal étant encodé en 16 bits, chaque échantillon est recalculé et stocké dans la variable **s**. La même opération est effectuée pour l'échantillon décalé de la période **t**, c'est-à-dire décalé de **offset** = **t** * 44 100.

Ces deux échantillons **s** et **st** sont multipliés et additionnés à **dist**.

En itérant sur tous les échantillons de notre buffer, nous obtenons dans **dist** une estimation de l'autocorrelation. Cette valeur, comme indiqué dans le paragraphe ci-dessus, est d'autant plus grande que la la période **t** est proche de la période propre du signal.

En appliquant cette procédure à différentes périodes **t**, on obtient un ensemble de valeurs d'autocorrelation, dont la plus grande donne la période la plus probable du signal. C'est ce que fait la méthode **slowPitchDetection** :

```
def slowPitchDetection(samples, size, Athres):
   dist = 0
   note = 0
   freq = keyFreq(note)
   maxDist = Athres
   offset = round(44100.0 / freq)


   ct0 = float(computeCorrelation(samples, size, 0))
   dist = computeCorrelation(samples, size, offset) / ct0
   for key in range(27, 52):
       freq = keyFreq(key)
       offset = round(44100.0 / freq)
       dist = computeCorrelation(samples, size, offset) / ct0
       if dist > maxDist:
           maxDist = dist
           note = key
  return note, keyName[note]
```

2.1.4 Le code C/C++

Le code C/C++ est tout à fait similaire. Le calcul de l'autocorrelation pour une période donnée s'effectue comme suit :

```
double computeCorrelation(const short* samples, const int size, const
int offset)
{
   double dist = 0.;
   int idx = 0;

   for(idx = 0 ; idx < size ; idx++)
       dist += samples[idx] * samples[(idx+offset)%size];
   return dist;
}
```

La différence principale avec le code Python réside dans l'utilisation d'un tableau de **short int**, donc d'entiers sur 16 bits. Il n'est donc plus nécessaire d'effectuer la conversion entre couple d'octets et valeur sur 16 bits. Le reste est identique.

La détection de la note la plus probable se fait de la même manière, en itérant sur toutes les notes, et en ne considérant que la note du signal donnée pour la plus haute valeur d'autocorrelation :

```
int autoCorrelationOpt(const char* samples, const int size,
const double Athres)
{
   short int* ptr = (short int*)samples;
   double dist = 0.;
   double freq = 0.;
   double maxDist = 0.;
   int offset = 0;
   int note = 0;
   freq = keyFreq(note);
   int key = 0;

   const double ct0 = computeCorrelation(ptr, size, 0);

   offset = round(44100.0 / freq);
   dist = computeCorrelation(ptr, size, offset) / ct0;
   maxDist = Athres;
   for(key = 27 ;  key <= 52 ; key++)
   {
      freq = keyFreq(key);
      offset = round(44100.0 / freq);
      dist = computeCorrelation(ptr, size, offset) / ct0;
      if( dist > maxDist)
         {
         maxDist = dist;
         note = key;
         }
   }
   return note;
}
```

La fonction suivante est utilisée pour calculer la fréquence d'une note :

```
double keyFreq(int key)
{
   return pow(2, (key-49.0) / 12.0) * 440.0;
}
```

Toutes ces fonctions sont placées dans le fichier **autocorrelationOpt.c**.

3 Cython

Nous avons présenté dans la section précédente le concept d'autocorrelation, et écrit les quelques lignes de code Python et C++ mettant en musique ce concept. Après la lecture du précédent article sur Kivy, vous êtes en mesure d'intégrer directement et facilement le code Python à votre application Android basée sur Kivy.

Par contre, la chose paraît moins immédiate pour le code C/C++. Il nous faut un moyen de coupler facilement ce code au code Python de notre application Android, en espérant ainsi améliorer les performances de notre logiciel.

Nous allons pour cela utiliser Cython. C'est un outil qui propose principalement deux fonctions :

- Compiler du code Python en l'optimisant vers du C/C++,
- Interfacer facilement du C/C++ avec Python.

C'est la deuxième fonctionnalité que nous allons exploiter dans notre cas, en permettant l'appel de méthode C/C++ depuis le Python.

Kivy utilise déjà en interne Cython pour optimiser un partie de son code, mais aussi pour accéder à du code C/C++, notamment du code OpenGL.

3.1 Lien Python / C/C++

Pour générer la glu entre le code C/C++ et Python, Cython utilise un fichier **.pyx**. Classiquement, ce fichier sert à écrire du code en Python, qui sera converti au mieux en C/C++. Cette conversion permet en théorie d'accroître les performances du code. Cependant, comme bien souvent dans le cas de la conversion automatique d'un langage dans un autre, le code généré n'est pas optimal, et le gain obtenu ne l'est pas non plus. C'est pour cette raison que nous avons écrit notre propre code C/C++.

Fort heureusement, les fichiers **.pyx** permettent d'appeler des méthodes C/C++ définies par ailleurs. Soit dans notre cas :

```
#fichier autocorrelation.pyx

cdef extern int autoCorrelationOpt(const char*
samples, int size, const double Athres)
cdef extern char* getKeyName(int key)

def autoCorrelation(const char* samples, int size,
const double Athres):
   key = autoCorrelationOpt(samples, size, Athres)
   return key, getKeyName(key)
```

Ce fichier **.pyx** déclare donc l'existence de deux fonctions C/C++ : **autoCorrelationOpt** et **getKeyName**, qui sont réutilisées très simplement dans la fonction Python **autoCorrelation**. C'est à cette dernière fonction que nous accéderons dans le code Python de notre application.

3.2 Création du package

Nous n'allons pas utiliser Cython directement, mais à travers les **setuptools** de Python, qui vont nous permettre de construire facilement un package que nous pourrons réutiliser facilement avec Kivy.

Le fonctionnement des **setuptools** repose sur l'écriture d'un fichier **setup.py**, qui va définir les propriétés du package que l'on souhaite créer. Dans notre cas, nous allons procéder en deux étapes :

1. Génération du code C/C++ depuis le fichier **autocorrelation.pyx**. C'est la responsabilité du fichier **setupBuild.py** ;
2. Création du package. Le fichier **setup.py** s'en charge.

Le fichier **setupBuild.py** s'écrit comme suit :

```
from distutils.core import setup
from Cython.Build import cythonize

setup(name='autocorrelation',
     version='1.0',
     ext_modules = cythonize("autocorrelation/
autocorrelation.pyx")
     )
```

Il importe notamment le module Cython, et définit le module **autocorrelation** basé sur **autocorrelation.pyx**. Nous allons utiliser ce fichier de manière détournée pour générer le fichier **autocorrelation.c**, qui contiendra le code glu liant notre code optimisé et Python.

Setup.py empaquette ce fichier au côté de **autocorrelationOpt.c**, qui contient le code optimisé de la fonction d'autocorrelation.

```
from distutils.core import setup
from distutils.extension import Extension

setup(name='autocorrelation',
```

```
    version='1.1',
    ext_modules =  [Extension("autocorrelation", ["autocorrelation/
autocorrelation.c", "autocorrelation/autocorrelationOpt.c"])]
    )
```

Notre package s'appellera donc **autocorrelation**, portera la version 1.1, et est compressé dans le fichier **autocorrelation-1.1.tar.gz**.

3.3 Déploiement du package sous Android

Il ne nous reste plus qu'à rendre ce module accessible sous Android. Pour cela, nous allons utiliser le mécanisme de *recipes* de Kivy. Ce système, basé sur des scripts bash, repose sur l'utilisation d'un serveur web pour télécharger le package Python que nous venons de créer.

Voici un exemple de fichier recipe permettant d'installer notre package, dans l'hypothèse où il serait stocké localement, et accessible à travers un serveur HTTP. Ce fichier doit être placé dans le répertoire **pyton-for-android/recipes/autocorrelation/recipe.sh**.

```
#!/bin/bash
# python-for-android/recipes/autocorrelation/recipe.sh
VERSION_autocorrelation=1.1
URL_autocorrelation=http://localhost/download/autocorrelation-$(echo $VERSION_
autocorrelation).tar.gz
DEPS_autocorrelation=(python)
MD5_autocorrelation=85a7dc45b86716e0aa642381835bbc9b
BUILD_autocorrelation=$BUILD_PATH/autocorrelation/$(get_directory $URL_autocorrelation)
RECIPE_autocorrelation=$RECIPES_PATH/autocorrelation

function prebuild_autocorrelation() {
  true
}

function build_autocorrelation() {
  cd $BUILD_autocorrelation

  push_arm

  try $BUILD_PATH/python-install/bin/python.host setup.py install -O2

  pop_arm
}

function postbuild_autocorrelation() {
  true
}
```

Son fonctionnement est simple. Il indique simplement où télécharger l'archive du package. L'installation de ce dernier se fait ensuite à l'aide de la fonction **build_autocorrelation()** qui se contente d'invoquer Python sur le fichier d'installation **setup.py**.

Reste alors à créer la distribution Kivy incluant ce package. Rien de plus simple, comme nous l'avons vu dans le précédent article. Rendez-vous dans le répertoire où vous avez installé **python-for-android**, et lancez :

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

Notez l'ajout du package **pyjnius** qui va nous permettre d'accéder aux fonctionnalités audio.

4 Lancement de l'application et gain en performance

4.1 Mise en musique dans Kivy

Nous avons désormais une distribution Kivy intégrant notre module d'autocorrelation. Nous allons donc pouvoir écrire une petite application Android très basique, qui va nous permettre de comparer les temps d'exécution de nos deux codes.

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

from kivy.lang import Builder
from kivy.uix.floatlayout import FloatLayout
from kivy.uix.label import Label
from kivy.app import App
from kivy.clock import Clock
from kivy.properties import ObjectProperty

from jnius import autoclass

import autocorrelation
import random
from time import time

keyName = ["Not found", "A0", "A0#", "B0",
        "C1", "C1#", "D1", "D1#", "E1", "F1", "F1#", "G1", "G1#", "A1", "A1#", "B1",
        "C2", "C2#", "D2", "D2#", "E2", "F2", "F2#", "G2", "G2#", "A2", "A2#", "B2",
        "C3", "C3#", "D3", "D3#", "E3", "F3", "F3#", "G3", "G3#", "A3", "A3#", "B3",
        "C4", "C4#", "D4", "D4#", "E4", "F4", "F4#", "G4", "G4#", "A4", "A4#", "B4",
        "C5", "C5#", "D5", "D5#", "E5", "F5", "F5#", "G5", "G5#", "A5", "A5#", "B5",
        "C6", "C6#", "D6", "D6#", "E6", "F6", "F6#", "G6", "G6#", "A6", "A6#", "B6",
        "C7", "C7#", "D7", "D7#", "E7", "F7", "F7#", "G7", "G7#", "A7", "A7#", "B7",
        "C8"]

def computeCorrelation(samples, size, offset):
   dist = 0.0
   for idx in range(0, size):
       s  = float(samples[2*idx] + 255 * samples[2*idx+1] )
       st = float(samples[int(2*(idx))%size]
            + 255 * samples[int(2*(idx+offset)+1)%size])
       dist += s * st
   return dist

def keyFreq(key):
   return  pow(2, (key-49.0) / 12.0) * 440.0

def slowPitchDetection(samples, size, Athres):
   dist = 0
   note = 0
   freq = keyFreq(note)
   maxDist = Athres
   offset = round(44100.0 / freq)


   ct0 = float(computeCorrelation(samples, size, 0))
   dist = computeCorrelation(samples, size, offset) / ct0
   for key in range(27, 53):
       freq = keyFreq(key)
       offset = round(44100.0 / freq)
       dist = computeCorrelation(samples, size, offset) / ct0
       if dist > maxDist:
           maxDist = dist
           note = key
```

```
   return note, keyName[note]


class PitchDetector(FloatLayout):
   pitchLabel = ObjectProperty()
   pitchLabelPython = ObjectProperty()
   toggleButton = ObjectProperty()

   AudioRecord = autoclass('android.media.AudioRecord')
   Buffer  = autoclass('java.nio.ByteBuffer')
   AudioFormat = autoclass('android.media.AudioFormat')
   AudioSource = autoclass('android.media.MediaRecorder$AudioSource')
   buff = Buffer.allocateDirect(3400*2)
   bufferSize = AudioRecord.getMinBufferSize(44100, AudioFormat.CHANNEL_
CONFIGURATION_MONO, AudioFormat.ENCODING_PCM_16BIT);
   mAudio = AudioRecord(AudioSource.MIC, 44100, AudioFormat.CHANNEL_CONFIGURATION_
MONO, AudioFormat.ENCODING_PCM_16BIT, bufferSize)
   enablePythonCode = True

   def build(self):
     self.mAudio.startRecording()

   def startRecord(self):
     Clock.schedule_interval(self.record, .500)

   def stopRecord(self):
     Clock.unschedule(self.record)

   def togglePython(self):
     self.enablePythonCode = not self.enablePythonCode
     if self.enablePythonCode:
  self.toggleButton.text = "Disable Python"
     else:
  self.toggleButton.text = "Enable Python"
   def record(self, dt):
     threshold = 0.8
     # create out recorder
     read = self.mAudio.read(self.buff, 3400)
     data = self.buff.array()

     startTime = time()
     strg = ""
     for d in data:
          strg += "%c"%d

    # Call C code
     self.noteVal, self.note = autocorrelation.autoCorrelation(strg, 3400, threshold)
     last = (time() - startTime)
     if self.note != "Not found":
        self.pitchLabel.text = "%d Samples Recorded : %s in %f"%(read, self.note, last)

     # Call Python code
     startTime = time()
     self.noteVal, self.note = slowPitchDetection(data, 3400, threshold)
     last = (time() - startTime)
     if self.enablePythonCode:
         if self.note != "Not found":
            self.pitchLabelPython.text = "%d Samples Recorded : %s in %f"%(read,
self.note, last)
     else:
         self.pitchLabelPython.text = "Python code disabled"

class PitchDetectorApp(App):
   def build(self):
       self.pitch = PitchDetector(name="pitch")
       self.pitch.build()
       return self.pitch

PitchDetectorApp().run()
```

Le code est délibérément simple. On y retrouve le code Python présenté au début de l'article et calculant l'autocorrelation de notre signal sonore. Le code C/C++ est pour sa part appelé en incluant le module **autocorrelation**, et en appelant tout bêtement la fonction **autoCorrelation**.

L'accès au micro sera fait à l'aide des classes **audioRecord** et **mediaRecorder** qui sont « bindés » automatiquement depuis Java par **pyjnius**.

L'interface est pour sa part décrite à l'aide du langage **kv** :

```
#:kivy 1.0

<PitchDetector>:
   pitchLabel: pitchLabel
   pitchLabelPython: pitchLabelPython
   toggleButton: toggleButton
   BoxLayout:
       padding: 5
       spacing: 5
       orientation: 'vertical'
       Label:
           id: pitchLabel
           text: 'Pitch detected(C) : ?'
       Label:
           id: pitchLabelPython
           text: 'Pitch detected (Python) : ?'
       BoxLayout:
           orientation: 'horizontal'
           Button:
               on_press: root.startRecord()
               text: "Start"
           Button:
               on_press: root.stopRecord()
               text: "Stop"
       BoxLayout:
           orientation: 'horizontal'
           Button:
               id: toggleButton
               on_press: root.togglePython()
               text: "Disable Python"
```

La génération du **.apk** contenant notre petite application se fait alors avec :

```
# ./build.py --package org.MyCompany.KillerApp --name pitcDetector
--version 1.1 --dir ~/projects/Kivy/pitchDetector/ debug
```

Ce dernier s'installera alors avec :

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

Conclusion

Vous verrez à l'exécution de l'application que le gain de performances est substantiel, dans la mesure où le code C/C++ s'exécute environ dix fois plus rapidement que le code 100% Python. Le duo Kivy+Cython offre donc un moyen simple et efficace pour optimiser des sections de code et rendre nos applications Android plus performantes, et donc plus attrayantes pour les utilisateurs. ■

LES MATHS NE SERVENT À RIEN EN INFORMATIQUE !

par Dr Kiss Cool [Attention au deuxième effet...]

Ah les mathématiques ! Nous y avons tous été confrontés un jour ou l'autre pendant notre scolarité... Mais est-ce que ça vous sert pour écrire des programmes ?

« Une baignoire se remplit à ras bord en 10 minutes avec l'eau du robinet, la bonde étant fermée. Pour la vider entièrement, le robinet étant fermé, il faudra 20 minutes. Combien faudra-t-il de temps pour la remplir au maximum si l'on ne ferme pas la bonde ? ».

Ça y est, vous commencez à transpirer, à respirer difficilement, votre vue se trouble... Mais non, tout va bien, on ne vous demandera pas de résoudre ce problème. En effet, quelle relation y a-t-il entre les mathématiques et l'informatique ? Est-ce que, parce que je ne sais pas résoudre un problème simple, je ne serai pas capable de créer un site Internet ? Les études en informatique sont ouvertes aux matheux : il faudrait forcément avoir des connaissances mathématiques pour pouvoir programmer ? Foutaises ! Je m'en vais vous le démontrer dans cet article.

1 Les jeux vidéo

Voilà un domaine qui fait rêver beaucoup d'adolescents (et aussi des plus grands). A-t-on besoin des maths pour jouer à Doom ou à Day of the Tentacle ? Ah... notre bien-aimé rédacteur en chef vient de me signaler par oreillette qu'il faudrait que je mette à jour mes références. Oui, à *GNU/Linux Magazine*, tous les auteurs sont contrôlés en temps réel par Denis Bodor. Je vous en dirai d'ailleurs un peu plus si on me le permet, car... Aïeuuu ! J'avais oublié les électrochocs... Une saine thérapie pour ramener les auteurs à « une totale liberté de pensée cosmique vers un nouvel âge réminiscent ». Je vais donc traduire mes propos pour la jeune génération : les maths ça sert à que dalle pour avoir le swag dans GTA V !

Effectivement, à part pour résoudre les énigmes de certains jeux de réflexion, inutile de savoir résoudre une équation pour jouer à un jeu de course, d'aventure, de stratégie ou autre ! Donc, si on n'a pas besoin de ça pour être un « killer » en jeux vidéo, eh bien on va pouvoir en créer facilement ! En reprenant l'exemple des ados, fions-nous au bon sens populaire. Que vont dire les parents en voyant leur petit génie de fils être capable de passer une commande sur Internet, de copier une page de Wikipédia pour rendre un devoir d'histoire, d'écrire des messages sur Facebook à la vitesse de la lumière, ou même de faire un montage vidéo ? Eh bien simplement que c'est un informaticien, et ils auront bien raison ! Et si, en plus, celui-ci a terminé cinq fois GTA en bastonnant pas moins de 2000 vieilles pour leur prendre leur fric, comment croire le prof de maths, ce demeuré, qui dit que le petit génie ne pourra pas s'engager dans des études en informatiques car il n'est pas assez bon en maths ?

Décortiquons grossièrement ce qu'il faut savoir pour faire un jeu et parvenir au statut envié de « Développeur de jeux vidéo » :

- Pour la 3D, on nous parle de « matrices », « transformations géométriques », « homothéties » et autres... Tout cela pour quoi ? Pour nous embrouiller ! Il y a plein de moteurs 3D tout prêts, inutile de savoir comment ça se passe dedans, on lui dit « dessine-moi un 4x4 » et on obtient le véhicule en 3D.
- Pour le comportement physique des objets, on nous parle de « plans », d'équations diverses (« Euler », « Navier-Stokes »), etc. Là encore, un moteur tout prêt fera l'affaire. Nous faisons entièrement confiance aux informaticiens qui l'ont développé. D'ailleurs... puisqu'eux ils maîtrisent ces équations, ce ne sont pas des informaticiens ! Sans doute des matheux inutiles que l'on a cherché à caser quelque part.
- Pour les jeux d'arcade, de plateforme, la position des éléments est déterminée par des « coordonnées » dans un « repère cartésien ». Mais ils le font exprès ou quoi ? Là aussi, on se débrouillera avec un machin tout prêt. Peut-être même avec une interface graphique, ça nous permettra de ne pas écrire trop de code : l'informaticien moderne sait utiliser les outils qui sont à sa disposition, il fait tout à la souris et en mode graphique !
- Il nous reste les jeux de stratégie. Là, enfin, plus question de mathématiques ; on aura plutôt besoin de créer des intelligences artificielles, de logique et d'algorithmique, des termes bien issus de l'informatique ! Ah, Denis me dit dans mon oreillette que tous ces domaines sont basés sur des maths et qu'en logique on utilise par exemple « l'algèbre de Boole »... Ahahah ! Comme il est naïf ! Encore un coup des matheux pour prendre possession de l'informatique. Il se fait rouler dans la farine et... aïeuuuu ! Tyran ! J'en resterai donc là pour ce qui est des jeux vidéo.

Nous avons pu voir qu'il n'était nullement impossible de développer des jeux vidéo sans aucune connaissance mathématique, des moteurs permettant de tout faire à notre place.

2 Les logiciels de gestion

Par « logiciel de gestion », j'entends tout logiciel permettant d'automatiser des processus : gestion des achats, des congés, des comptes en banque, etc. Avant de m'établir en tant qu'éminence scientifique et de prêcher la bonne parole, j'ai pris mon bâton de pèlerin et j'ai rencontré des chefs de projets, des directeurs techniques dénués de tout sens mathématique... Donc, c'est bien la preuve que l'on peut être informaticien et ne rien comprendre aux mathématiques. Bien sûr, les mauvaises langues trouveront toujours des exemples prouvant qu'une application dysfonctionne : une somme de contrôle qui n'a pas la valeur attendue (on la fixe en dur et le problème est réglé), le site des impôts qui associe à chaque impôt un compte bancaire différent sans en avertir l'utilisateur, une liste déroulante où les choix les plus courants ne se trouvent pas en tête de liste mais éparpillés deci delà, etc. On voit bien qu'il n'est pas question ici de mathématiques, mais simplement d'un utilisateur un peu simplet qui s'attendrait à ce que tout le travail soit fait pour lui.

J'admets que pour pouvoir calculer le montant d'un panier lors d'une commande, il faut être capable d'effectuer les quatre opérations élémentaires et que cet enseignement devrait être au programme des licences d'informatique. Mais en dehors de cela, nul besoin de connaissances particulières.

3 Les sites Internet

Ici, mon argument sera imparable et tiendra en quelques lignes de code :

```
<!doctype html>
<html>
    <head>
        <meta charset= "utf-8">
        <title>À mort les maths!</title>
    </head>
    <body>
        <h1>Dr KissCool</h1>
    </body>
</html>
```

Alors messieurs les sceptiques ? Voyez-vous un seul résidu mathématique dans ce code pourtant hautement complexe ? Aujourd'hui Denis abuse de l'oreillette... Il me dit, je cite : « Non mais là tu dis n'importe quoi, le HTML c'est même pas un langage de programmation ! ». Rhaaa, mais c'est incroyable !! En plus d'être à la solde du Grand Serpent et de ne jurer que par Python, le voici converti par les matheux ! Bien, je me calme... Essayons de le convaincre :

Et PHP, c'est bien un langage, PHP ?

Malheureusement, je te le concède...

Comment ça « malheureusement » ? Non mais je vais te me le... aïeuuu ! Ok, ok, ça va. Donc, tu as déjà vu des sites dont la partie serveur était gérée en PHP ? Tu crois qu'il y a des équations dans le code ?

Bah, si les développeurs PHP savaient faire autre chose que du PHP ça se saurait...

Quoi ??? Mais ce sauvage est en train de dire que les développeurs PHP sont incapables de faire des maths !

C'est un peu ce que tu voulais, non ?

Je... Allez, retourne faire tes petits montages électroniques et arrête de polluer mon bel article !

Excusez-moi pour cet intermède indépendant de ma volonté. En tout cas, vous l'aurez compris, inutile de savoir faire ne serait-ce qu'une addition pour pouvoir faire des développements pour le Web.

4 Recherche & développement

Ce domaine est un peu plus complexe à cerner, puisqu'il s'agit par essence d'innover dans des disciplines diverses. Cela peut être au sein d'une interface graphique, dans une application de bio-informatique, d'astronomie, de psychologie ou autre. D'après la littérature, on peut alors être amené à utiliser du « clustering », des « statistiques », ou les techniques 3D vues pour les jeux vidéo. Inutile de développer plus que cela car, vous l'aurez compris, on peut là encore utiliser des librairies toutes prêtes qui feront tout le travail pour nous...

Ah oui ? Et comment tu fais pour appliquer un traitement statistique particulier, une ANOVA par exemple, si tu ne sais pas à quoi ça correspond ?

Mais tu n'as pas des articles à écrire toi ? Ça t'amuse d'intervenir à tout bout de champ dans les articles des autres ? Ton fer à souder va refroidir...

Je pose une simple question.

Eh bien ma réponse sera simple : je cherche sur Google, on me dit qu'il faut appliquer ce traitement et je le fais.

Et ce « on » tu sais qui c'est ? Tu es sûr de ta source ? Tu sais quels sont les paramètres à appliquer à ton traitement ?

Ok, tu as décidé d'être d'une mauvaise foi exemplaire, je laisse tomber. Tu n'es pas accessible au moindre raisonnement un tant soit peu construit et scientifique.

Encore une fois, toutes mes excuses... Et quand on pense que c'est lui qui décide ce qui doit être publié ou pas... aïeuuu !

Conclusion

Bien que mon discours ait été quelque peu perturbé par notre « bien-aimé » rédacteur en chef, la conclusion est évidente : les mathématiques sont parfaitement inutiles pour être un informaticien. Et j'irai même jusqu'à dire qu'il s'agit d'un préalable pour être un bon informaticien : ne pas avoir l'esprit embrouillé par un tas de considérations complexes qui ne servent absolument à rien. Nombre d'informaticiens vous l'avoueront sans peine et s'en vanteront même : ils sont nuls en maths. Et après ? Ça ne les empêche pas de diriger des équipes...

Ben, heureusement que je ne bosse pas avec eux... J'ai déjà suffisamment à faire avec les infographistes...

Ah mais là ça suffit môssieur le suppôt du Grand Serpent Matheux ! On tape sur tout le monde à ce que je vois, et... aïeuuuu... Non, je ne me tairai plus, je... aïeuUUU ! Bon, bon, je rentre dans le droit chemin, mais n'oubliez pas :

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

OPENSTACK : 1, 2, 3... DÉPLOYEZ !

par Benjamin Zores [Architecte Logiciel @ Alcatel-Lucent]

Après après avoir découvert le projet d'infrastructure Cloud « as a Service » OpenStack au sein du numéro 165, intéressons-nous désormais à son déploiement de manière très pratique. Comme nous allons nous en rendre compte, rien n'est moins simple ;-)

Le projet OpenStack suit un cycle de développement de 6 mois et comble chaque jour un peu plus le fossé qui le sépare de son opposant propriétaire principal : *Amazon Web Services*. Ainsi, le projet qui fut initialement lancé avec 2 services simples, *Nova Computing* (équivalent d'Amazon EC2) et *Swift Object Storage* (équivalent d'Amazon S3), dispose aujourd'hui d'une bonne dizaine de composants/modules optionnels. Selon votre besoin, il n'existe donc pas un OpenStack, mais bien une multitude de configurations diverses et variées que vous pourrez déployer. En fonction de vos moyens (financiers et matériels), de vos enjeux (besoin personnel, petite entreprise souhaitant déployer facilement quelques machines virtuelles, ou hébergeur souhaitant ouvrir un service de *cloud computing* équivalent à Amazon pour des clients désireux d'un cloud privé ou hybride), OpenStack sera déployé de manière plus ou moins évidente. En outre, et c'est un reproche récurrent au projet, les mises à jour ne sont guère faciles ou automatisées.

1 DevStack

Commençons par l'approche la plus simple et qui vous permettra rapidement de tester le projet, sans gros investissement, mais qui n'a que peu d'intérêt en production. Cette approche est permise grâce au projet DevStack **[1]**. DevStack est un ensemble de shell scripts permettant le déploiement rapide d'un ou plusieurs composants d'OpenStack à des fins de test. Le projet a été initié par RackSpace (l'une des compagnies initiatrices du projet OpenStack) et est désormais maintenu par la communauté OpenStack elle-même. Il permet de déployer OpenStack sur une unique machine et est principalement utilisé par les développeurs du projet à des fins d'intégration continue, tests de non-régression... Il n'a clairement pas vocation à être utilisé pour créer un environnement de production. À noter que le projet permet le déploiement sur un simple PC, mais également dans une machine virtuelle. Autant dire que dans ce second cas, instancier un orchestrateur de machines virtuelles dans une machine virtuelle ne vous fournira guère de performances. Encore une fois, le but est bien de tester. Si vous souhaitez vous y essayer, rien de plus simple. Il suffit de récupérer les sources de DevStack :

```
git clone https://github.com/openstack-dev/devstack.git
```

et de demander le déploiement dans un répertoire de votre distribution, via un simple :

```
./stack.sh
```

Et c'est tout ! En 20 minutes, tout est prêt, dans un environnement « bac à sable ». Lorsque vous en avez terminé, le projet peut nettoyer tout ce qui a été installé avec la même simplicité :

```
./unstack.sh
```

Voyons maintenant les choses dans un cadre un peu plus professionnel ou opérationnel.

2 All-In-One ou Multi-Nodes ?

Si OpenStack vous satisfait et semble la réponse à vos besoins à venir, passons donc à son déploiement réel et mise en production. Avant de se lancer sans réfléchir dans l'installation et le scripting, il est nécessaire de poser le crayon (ou en l'occurrence le clavier) et définir votre besoin.

Comme précisé préalablement, OpenStack fournit de nombreux composants. Certains comme **Nova** (computing et virtualisation), **Glance** (serveur d'images ou modèles de machines virtuelles), **KeyStone** (composant de gestion des identités), **Horizon** (interface web de contrôle), ou encore **Cinder** (gestion des disques virtuels) sont essentiels. D'autres, tels que **Neutron** (virtualisation du réseau), **Swift** (gestionnaire de stockage ou d'objets), ou plus récemment **Ceilometer** (moniteur de ressources système) et **Heat** (orchestrateur cloud) sont optionnels. À vous donc de bien définir vos besoins à court terme (il est toujours possible de compléter une installation d'OpenStack, mais cela requiert de re-configurer l'ensemble des modules).

À l'installation, OpenStack propose ainsi 2 approches : *All-In-One* et *Multi-Nodes* (ou nœuds multiples). Dit autrement, vous pouvez installer l'ensemble des composants (choisis) sur une unique machine physique (c'est évidemment l'approche la plus facile), ou vous pouvez préférer de dédier des machines physiques à certains composants d'OpenStack. Pour une petite entreprise, l'approche All-In-One peut tout à fait convenir. Si tant est que vous vous équipiez d'un petit serveur conséquent (e.g. 64 à 128 Go de RAM, 1 à 2 CPU de 6 à 8 cœurs et quelques To de disque dur), vous aurez tout ce qu'il faut pour permettre l'automatisation du déploiement d'un grand nombre de machines virtuelles et de services. Une configuration de ce type se négocie facilement à 5000 euros. C'est trop pour un particulier, mais reste raisonnable pour une entreprise.

Si vos enjeux sont conséquents : investissez d'emblée ! Dédiez une (ou plusieurs, pour de la haute disponibilité) machine(s) aux composants de contrôle (KeyStone, Ceilometer, Heat, Cinder, Glance, Neutron,...), plusieurs machines véloces (RAM et CPU) aux nœuds de computing Nova, et plusieurs machines plus « classiques » mais avec de nombreux et gros disques durs à Swift. L'investissement ne sera certes pas le même, mais le service rendu non plus. À vous les joies de la scalabilité à la demande pour chaque type de ressource. Vous l'aurez compris : vous devrez également acheter votre matériel en fonction du type de composant logiciel d'OpenStack qu'il sera amené à opérer. Mais surtout, il va falloir faire communiquer toutes ces machines ensemble et distribuer OpenStack au sein de ces dernières. La configuration et le déploiement de ce dernier devient alors plus « sportif » pour ne pas dire autrement ;-)

3 Déploiement basique

Un environnement de production classique d'OpenStack réside dans une approche multi-noeuds (avec un minimum de 3), comme présenté sur la figure 1.

Mais voilà, un déploiement multi-noeuds étant très spécifique et nécessitant un ouvrage complet pour être correctement expliqué, nous détaillerons donc l'approche All-In-One. Avouons-le, c'est aussi plus facile à décrire ;-) OpenStack peut être installé sur de nombreuses distributions. Pour ma part, j'ai choisi Ubuntu, qui travaille conjointement avec les équipes d'OpenStack pour intégrer le tout. Nous rentrerons dans les détails du déploiement plus loin, mais je ne saurai que trop vous conseiller de jeter un œil à la référence **[2]**, qui propose un script d'aide au déploiement (avec support des approches All-In-One et Multi-Nodes) pour Ubuntu. Le script fonctionne « globalement » bien (certains cas nécessitent une correction manuelle) et, contrairement à DevStack, génère réellement un environnement OpenStack de production. Voyons donc pour les détails.

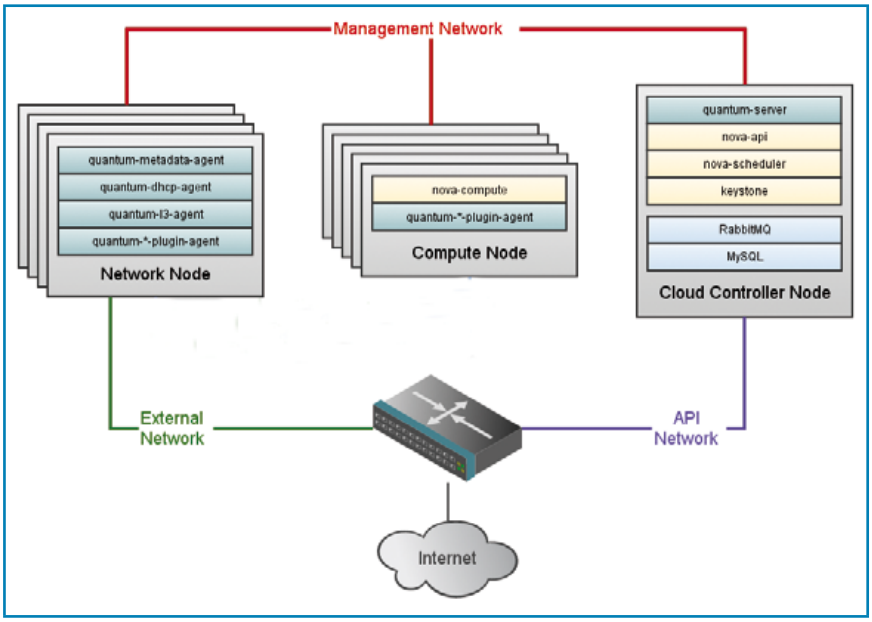


Figure 1 : Topologie OpenStack minimale de production

Avant toute chose, assurez-vous d'installer Ubuntu Server sur votre machine, en version 12.04.3 LTS pour processeurs 64 bits et procédez, si besoin, aux quelques mises à jour de rigueur :

```
sudo apt-get update && sudo apt-get dist-upgrade
```

Ceci fait, assurez-vous de la bonne configuration de votre réseau. Dans notre cas, nous disposons de 2 interfaces réseau : **eth0** qui est reliée à notre LAN, et **eth1** qui dispose d'un sous-réseau distinct dans lequel tourneront les machines virtuelles. Cette configuration permet d'isoler les machines virtuelles du LAN et ainsi de les rendre inaccessibles. C'est ainsi en configurant OpenStack, à la demande, et pour chaque VM, que l'on procédera au *mapping* des adresses IP « privées » des machines virtuelles, vers des IP « publiques » (accessibles depuis notre LAN).

Dans la vraie vie, en tant qu'hébergeur de services, ces IP « publiques » sont de véritables adresses IPv4 routables. Je ne pense pas cependant que beaucoup parmi les lecteurs disposent de plus d'une (celle de leur fournisseur câble/ADSL/fibre). Modifiez donc le fichier **/etc/network/interfaces** de la sorte :

```
auto lo
iface lo inet loopback

# this NIC must be on management network
auto eth0
iface eth0 inet static
    address 10.28.200.100
```

```
        netmask 255.255.255.0
        gateway 10.28.200.1
        dns-nameservers 8.8.8.8 8.8.4.4

    # this NIC will be used for VM traffic
    # to the internet
    auto eth1
    iface eth1 inet static
        address 10.28.201.100
        netmask 255.255.255.0
        dns-nameservers 8.8.8.8 8.8.4.4
```

Récupérez maintenant le script de déploiement :

```
sudo apt-get install git
git clone git://github.com/jedipunkz/
openstack_grizzly_install.git
cd openstack_grizzly_install
```

Installez ensuite un client NTP, afin de vous assurer que votre serveur est à l'heure. Cette étape est ici recommandée, mais indispensable dans une configuration multi-noeuds (tous les nœuds doivent être synchronisés) :

```
sudo apt-get install ntp
sudo cp conf/etc.ntp.conf /etc/ntp.conf
sudo service ntp restart
```

Continuons avec le minimum nécessaire à la configuration réseau de notre réseau virtuel. Dans notre cas, nous simplifions au possible. Dites au revoir à Neutron et sa configuration extrême et laissons place à un simple sous-réseau virtuel et pont, au travers de Linux Bridging :

```
sudo apt-get install -y vlan bridge-utils
```

Enfin, récupérez les archives cloud des dépôts Ubuntu, de manière à installer facilement les paquetages d'OpenStack :

```
sudo apt-get install ubuntu-cloud-keyring
sudo echo deb http://ubuntu-cloud.archive.
canonical.com/ubuntu precise-updates/grizzly
main >> /etc/apt/sources.list.d/grizzly.list
sudo apt-get update
```

4 Disque réseau iSCSI

Pour ne rien vous cacher sur ma configuration de production « low-cost », j'utilise un « petit » serveur Dell disposant de 2 CPU Xeon E5, chacun offrant 6 cœurs hyper-threadés, et auquel j'ai associé 64 Go de mémoire. Pour mon stockage, j'y ai attaché un NAS RackStation 812+ de chez Synology, avec 4 disques durs SATA de 2 To chacun. La marque du NAS importe peu, mon besoin était de disposer d'un support iSCSI, afin de pouvoir adresser mes disques à distance. Ces 4 disques sont configurés en RAID 5, me permettant de les changer à chaud et sans perte de données, si l'un venait à tomber.

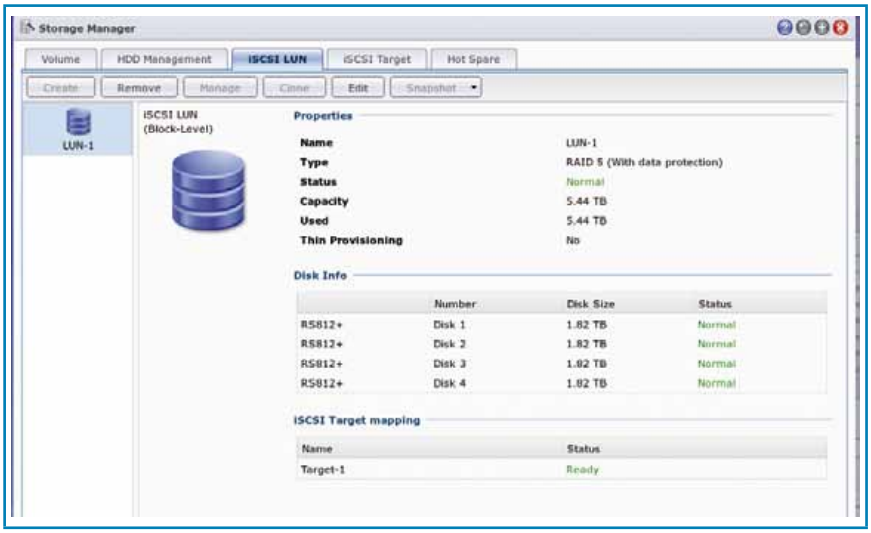


Figure 2 : Création d'un LUN iSCSI

Je décide donc de les agréger et de créer un LUN iSCSI, ainsi qu'une cible (ou *target*) iSCSI. Le LUN iSCSI est créé au niveau bloc (et non au niveau du système de fichiers) afin d'obtenir les performances maximales. Le LUN ainsi créé s'appellera simplement « LUN-1 » (Fig. 2). De la même façon, nous créons une cible iSCSI (appelée « Target-1 »), au-dessus du LUN, et dont l'IQN sera dans notre cas **iqn.2000-01.com.synology:RackStation812.name** (Fig. 3).

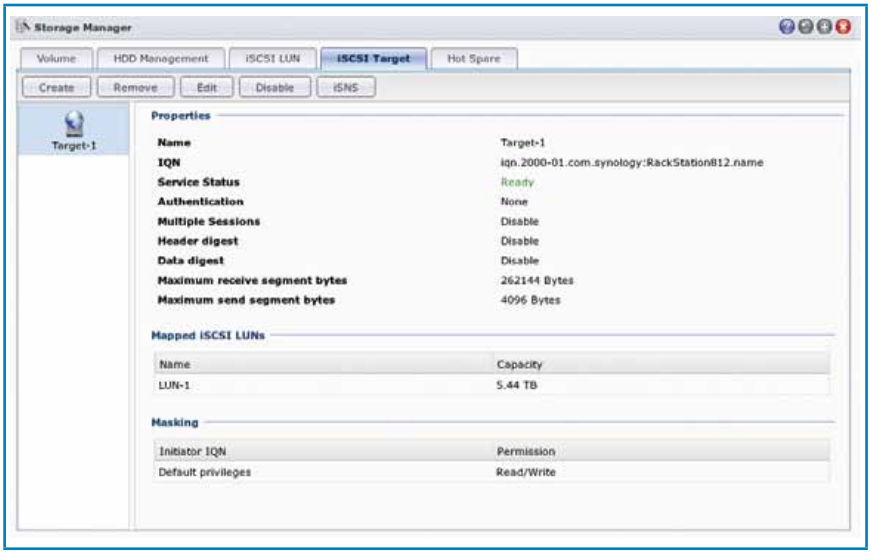


Figure 3 : Création d'une cible iSCSI

Ceci nous permet donc d'avoir un disque réseau adressable d'un peu moins de 6 To. Ne souhaitant pas profiter de Swift pour l'instant, ce disque permettra de stocker les disques virtuels de nos machines virtuelles, et donc d'être adressé

par Cinder. Vous pouvez (ou non) décider d'utiliser l'authentification CHAP, lors de la création de votre cible iSCSI. Cela permet de protéger l'accès au disque réseau. Seules les personnes disposant des codes d'accès pourront ainsi monter le disque. Dans notre cas d'exemple, j'ai toute confiance en moi-même, et m'en passerai donc ;-).

Installons le nécessaire pour monter notre disque iSCSI :

```
sudo apt-get install open-iscsi open-iscsi-utils
```

Et configurez le fichier **/etc/iscsi/iscsid.conf** pour que le montage ne soit plus manuel, mais automatique :

```
node.startup = automatic
```

Si vous aviez choisi d'utiliser l'authentification CHAP, c'est au sein de ce fichier qu'il faudrait la déclarer. Redémarrez maintenant le service iSCSI :

```
sudo /etc/init.d/open-iscsi restart
```

Et lancez la découverte des cibles iSCSI disponibles sur votre NAS (qui, dans notre exemple, dispose de l'adresse IP 10.28.200.102) :

```
sudo iscsiadm -m discovery -t sendtargets -p 10.28.200.102
10.28.200.102:3260,0 iqn.2000-01.com.synology:rackstation812.name
```

Nous retrouvons bien l'IQN de la figure 3. Reste maintenant à se connecter au disque distant, via la commande suivante :

```
sudo iscsiadm --mode node --targetname iqn.2000-01.com.
synology:rackstation812.name --portal 10.28.200.102:3260 --login
Logging in to [iface: default, target: iqn.2000-01.com.
synology:rackstation812.name, portal: 10.28.200.102,3260]
Login to [iface: default, target: iqn.2000-01.com.
synology:rackstation812.name, portal: 10.28.200.102,3260]: successful
```

Confirmez ensuite la session de connexion et redémarrez le service iSCSI :

```
sudo iscsiadm -m session -o show
tcp: [1] 10.28.200.102:3260,0 iqn.2000-01.com.synology:rackstation812.name
sudo /etc/init.d/open-iscsi restart
```

Vérifions tout ça, mais nous devrions maintenant disposer d'un disque de 6 To :

```
dmesg
Loading iSCSI transport class v2.0-870.
iscsi: registered transport (tcp)
iscsi: registered transport (iser)
scsi7 : iSCSI Initiator over TCP/IP
scsi 7:0:0:0: Direct-Access     SYNOLOGY iSCSI Storage    3.1  PQ: 0 ANSI: 5
sd 7:0:0:0: Attached scsi generic sg2 type 0
sd 7:0:0:0: [sdb] 11692753536 512-byte logical blocks: (5.98 TB/5.44 TiB)
sd 7:0:0:0: [sdb] Write Protect is off
sd 7:0:0:0: [sdb] Mode Sense: 3b 00 00 00
sd 7:0:0:0: [sdb] Write cache: disabled, read cache: enabled, doesn't support DPO or FUA
sdb: unknown partition table
sd 7:0:0:0: [sdb] Attached SCSI disk
```

Bingo ! Notez qu'au niveau utilisateur, vous disposez d'un disque de 6 To via **/dev/sdb**. Impossible de dire à première vue que ce disque n'est pas local à votre serveur. Vous retrouverez également le nom complet du disque dans **/dev/disk/by-path** (qui est un alias vers **/dev/sdb**) :

```
/dev/disk/by-path/ip-10.28.200.102:3260-iscsi-iqn.2000-01.com.
synology:rackstation812.name-lun-0
```

Nous en avons enfin fini avec la configuration de notre serveur. Installons désormais OpenStack (il était temps...) !

5 Configuration

Comme précisé précédemment, nous procéderons à un déploiement de type All-In-One. Nous n'utiliserons pas non plus Neutron, mais un simple bridge, réalisé par le composant Nova-Network d'OpenStack. Pour ce faire, copiez le modèle suivant et paramétrons-le :

```
cp setup.conf.samples/setup.conf.allinone.nova-network setup.conf
```

Le répertoire **setup.conf.samples** dispose de multiples exemples de configuration. Libre à vous de choisir celle qui

vous semblera la plus proche. OpenStack et ses différents composants utilisent MySQL pour gérer son environnement. Chaque composant ou service dispose de sa propre table et potentiellement de son propre utilisateur et mot de passe. Dans notre cas, nous opterons pour un utilisateur par composant (dont le nom est celui du composant) et d'un mot de passe unique : « password ». Modifiez donc le fichier **setup.conf** en conséquence :

```
# Database user and password
MYSQL_PASS='password'
DB_KEYSTONE_USER='keystone'
DB_KEYSTONE_PASS='password'
DB_GLANCE_USER='glance'
DB_GLANCE_PASS='password'
DB_NOVA_USER='nova'
DB_NOVA_PASS='password'
DB_CINDER_USER='cinder'
DB_CINDER_PASS='password'
```

Tant qu'à parler d'identité, définissons également un administrateur, un compte de démonstration et un service (naïvement appelé « service ») pour notre installation. Ces derniers vous permettront de vous interfacer avec les composants d'OpenStack, de les configurer (par CLI ou par l'interface Web Horizon) et de les administrer.

La notion de service est importante. Elle est également appelée « projet ». Typiquement, on associe un projet à un client et cela permet de lui allouer des ressources. Ainsi, votre environnement OpenStack devient « multi-tenant ». Une même installation peut ainsi se voir utiliser par différents clients, différentes compagnies et il est alors facile de les isoler (ou les facturer) les uns des autres. Les ressources globales (nombre de CPU virtuels, de mémoire, d'espace disque, d'adresses IP publiques...) peuvent ainsi être cloisonnées et des quotas maximums d'utilisation peuvent être mis en place par projet.

Dans notre exemple, nous ne définirons qu'un seul et unique projet ou service. Procédez donc à la configuration suivante :

```
# Keystone username and password
ADMIN_PASSWORD='password'
SERVICE_PASSWORD='password'
OS_TENANT_NAME='admin'
OS_USERNAME='admin'
OS_PASSWORD='password'
SERVICE_TOKEN='ADMIN'
SERVICE_TENANT_NAME='service'
DEMO_USER='demo'
DEMO_PASSWORD='demo'
```

Procédez de même pour la configuration réseau de notre solution :

```
# ------------------------------------------------------------
# if you run on one node (all in one), enable these parameters
# ------------------------------------------------------------
HOST_IP='10.28.200.100'
HOST_PUB_IP='10.28.200.100'
FLAT_INTERFACE='eth1'
# ------------------------------------------------------------
# network component type : 'quantum' or 'nova-network'
# ------------------------------------------------------------
NETWORK_COMPONENT='nova-network
# ------------------------------------------------------------
# nova-network parameters
# ------------------------------------------------------------
FIXED_RANGE='192.168.0.0/24'
FIXED_START_ADDR='192.168.0.2'
FLOATING_RANGE='10.28.200.192/27' # from .193 to .222
NETWORK_SIZE='256'
```

Les derniers paramètres définissent la plage IP adressable pour les machines virtuelles que vous créerez via OpenStack. Nova Computing, le composant de virtualisation d'OpenStack utilisera pour ce faire KVM comme hyperviseur et son serveur DHCP/DNS intégré pour attribuer une IP de cette plage à chaque VM nouvellement créée. Nova s'occupera également du routage et des règles de pare-feu spécifiques à chaque VM.

Les VM seront donc toutes cloisonnées à ce sous-réseau **192.168.0.0/24**, accessible via l'interface bridge **br100** de votre serveur. Elles restent donc toutes accessibles entre elles (d'un point de vue IP tout du moins), peuvent sortir vers l'extérieur via leur routeur (Nova, votre serveur), mais ne peuvent être jointes. Pour chaque VM, vous pouvez demander à OpenStack (encore une fois, par CLI ou via l'interface Web Horizon) d'attribuer une adresse IP publique. Si cette demande est faite, une IP sera attribuée dans la plage spécifiée par le champ **FLOATING_RANGE**. Charge ensuite à Nova-Network de devenir routeur et de transférer les paquets à destination de cette IP « publique » vers l'IP « privée » de notre réseau **192.168.0.0/24**.

Toutes vos machines n'ayant pas besoin d'être « publiquement » accessibles (e.g. une VM de serveur web le sera, mais la VM avec sa base de données ne le sera pas), ceci vous permet ainsi de restreindre les accès aux seuls besoins réels. Notez, encore une fois, qu'il s'agit là de la version simple des choses. Si nous avions opté pour Neutron, nous aurions pu créer dynamiquement des sous-réseaux virtuels. Dans notre cas, toutes les machines virtuelles feront partie du même sous-réseau privé. Dans la vraie vie, chaque service/projet ou client nécessitera d'avoir un ou plusieurs sous-réseaux qui lui sont propres, de manière à sécuriser sa solution de bout en bout. Neutron lui permettra de réaliser tout cela (et plus encore).

Enfin, terminons la configuration de notre script d'installation en spécifiant le disque qui sera utilisé par Cinder et l'adresse d'une image de machine virtuelle « Cloud-ready » d'Ubuntu, qui sera intégrée dans Glance, notre dépôt d'images :

```
# ------------------------------------------------------------
# misc parameters
# ------------------------------------------------------------
CINDER_VOLUME='/dev/disk/by-path/ip-10.28.200.102:3260-iscsi-
iqn.2000-01.com.synology:rackstation812.name-lun-0'
# ------------------------------------------------------------
# OS image parameters
# ------------------------------------------------------------
OS_IMAGE_URL="http://cloud-images.ubuntu.com/precise/current/precise-
server-cloudimg-amd64-disk1.img"
OS_IMAGE_NAME="Ubuntu 12.04 LTS x86_64"
```

Il ne vous reste plus qu'à lancer le script, patienter 15 à 20 minutes, prier pour que tout fonctionne et voilà :

```
sudo ./setup.sh allinone
Please check network interface which you login with.
If you login to host via wrong interface, you will lost network connectivity.
In all in one node, You have to login via management network.
Do you login to this IP address : 10.28.200.100 ? (y/n)
[...]
This script was completed. :D
You have done! Enjoy it. :)))))
```

Comme attendu, le script a installé pour vous OpenStack (une configuration basique du moins), tel que spécifié. Vous retrouverez donc le minimum nécessaire pour démarrer : Keystone, Glance, Cinder, Nova et Horizon (mais pas Neutron, Swift, Heat et Ceilometer).

L'interface web de configuration est à présent disponible via http://10.28.202.100/horizon/.

Pour les moins patients d'entre vous, c'en est fini. OpenStack est prêt à être utilisé. Le script étant ce qu'il est, il est facile à parcourir et d'en découvrir le contenu par vous-même si vous êtes intéressé. Pour les plus curieux, voyons dans les grandes lignes ce que le script nous a produit.

6 KeyStone

KeyStone est notre composant d'identité. Ce dernier s'interface avec MySQL pour y stocker les informations de configuration et d'identité. Sans trop rentrer dans les détails, il est possible, par commandes de type CLI, de créer des compagnies (ou « tenant »), des utilisateurs (ou « user »), des rôles (ou « role ») et des projets (ou « service ») via la commande **keystone**. Il est également évidemment possible de les associer les uns aux autres. C'est typiquement ce qu'a fait pour vous notre script, en jouant judicieusement avec les commandes suivantes (pour ne citer qu'elles) :

```
keystone tenant-create --name admin
keystone user-create --name admin  [...]
keystone role-create --name admin [...]
keystone user-role-add --user-id [...]
keystone service-create --name nova --type compute --description
'OpenStack Compute Service' [...]
```

7 Glance

Glance, comme tout autre service d'OpenStack, dispose également de sa propre base de données, créée par notre script. Comme les autres, il expose une API REST permettant de s'y interfacer. Son but est de servir des modèles de machine virtuelle. Dans les grandes lignes, notre script a permis d'en ajouter une à sa liste (la première en fait) :

```
wget --no-check-certificate ${OS_IMAGE_URL} -O os.img
glance image-create --name="${OS_IMAGE_NAME}" --is-public true
--container-format bare --disk-format qcow2 < ./os.img
```

Glance supporte de nombreux formats de conteneurs et de disques virtuels. Le plus utilisé reste cependant le format QCOW2 de Qemu, nativement supporté par KVM.

Vous trouverez sur Internet de nombreuses images « Cloud-ready » pour vos distributions préférées. Il est également très facile de réaliser vos propres modèles d'images. La documentation officielle du projet OpenStack vous proposera toutes les informations nécessaires pour créer vos propres images facilement **[3] [4]**. Grâce aux commandes précédentes, il vous sera ensuite facile de les mettre à disposition via Glance.

8 Cinder

Cinder est le composant fournissant un espace de stockage à vos machines virtuelles. Pour ce faire, nous avons utilisé notre disque iSCSI. Pour plus de simplicité, Cinder a été configuré pour utiliser LVM au-dessus d'iSCSI. De ce fait, le script a créé un volume physique (ou *p(rivate) v(olume)*) au-dessus de notre disque **/dev/sdb** et un groupe de volumes (ou *v(olume) g(roup)*) appelé « cinder-volumes », via les commandes LVM suivantes :

```
pvcreate ${CINDER_VOLUME}
vgcreate cinder-volumes ${CINDER_VOLUME}
```

Cinder étant configuré pour utiliser le VG « cinder-volumes », peu lui importe le ou les disque(s) qui le compose(nt). Il vous sera ainsi très facile d'ajouter de l'espace disque si le besoin s'en faisait sentir en ajoutant un nouveau disque à notre VG.

9 Nova Network

Nova est le composant historique de virtualisation. Si vous n'utilisez pas Neutron (c'est ici notre cas), il s'occupera également de la gestion de votre réseau virtuel. Dans notre cas, le script a donc permis de créer un réseau virtuel associé à l'interface de bridge **br100** et avec la configuration réseau détaillée précédemment.

```
nova-manage network create private --fixed_range_v4=${FIXED_RANGE} --num_networks=1 --bridge=br100
--bridge_interface=${FLAT_INTERFACE} --network_size=${NETWORK_SIZE} --dns1=8.8.8.8 --dns2=8.8.4.4
-multi_host=T
```

De la même façon, nous demandons à Nova de créer une plage d'adresses IP flottantes. Comprenez par là que les IP de cette plage sont associables (et dés-associables) à vos machines virtuelles, à la demande :

```
nova-manage floating create --ip_range=${FLOATING_RANGE}
```

OpenStack permet également à chaque utilisateur de créer des groupes de sécurité, associables à l'une (ou plusieurs) de ces machines virtuelles. Il s'agit là d'une notion classique de pare-feu. La différence étant qu'il s'agit d'un pare-feu virtuel, au niveau de Nova. Souvenez-vous, vos machines sont liées à un sous-réseau virtuel, dont Nova (ou Neutron, bref, le contrôleur réseau de votre solution) fait office de routeur. C'est ainsi via une configuration **iptables** de votre serveur physique (mais dépendant des groupes de sécurité définis dans OpenStack), que le routage vers vos machines virtuelles sera ou non autorisé. Dans l'exemple ci-dessous, notre script ajoute un nouveau groupe de sécurité appelé « default » (s'il n'existait pas déjà) et lui autorise le **ping** et les connexions entrantes TCP sur le port 22 (i.e. SSH) :

```
nova --no-cache secgroup-add-rule default tcp 22 22 0.0.0.0/0
nova --no-cache secgroup-add-rule default icmp -1 -1 0.0.0.0/0
```

Toutes les machines virtuelles créées pourront ainsi choisir de se voir assigner ce groupe de sécurité.

10 Pour aller plus loin...

Voici donc qui conclut notre article lié au déploiement d'OpenStack. Il est bien évident cependant qu'il ne s'agit là que d'un survol de cette solution, tant elle peut être complexe à mettre en place dans un environnement et une topologie réseau complexe (que l'on retrouve dans la vraie vie). La multiplication des nœuds étant chose naturelle (les besoins de disques de stockage, de virtualisation... étant croissants vitesse grand V), votre environnement est naturellement destiné à croître et il vous faudra opter pour l'approche multi-noeuds. Je ne saurai donc que trop vous recommander une plongée dans la documentation d'OpenStack **[5]**, qui a le mérite d'être relativement complète.

Nous ne nous sommes pas non plus attardés sur les composants Ceilometer et Heat, apparus il y a quelques mois seulement, avec la version Havana d'OpenStack, et permettant monitoring et orchestration de la solution pour automatiser la scalabilité de vos machines virtuelles et des services associés. Nous sommes également passés à côté de Swift, qui est le service OpenStack analogue à Amazon S3, de serveur de ressources disques distribuées. Ce choix est cependant voulu. Tout d'abord parce qu'il est complexe et pas nécessairement utile à tout le monde (moins que la virtualisation en tout cas). Et ensuite, parce que Swift a ses limites, et un concurrent appelé Ceph **[6]** tend à le remplacer.

Swift est ce que l'on appelle un « Object Store » distribué. Il permet de stocker des fichiers ou ressources et de les récupérer (via API REST) et en assure l'intégrité et la haute-disponibilité. Mais il se contente de ces objets. Impossible donc de l'utiliser comme un véritable système de fichiers distribué, auquel on pourrait connecter Glance pour fournir des « Block Storage » où créer des disques virtuels. Ceph est un système de fichiers distribué, qui fait également office d'« Object Store ». Mais via une passerelle RADOS, il peut également être utilisé comme « Block Device ». Ceph a donc l'avantage d'être connectable à Glance (ce dernier le supporte nativement depuis Havana) et d'offrir une compatibilité avec l'API REST de Swift. L'implémentation est donc différente, mais la fonctionnalité est transparente à l'utilisateur. Nous reviendrons d'ailleurs sur Ceph (et son intégration à OpenStack) dans un article qui lui sera dédié.

Et enfin, si vous êtes curieux d'en apprendre davantage, je vous invite également à regarder en détails le composant Neutron et ses formidables capacités de virtualisation de réseau (qui vont jusqu'à fournir des équilibreurs de charges, ou « Load Balancer as a Service »). S'ajoutent également toutes les problématiques « classiques » de haute-disponibilité (ou HA), dont vous pourrez avoir davantage de détails au sein de la référence **[7]**.

Bon déploiement !! ■

Références

[1] http://devstack.org/

[2] http://jedipunkz.github.io/openstack_grizzly_install/

[3] http://docs.openstack.org/image-guide/content/ch_creating_images_manually.html

[4] http://docs.openstack.org/image-guide/content/ch_creating_images_automatically.html

[5] http://docs.openstack.org/grizzly/basic-install/apt/content/

[6] http://ceph.com/

[7] https://wiki.ubuntu.com/ServerTeam/OpenStackHA

JE PROGRAMME

LE GUIDE POUR APPRENDRE À PROGRAMMER EN 7 JOURS SEULEMENT !

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

GNU/LINUX MAGAZINE HORS-SÉRIE N°71

DISPONIBLE DÈS LE 14 MARS 2014

CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

boutique.ed-diamond.com

RÉALISER UNE EXTENSION PUPPET

par Romain Pelisse [Cloud Architect @ Red Hat Gmbh]

Si Puppet vient déjà naturellement avec le support de nombreux produits et outils, et que PuppetForge regorge aussi de nombreux modules complémentaires, il n'en reste pas moins que la plupart des systèmes d'information incorporent des logiciels « maison », ou simplement trop spécifiques pour bénéficier du support de la communauté. Heureusement, Puppet, comme pratiquement toutes les solutions open source, propose un mécanisme d'extensions permettant d'ajouter aisément le support de nouvelles « ressources », que nous allons étudier en détails dans cet article.

1 Prérequis

De précédents articles, pas tous issus de ma plume, ont déjà longuement évoqué les vertus de Puppet et de solutions semblables, telles que Chef **[Chef]** et CFEngine **[CFEngine]**. Cet article considère donc que le lecteur est déjà quelque peu familier avec l'outil, et que l'objectif est donc d'aller plus dans le détail que ne le ferait une présentation générale. Pour le lecteur peu familier du sujet, il est recommandé, en prélude à la lecture de cet article, de consulter l'article « Les sysadmins jouent à la poupée » **[PuppetArticle]**, disponible en ligne sur le site UnixGarden.

2 Pourquoi réaliser une extension pour Puppet ?

Comme nous allons l'évoquer en détails, réaliser une extension n'est pas difficile et peut se révéler crucial pour mettre en place, avec succès, Puppet au sein d'une infrastructure. Néanmoins, comme toujours avec ce genre de mécanisme, le risque de la personnalisation à outrance - le fameux « *Not Invented Syndrom* » **[NotInventedHere]**, n'est jamais loin. Pour éviter ce dernier, nous prendrons donc le temps nécessaire afin d'expliciter les cas d'usage où ce genre de développement peut se révéler réellement pertinent.

2.1 Intégrer un composant non supporté par Puppet

La première motivation qui vient à l'esprit est bien évidemment celle de réaliser l'intégration avec un produit qui n'est pas déjà présent dans les nombreuses ressources proposées par Puppet **[PuppetResources]**, ni dans les extensions déjà publiées sur PuppetForge **[PuppetForge]** ou ailleurs sur le net. Néanmoins, dans les faits, ceci est assez rare, car la plupart des logiciels les plus usuels sont intégrés sous forme de modules embarquant les extensions nécessaires.

2.2 Absence d'une extension appropriée

Malheureusement, les modules disponibles sur PuppetForge sont très nombreux, et à l'image des modules Perl disponibles sur le célèbre CPAN **[CPAN]**, leurs qualités sont très variables. En effet, pour un seul produit, comme MySQL, on peut aisément trouver une demi-douzaine d'extensions, aux fonctionnalités et à la maintenance fort variables. Pour cette raison, il est souvent tentant - et parfois entièrement justifié, de développer soi-même sa propre extension, plutôt que de se risquer à utiliser, et potentiellement maintenir, une extension de la communauté dont l'intégration peut se révéler difficile, voire problématique.

2.3 Comment décider ?

Dans tous les cas, il est extrêmement difficile d'évaluer quelle serait la meilleure décision à prendre. La meilleure stratégie consiste, comme toujours, à se concentrer sur le besoin. Si l'on peut trouver sur PuppetForge un module ou une extension offrant les fonctionnalités nécessaires pour le satisfaire, il est probablement plus pertinent de l'utiliser, plutôt que de réaliser une extension...

Dans le cas inverse, on peut alors considérer la réalisation d'une extension spécifique, mais uniquement en se posant, au préalable, une question importante : pourquoi n'existe-t-il aucun module implémentant mon besoin ? En effet, ce genre de situation peut être un excellent indicateur du fait que l'approche choisie n'est tout simplement pas la bonne...

2.4 Implémenter un processus spécifique

Une autre situation dans laquelle il est tout à fait légitime de se poser la question de la conception d'une extension spécifique

est la mise en place de processus spécifiques. Par exemple, l'exécution d'un script « maison » à l'issue de l'installation d'un composant. Bien que ce dernier puisse être simplement exécuté par Puppet, par l'intermédiaire de l'utilisation d'une ressource **exec** **[Exec]** - nous verrons plus loin le défaut d'une telle stratégie, réaliser une extension est souvent une approche bien plus propre.

En outre, si le jeu de scripts est régulièrement utilisé, il est bien plus propre d'encapsuler (*wrap*) leur utilisation au sein d'un module, voire d'une extension, si ce n'est que pour faciliter la maintenance de ces appels.

3 Étude de cas : tuned

Cet article va utiliser, pour illustrer son sujet, la réalisation d'une extension pour **tuned**. Ce service, spécifique aux distributions de Red Hat **[DocTuned]**, comme RHEL **[RHEL]** ou Fedora, permet de facilement optimiser le paramétrage de son système et la configuration de ses interfaces réseau selon l'utilisation de la machine. Rapide démonstration :

```
$ tuned-adm list
Available profiles:
- virtual-guest
- latency-performance
- powersave
- balanced
- throughput-performance
- virtual-host
Current active profile: /usr/lib/tuned/powersave/tuned.conf
```

Comme clairement illustré par l'extrait ci-dessus, **tuned** regroupe l'ensemble des paramétrages relatifs à un cas d'utilisation sous un nom de profil. Il suffit ensuite de sélectionner le nom de profil correspondant le mieux à l'utilisation de la machine.

Bien évidemment, si aucun nom de profil ne convient, il est probable que votre utilisation de la machine soit trop spécifique pour avoir recours à cet outil. Il faudra donc choisir le profil le plus proche, et procéder ensuite à des modifications.

Dans l'extrait ci-dessus, le nom des profils se passe de commentaire, et on constatera que le profil **powersave**, fort adapté à un ordinateur portable, a été activé sur le mien, ce qui semble un choix somme toute fort raisonnable. L'activation d'un profil s'effectue ainsi :

```
$ tuned-adm powersave
```

Pour fonctionner, la commande nécessite aussi que le service **tuned** soit démarré :

```
# service tuned start
Redirecting to /bin/systemctl start  tuned.service
[root@august ~]# service tuned status
Redirecting to /bin/systemctl status  tuned.service
tuned.service - Dynamic System Tuning Daemon
          Loaded: loaded (/usr/lib/systemd/system/tuned.service; disabled)
          Active: active (running) since Sat, 30 Mar 2013 16:45:38 +0100; 1s ago
         Process: 6870 ExecStart=/usr/sbin/tuned -d (code=exited, status=0/SUCCESS)
        Main PID: 6872 (tuned)
          CGroup: name=systemd:/system/tuned.service
                  6872 /usr/bin/python /usr/sbin/tuned -d
```

En effet, comme son nom l'indique (*Dynamic System Tuning Daemon*), une partie des optimisations effectuées par ce dernier sont réalisées de manière dynamique, et l'exécution de ce service est donc requise.

L'utilisation de **tuned** est trop spécifique pour être disponible au sein des ressources fournies par défaut par Puppet (il n'existe pas d'équivalent, à ma connaissance, sur les autres distributions), et même dans PuppetForge, on ne peut trouver d'implémentation d'une ressource Puppet pour ce service. Ce mécanisme de configuration dynamique du système est donc un parfait exemple pour notre cas d'étude.

En outre, et c'est un point non négligeable, le travail d'intégration à réaliser est très simple. Il s'agit de vérifier que le nom du profil correspond à celui configuré. Dans le cas contraire, on corrige la configuration. Cette simplicité facilitera l'analyse de l'étude de cas, et permettra surtout de se concentrer sur les aspects techniques de l'extension.

4 Intégrer « tuned » à l'aide de « exec »

Avant de réaliser une extension, nous allons tenter une première intégration, à l'aide de la ressource **exec**, qui nous permettra d'illustrer les limites d'une telle approche, et, espérons-le, de donner au lecteur une bonne idée de quand tenter (ou non) une telle implémentation...

4.1 Activer un profil « tuned » à l'aide de Puppet

L'objectif de notre cas d'étude est bien évidemment de pouvoir activer, sur n'importe quel système placé sous le contrôle de Puppet, le profil adéquat de **tuned**. En première approche, on peut donc simplement se contenter d'utiliser **exec** pour le faire :

```
package { 'tuned':
  ensure => installed,
}

service { 'tuned':
  ensure => running,
  require => Package['tuned']
}

exec { 'tuned-profile':
  command => '/sbin/tuned-adm active throughput-performance'
  require => Service['tuned'],
}
```

Une telle approche fonctionne mais, à chaque exécution de Puppet, la commande sera exécutée, et le profil souhaité - en l'occurrence **throughput-performance**, sera défini pour la machine. C'est loin d'être élégant, mais cela fonctionne.

Pour éviter d'avoir à dupliquer cet ensemble de code dès que l'on souhaite invoquer **tuned**, mais aussi pour le rendre plus paramétrable, on peut aisément le placer au sein d'un module :

/etc/puppet/modules/tuned/manifests/init.pp :

```
define tuned::profile($profile='throughput-performance', $tuned_
package='tuned', $tuned_service='tuned') {
```

```
  package { $tuned_package:
    ensure => installed,
  }

  service { $tuned_service:
    ensure => running,
    require => Package[$tuned_package]
  }

  exec { 'tuned-profile':
    command => "/sbin/tuned-adm active $profile"
    require => Service[$tuned_service],
  }
}
```

Tout ceci fonctionne sans incident, mais pose néanmoins un problème. En effet, si le système cible ne dispose pas de paquetage **tuned**, comme par exemple sur un système basé sur la distribution Debian, l'exécution échouera. Nous allons voir immédiatement comment corriger ceci.

4.2 Gérer les systèmes sans « tuned »

Pour supporter ce cas, nous allons donc introduire une instruction de sélection de type **case**, qui utilisera la valeur du **fact osfamilly**, fournie par **facter** **[Facter]**, afin de déterminer si **tuned** doit, en effet, être disponible sur le système.

/etc/puppet/modules/tuned/manifests/init.pp :

```
define tuned::profile($profile='throughput-performance',
$tuned_package='tuned', $tuned_service='tuned') {

  case $osfamilly {
    "RedHat":
       package { $tuned_package:
         ensure => installed,
       }

       service { $tuned_service:
         ensure => running,
         require => Package[$tuned_package]
       }

       exec { 'tuned-profile':
         command => "/sbin/tuned-adm active $profile"
         require => Service[$tuned_service],
       }
    }
    default: {
      notice("No support for 'tuned' on system: $hostname)
    }
  }
}
```

4.3 Conserver l'idempotence

Tel que notre module est conçu pour le moment, à chaque exécution de l'agent **puppet** ou à chaque invocation de la commande **puppet apply**, le profil actif de **tuned** sera redéfini, quelle que soit sa valeur actuelle. En plus d'être relativement inélégant, et de consommer de la ressource (CPU et mémoire) de manière complètement inutile, ce défaut nuit à la nature « idempotente » que devrait avoir toute exécution de l'agent de Puppet. Bref, c'est un défaut qu'il nous faut corriger.

La première approche du problème consiste à définir une commande, dans l'attribut **unless**, qui retournera un statut **0** si le profil est déjà correctement défini. Dans ce cas, la commande ne sera pas exécutée.

Malheureusement, une telle commande n'existe pas dans l'interface proposée par **tuned-adm**, il est donc nécessaire de réaliser un petit script pour parvenir à obtenir un tel résultat :

```
# la commande retourne le nombre d'occurrences de
la valeur de "$profile"
tuned-adm active $profile | grep -c -e $profile
if [ ${?} -ne 0 ]; then
  echo "Wrong profile for tuned"
fi
```

Heureusement pour nous, le script ci-dessus peut être factorisé pour rentrer en une ligne :

```
test $(tuned-adm active $profile | grep -c
-e $profile) -ne 0
```

Néanmoins, avant de pouvoir placer cette commande dans notre attribut **unless**, il nous faut encore - afin de rester cohérent avec le fonctionnement de Puppet, remplacer chaque commande par son chemin complet, ce qui donne :

```
  ...
  exec { 'tuned-profile':
      command => "/usr/sbin/tuned-adm profile $profile":
      unless => "/usr/bin/test $(/usr/sbin/tuned-adm
active $profile | grep -c -e $profile) -ne 0",
  }
  ...
```

Rendus à ce stade, notre tâche **exec** fonctionne comme nous le souhaitions, c'est-à-dire que la commande de mise à jour du profil pour **tuned** ne sera exécutée que, si et seulement si, le profil n'est pas celui souhaité.

Malgré ce succès, il apparaît clairement que cette tâche **exec** est loin d'être élégante, et constitue le point faible de notre implémentation. En outre, l'exécution systématique du petit script embarqué dans la définition du paramètre **unless** n'est probablement pas des plus performantes...

```
# time /usr/bin/test $(/usr/sbin/tuned-adm active
$profile | grep -c -e $profile) -ne 0

real    0m0.192s
user    0m0.086s
sys     0m0.089s
```

Encore plus embêtant, imaginons que notre petit script présente un défaut de logique et qu'il nous faille l'améliorer - arriverons-nous encore à en faire un *one liner* ? Nous pourrions être alors amenés à devoir déployer un script local, uniquement afin que notre tâche **exec** fonctionne proprement !

```
...
file { '/usr/bin/check-tuned-profile':
  source => 'puppet:///modules/tuned/check-profile',
}

exec { 'tuned-profile':
    command => "/usr/sbin/tuned-adm profile $profile":
    unless => "/usr/bin/check-tuned-profile $profile",
    require => File['/usr/bin/check-tuned-profile'],
}
...
```

Bref, il apparait très clairement que même si notre implémentation fonctionne, elle n'est pas aisée à maintenir et se révèlera rapidement bloquante pour toute évolution. Bien évidemment, elle forme déjà une base pour tester le principe et valider sa réelle valeur ajoutée.

Dans un contexte industriel, il serait probablement pertinent de déployer ce module quelques temps, avant de considérer la ré-implémentation de la partie faite sous forme de **exec**. Dans notre cas, nous allons de suite améliorer ceci, ce qui nous permettra d'illustrer comment concevoir, aussi proprement que possible, une extension pour Puppet.

5 Implémentation de l'extension Puppet

5.1 Resource et Provider

Un des concepts clés de la couche d'abstraction de Puppet est le distinguo réalisé entre la notion de *Resource* et de *Provider*. Si la première décrit un état à atteindre sur la machine cible (un fichier à déployer, un service à démarrer, ...), le second se charge, selon le système d'exploitation et son contenu, d'exécuter les actions nécessaires.

COMMENT CRÉER UN SYSTÈME SUR-MESURE POUR VOTRE PLATEFORME ET INTÉGRER FACILEMENT VOS APPLICATIONS ?

PERSONNALISATION DE BUILDROOT

OPEN SILICIUM N°10

ACTUELLEMENT DISPONIBLE

CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

boutique.ed-diamond.com

Ainsi la ressource **Service** permet d'indiquer qu'un service doit être démarré, et le **Provider** `initd` sera utilisé, par défaut, pour assurer ceci sur une distribution RHEL. Si votre système utilise, par exemple, `daemontools` **[DAEMONTOOLS]** plutôt que `initd`, Puppet peut bien évidemment être configuré pour utiliser ce dernier.

5.2 Définition de la ressource « tuned »

5.2.1 Déclaration d'un nouveau Type

Fort de l'analyse précédente, ce ne sera pas une surprise pour le lecteur d'apprendre que la première étape pour la construction d'une extension est donc de définir un nouveau **Type** de ressource. Cette définition s'effectue très naturellement par la création, à l'aide de l'API de Puppet, en Ruby, d'un nouveau ... Type ! :)

```
Puppet::Type.newtype(:tuned) do
  @doc = "Ensure that target has the
appropriate tuned profile activated."
end
```

Les trois lignes ci-dessus suffisent donc à créer un nouveau type, nommé `tuned`, qui n'accepte, pour le moment, aucun paramètre et peut seulement être utilisé ainsi au sein d'un fichier `manifests` :

```
tuned { 'tuned':
}
```

On notera aussi l'ajout de l'attribut `doc` qui permet de documenter le propos de la nouvelle ressource. Il n'est probablement pas nécessaire d'ajouter cet attribut pour permettre la définition d'une nouvelle ressource, mais s'y astreindre constitue certainement une bonne pratique.

À ce stade, néanmoins, Puppet, lors de l'exécution de son agent, relèverait une erreur, car cette nouvelle ressource, bien que proprement déclarée, valide et connue de ce dernier, ne dispose pour le moment d'aucun Provider à même de mettre en place l'état désiré sur le système cible.

5.3 Ajout de paramètres au nouveau type

Avant d'aller plus loin et de réaliser l'implémentation d'un Provider, nous allons tout d'abord ajouter un paramètre à notre nouvelle ressource. Compte tenu de notre cas d'étude, le seul paramètre réellement nécessaire est le nom du profil `tuned` à utiliser. On modifie donc ainsi notre nouveau type :

```
Puppet::Type.newtype(:tuned) do
  ...

  newparam(:profile) do
    desc "tuned profile to activate"
  end
  ...
end
```

L'API permettant de déclarer de nouveaux paramètres est relativement simple d'utilisation et a le bon goût de venir avec un attribut `desc` destiné à contenir une description du paramètre - de manière complètement similaire au précédent `@doc`. On peut donc maintenant ajouter un paramètre à la déclaration de notre ressource, pour indiquer le nom du profil `tuned` à utiliser :

```
tuned { 'tuned':
  profile => 'powersave',
}
```

5.3.1 Utiliser le paramètre « nom »

Comme tout utilisateur de Puppet le sait, une ressource dispose toujours d'un paramètre `name`, dont la valeur correspond à celle utilisée lors de la déclaration de la ressource dans le fichier `manifests`. Dans notre exemple ci-dessus, la valeur du paramètre `name` est donc `tuned`.

Compte tenu du fait que notre seul paramètre est le nom du profil, il semble très pertinent d'utiliser ce dernier pour le paramètre `name`. En outre, ceci n'apportera que plus de clarté à notre déclaration de ressource :

```
tuned { 'powersave': }
```

Pour indiquer à Puppet que ce paramètre est le paramètre `name`, il suffit d'ajouter simplement l'appel à la fonction `isnamevar` :

```
Puppet::Type.newtype(:tuned) do
  ...

  newparam(:profile) do
    desc "tuned profile to activate"
    isnamevar
  end

end
```

5.3.2 Ajout du paramètre « ensure »

De nombreuses ressources Puppet proposent souvent les mêmes paramètres. Le meilleur exemple de ceci étant le paramètre `ensure`, qui permet d'indiquer quel état Puppet doit garantir pour la ressource. Pour rappel, il existe deux états possibles : `absent` et `present`. Le premier indiquant que la ressource ne doit pas être présente sur le système, et l'autre que, au contraire, Puppet doit s'assurer de sa disponibilité sur le système.

Pour intégrer notre extension à la sémantique usuelle de Puppet, il est donc approprié d'ajouter ce paramètre. Heureusement, l'API propose des fonctions prédéfinies pour ajouter ce genre de paramètres, il nous suffit donc d'utiliser la fonction associée (`ensurable`) :

```
Puppet::Type.newtype(:tuned) do
  ...

  ensurable

  ...
end
```

Pour conclure cette partie, voyons une dernière fois comment notre ressource peut désormais être intégrée au sein d'un fichier `manifest` :

```
tuned { 'powersave':
  ensure => present,
}
```

5.4 Création du Provider pour « tuned »

Si nous avons maintenant parfaitement défini notre nouvelle ressource, ainsi que les paramètres qu'elle accepte, il n'en reste pas moins que nous n'avons effectué aucune implémentation concrète. Pour faire un parallèle avec la programmation objet, nous n'avons pour le moment que défini la classe abstraite, ou plutôt l'interface au sens Java du terme.

Pour permettre à Puppet de mettre cette nouvelle ressource sur les systèmes cibles, il est maintenant nécessaire de fournir cette implémentation « concrète », et donc d'ajouter le code du Provider à notre extension.

Là encore, l'API Puppet rend le travail facile. Il suffit d'invoquer la même API que

précédemment, mais d'ajouter un appel à la méthode **provide**. Ensuite, on doit définir une classe Ruby disposant des trois méthodes nécessaires au cycle de vie de Puppet : **create**, **destroy** et **exists?**.

```
Puppet::Type.type(:tuned).provide(:tunedadm) do
    desc "Tuned provider."

    def create
    end

    def destroy
    end

    def exists?
    end
end
```

5.4.1 Implémentation de la méthode « create »

C'est la méthode **create** qui sera invoquée par Puppet s'il détermine que le système cible n'est pas dans l'état souhaité. Dans notre cas, il s'agit d'un système où le profil **tuned** n'est pas celui défini par le paramètre **profile**.

L'implémentation de cette méthode est réduite au minimum : il suffit d'invoquer la commande **tuned-adm** en lui passant l'option **profile** accompagnée de la valeur de l'attribut **profile** :

```
...
    def create
        Puppet.info("Tuned: Switching to profile '#{@
resource[:profile]}'")
        system("tuned-adm profile #{@resource[:profile]}")
    end
...
```

Pour les lecteurs peu familiers du langage Ruby, il faut noter que celui-ci dispose d'un mécanisme de *templating*, très apprécié, qui permet de placer le contenu de variables au sein d'une chaîne de caractères. C'est à ce mécanisme que l'on fait appel dans le code ci-dessus.

On notera qu'on pointe vers le nom du profil par l'intermédiaire de l'attribut **profile**, mais on eût pu tout aussi bien passer par l'attribut **name**. En effet, par construction, ces deux attributs ont la même valeur.

5.4.2 Implémentation de la méthode « destroy »

Cette méthode est invoquée lorsque Puppet souhaite assurer l'absence de la ressource. Ceci arrive, par exemple, quand le paramètre **ensure** contient la valeur **absent**. En tout état de cause, dans notre cas, il n'existe pas de logique à implémenter. Il n'existe pas d'élément à supprimer (fichier), ni d'état auquel revenir (configuration précédente de **tuned**).

On implémente néanmoins cette méthode en ajoutant un appel à la journalisation, avec le niveau **debug**, juste pour indiquer que la méthode a bien été invoquée par Puppet :

```
...
    def destroy
        Puppet.debug("Tuned: destroy called - nothing
to do...")
    end
...
```

Note

La suppression du paquetage associé à **tuned** ou l'interruption de son service n'est pas à la charge de notre extension, mais des ressources standards de Puppet (respectivement **package** et **service**). Il est essentiel de ne pas chercher ici à implémenter de tels comportements et de rester cohérent avec la logique du produit. Notre extension a pour seul propos de configurer le profil **tuned** approprié - rien d'autre.

5.4.3 Implémentation de la méthode « exists »

Cette méthode est de loin la plus cruciale et aussi la plus complexe. En effet, c'est ce code qui permettra à Puppet de déterminer si le système cible est dans l'état souhaité. Ceci signifie donc que cette méthode sera invoquée de manière systématique, à chaque exécution de l'agent Puppet.

Partant de ce principe, il est donc essentiel que ce code soit non seulement débarrassé de toute erreur de logique, mais également performant, sinon, chaque exécution de Puppet sera pénalisée par ce dernier.

Dans le cadre de l'article, nous allons nous contenter de réutiliser une commande système pour vérifier la configuration de **tuned**. Il est évident qu'il ne s'agit pas là de l'implémentation la plus performante, et nous discuterons plus loin de comment l'améliorer.

```
...
    def exists?
        Puppet.debug("Tuned: Check if profile #{@
resource[:name]} is activated.")
        status = system("tuned-adm active | grep -c #{@
resource[:name]} > /dev/null ")
        Puppet.debug("Tuned: status is #{status}")
        status
    end
...
```

5.5 Organisation d'une extension

Voyons maintenant comment organiser le code de notre extension. Bien qu'il soit possible d'ajouter directement des extensions au sein de Puppet, il est, dans la plupart des cas, plus élégant de placer simplement le code au sein d'un module dédié.

Nous allons donc concevoir un module pour **tuned**, dans lequel on place le code associé à l'extension. Ainsi, au sein du répertoire de l'extension, on ajoute un répertoire **lib/puppet** contenant un sous-répertoire **provider** et un sous-répertoire **type**.

Par convention, mais aussi par souci de clarté, on place, dans le répertoire **provider**, un sous-répertoire portant le nom du type que le code du *provider* qu'il contient réalise. Ainsi, dans notre cas, on y place donc un sous-répertoire **tuned**.

Enfin, on place le code du type **tuned** dans un fichier nommé **tuned.rb** dans le sous-répertoire **type**, et le code associé à son **provider** dans le sous-répertoire **provider/tuned**, dans un fichier nommé **tuned-adm.rb**.

Pourquoi ne pas nommer le fichier du **provider** comme le type **tuned.rb** ? Simplement pour rester cohérent avec la logique de Puppet. En effet, un type est une description d'une ressource abstraite, indépendante du système cible, alors qu'un **provider** est une implémentation concrète, justement spécifique au système cible.

Ainsi, pour garder cette logique, nous avons choisi de nommer le fichier contenant le code source du **provider**, **tuned-adm**, afin d'indiquer que cette implémentation du **provider** s'appuie sur la commande **tuned-adm**. En fait, cette nomenclature est tout à fait similaire à celle des implémentations en Java (où une implémentation de **List** utilisant un tableau en interne se nomme **ArrayList**).

Voici, ci-dessous, un récapitulatif graphique de l'organisation de notre module :

```
modules/tuned/
|-- lib
|   `-- puppet
|       |-- provider
|       |   `-- tuned
|       |       `-- tunedadm.rb
|       `-- type
|           `-- tuned.rb
|-- LICENSE
`-- manifests
    `-- init.pp

6 directories, 5 files
```

5.6 Le fichier manifests du module

L'organisation du module présentée dans la précédente section inclut aussi un fichier **init.pp** dont nous n'avons pas encore discuté le contenu et le propos. Nous allons donc maintenant nous intéresser à celui-ci.

Comme le lecteur l'aura probablement noté, l'implémentation de notre nouvelle ressource **tuned** ne s'occupe pas de vérifier l'installation du paquetage nécessaire, ni même le démarrage du service associé, comme évoqué au début de l'article. En effet, pour ces opérations, nous allons nous contenter de réutiliser les ressources Puppet existantes au sein de ce fichier **init.pp** :

```
define tuned::tune() {

  case $operatingsystem {
    default: { $package = 'tuned' }
  }

  package { "$package":
   ensure => installed,
  }

  service { 'tuned':
    ensure => running,
    require => Package[$package],
  }

  tuned { $name:
    ensure => present,
    require => Service[$package],
  }
}
```

5.7 Déployer le module

Il ne nous reste plus qu'à installer ce répertoire dans le sous-répertoire **modules** de Puppet pour déployer notre nouvelle ressource. Une fois ceci fait, nous pourrons non seulement utiliser la nouvelle ressource **tuned**, mais, mieux encore, la définition de **tuned::tune()** pour gérer l'installation des paquetages nécessaires et le démarrage du *daemon* **tuned**.

6 Qu'avons-nous gagné en réalisant cette extension ?

Le lecteur attentif notera que l'implémentation actuelle de notre extension n'est guère plus qu'un enrobage (*wrapping*) des commandes shell utilisées dans notre précédente implémentation à l'aide de **exec**. Nous allons donc maintenant étudier les améliorations et les évolutions possibles à partir de cette extension.

Afin de ne pas inonder l'article d'extraits de code, ces améliorations et évolutions seront seulement discutées.

6.1 Amélioration des performances

Comme évoqué plus haut, déterminer si le système est déjà configuré, ou non, pour utiliser le bon profil est une action systématique - déclenchée à chaque exécution de l'agent Puppet, et peu performante, car elle s'appuie non seulement sur l'exécution d'un script shell, mais aussi sur l'appel à la commande **tuned-adm**.

Comme le soulignait très clairement la série de commandes décrites ci-dessous, l'accès à un simple fichier contenant l'information sera déjà un gain important en termes de performance :

```
# time tuned-adm active
Current active profile: powersave

real    0m0.088s
user    0m0.064s
sys     0m0.015s
# tuned-adm active > /tmp/tuned-state
# time grep powersave /tmp/tuned-state
Current active profile: powersave

real    0m0.004s
user    0m0.000s
sys     0m0.002s
```

Pour corriger ce problème de performance, l'extension peut donc être modifiée pour tester, lors de son exécution, si un tel fichier existe, et le cas échéant, le créer. Ainsi, la commande **tuned-adm** ne sera invoquée qu'une seule fois, la première fois.

Bien évidemment, ce genre de fichier de cache peut vite être rendu invalide, il sera donc pertinent de placer, par exemple, une tâche dans la **crontab** pour effacer le fichier régulièrement (toutes les 24h par exemple) et forcer ainsi notre extension à le recréer.

6.2 Intégration avec Facter

La première amélioration, utilisant un fichier pour stocker l'état de la configuration de **tuned**, est déjà appréciable, mais on peut aller plus loin. En effet, comme tout utilisateur de Puppet le sait, ce dernier s'intègre naturellement avec Facter et expose tout *fact* connu de l'outil comme variable au sein d'un manifest Puppet.

Ainsi, on peut aller plus loin, et utiliser un *custom fact* **[CustomFact]** pour publier, au sein de **facter**, le nom du profil **tuned** utilisé. Cette solution a l'avantage, en plus, de renseigner Facter et d'offrir ainsi à d'autres outils (tels que MCollective **[MCollective]**) l'accès à l'information, et, plus important encore, de déléguer le cycle de vie de cette dernière à Facter. Plutôt que de devoir effacer régulièrement le fichier de stockage de l'état du système à l'aide d'une tâche au sein de la *cron*, on va donc laisser les mécanismes naturels de Facter s'en charger.

6.3 Support de profils personnalisés

Comme évoqué plus haut, il est tout à fait possible de définir ses propres profils, qui, techniquement, ne sont qu'un ensemble de fichiers placés sous le répertoire **/etc/tuned-profiles**. Il s'agit d'ailleurs, à ce jour d'une *feature request* **[SupportProfile-Perso]** ouverte sur le site de l'extension **[GithubTunedAdm]**.

Il serait donc assez élégant d'étendre l'extension pour prendre en charge une telle personnalisation. En première approximation, il s'agira vraisemblablement d'ajouter une définition dans le fichier **manifests**, pour permettre le déploiement des fichiers nécessaires au profil, puis déclencher un rafraîchissement de **tuned** par l'intermédiaire du paramètre **notify**. Néanmoins, il n'est pas exclu là encore qu'une modification du **provider** ne soit pas plus performante et facile à maintenir.

Conclusion

La conception d'une extension Puppet est donc très aisée et facile à déployer. Contrairement à ce que l'on pourrait croire, elle ne nécessite pas de réelles connaissances en Ruby, et, si on utilise de manière pertinente les ressources déjà existantes dans Puppet, le travail d'implémentation devrait se limiter au strict minimum.

Comme discuté en préliminaire à l'article, le plus difficile est de déterminer si la réalisation d'une extension est nécessaire, afin de ne pas tomber dans une personnalisation à outrance. Dans tous les cas, et dans la mesure du possible, discuter de son besoin avec la communauté et publier son extension.

L'avantage d'une extension est bien évidemment de disposer d'un meilleur contrôle sur les actions de Puppet et d'ajouter plus facilement des contrôles ou des optimisations de performances. D'une manière générale, une extension est bien plus facile à maintenir qu'un `exec` et offre surtout la possibilité d'ajouter des fonctionnalités.

Sans tomber dans une démarche systématique, consistant à remplacer toute instruction `exec` au sein d'un `manifest` par une extension, il reste pertinent d'analyser ces dernières et de se demander si leur implémentation ne sera pas plus efficace et de maintenance plus aisée sous forme d'extension. Ceci est d'autant plus pertinent si ces ressources `exec` concernent un composant logiciel récurrent. ■

Références

Puppet :

[PuppetArticle] http://www.unixgarden.com/?s=Puppet

[CustomFact] http://docs.puppetlabs.com/guides/custom_facts.html

[Facter] http://puppetlabs.com/facter

[PuppetResources] http://docs.puppetlabs.com/references/latest/type.html

[PuppetForge] https://forge.puppetlabs.com/

Ecosystème :

[MCollective] http://puppetlabs.com/mcollective

[Chef] http://www.opscode.com/chef/

[CFEngine] http://cfengine.com/

Autres :

[DocTuned] https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Power_Management_Guide/tuned-adm.html

[RHEL] https://www.redhat.com/products/enterprise-linux/

[DAEMONTOOLS] http://cr.yp.to/daemontools.html

[SupportProfilePerso] https://github.com/rpelisse/puppet-tuned/issues/2

CRÉER UNE APPLICATION PERL AUTOUR DE MYSQL :

INTÉGRATION AVEC MOJOLICIOUS, HTML::TINY ET HTML::FORMHANDLER (3/3)

par Dominique Dumont

Reprenons l'histoire du hacker à qui on a demandé de créer des pages web pour le suivi de la production logicielle de la société. Comme tout hacker qui se respecte, il préfère que la machine fasse son boulot à sa place. Pour ce faire, il a mis en place une base de données MySQL, un ORM (DBIx:Class). Reste maintenant à fournir un serveur web pour exploiter les données de cette base.

Petit rappel des épisodes précédents : notre hacker favori a expliqué comment mettre en place dans une base de données MySQL les tables et relations qui assureront la persistance des données de production. Puis, comment lire et écrire ces données avec **DBIx:Class**. Le code SQL de la structure de la base et le petit script contenant les exemples d'accès à la base sont disponibles dans un dépôt git **[GITHUBADOD]**.

1 Exploitation des données

Maintenant que nous savons écrire des données dans la base, nous allons voir comment récupérer ces données pour les afficher dans des pages HTML avec **Mojolicious::Lite**. Ce module fournit une infrastructure « légère » de serveur web. Ceci permet de tester rapidement les pages produites avec le mini-serveur de Mojolicious. Comme ce module a été décrit précédemment **[GLMF138]**, les explications sur Mojolicious seront rapides.

1.1 Mise en place d'un micro serveur avec Mojolicious

Pour démarrer plus facilement, on va créer un petit script avec **Mojolicious::Lite** qui sera chargé de lire les informations de la base de données. On verra plus tard comment mettre à jour ces données.

La commande suivante va créer un embryon de ce script :

```
$ mojo generate lite_app rapport.pl
$ cat rapport.pl
#!/usr/bin/env perl
use Mojolicious::Lite;

# Documentation browser under "/perldoc"
plugin 'PODRenderer';

get '/' => sub {
  my $self = shift;
  $self->render('index');
};

app->start;
__DATA__

@@ index.html.ep
% layout 'default';
% title 'Welcome';
Welcome to the Mojolicious real-time web
framework!

@@ layouts/default.html.ep
<!DOCTYPE html>
<html>
  <head><title><%= title %></title></head>
  <body><%= content %></body>
</html>
```

En résumé, la première partie de ce script déclare comment traiter la route **/** avec le template **index**. Celui-ci est décrit dans le script après la ligne **__DATA__**. Si ce résumé vous laisse perplexe, je vous invite à lire **[GLMF138]**.

Pour tester le résultat, lancez le mini-démon :

```
$ ./rapport.pl daemon
[Fri Jun  1 16:24:43 2012] [info] Listening at
"http://*:3000".
Server available at http://127.0.0.1:3000.
```

Et passez l'URL indiquée à votre butineur favori. Vous y verrez affichée une petite page d'accueil Mojolicious. En bonus, grâce au plugin **Podrenderer**, vous pouvez afficher des pages de doc de Perl sous l'URL http://localhost:3000/perldoc/. Par exemple, http://localhost:3000/perldoc/perlre.

Vous pouvez aussi lancer le script avec :

```
$ morbo rapport.pl daemon
```

morbo surveille les modifications du fichier **rapport.pl** et va le recharger si nécessaire. C'est très pratique dans les phases de développement.

2 Extraction et formatage des informations de la base

Revenons-en à nos produits. On a d'abord besoin d'afficher la liste de produits disponibles sous forme de table générée à partir du contenu de la base de données. Cette section va détailler la construction du script Mojolicious.

2.1 Utilisation des templates Mojolicious pour la liste des produits

D'abord, il faut se connecter à la base :

```
use Mojolicious::Lite;
use Integ::Schema ;

my $schema = Integ::Schema->connect(
   'DBI:mysql:database=Integ;host=localhost;port=3306',
   'integ_user', '', { RaiseError => 1 }
);
```

Ensuite, on gère la route **/** pour que l'URL http://localhost:3000 renvoie la liste des produits. **render** va utiliser le template **product_list** pour générer le code HTML. Le reste des paramètres est un ensemble de clés-valeurs utilisables dans le template :

```
get '/' => sub {
   my $self = shift;
   $self->render(product_list => rs => $schema-> resultset('Product') );
};
```

Cette dernière ligne démarre effectivement le daemon :

```
app->start;
```

Avec **Mojolicious::Lite**, les templates sont situés dans la section data (après la balise **__DATA__**). La syntaxe du template est détaillée dans **[GLMF138]**. En résumé :

- Les balises **<% ... %>** et les lignes commençant par **%** sont du code Perl ;
- Les balises **<%= ... %>** sont du code Perl dont la sortie est injectée dans le template, après une transformation pour le rendre moins lisible, mais compatible avec XML (e.g. **>** est changé en **>**) ;
- les balises **<%== ... %>** sont du code Perl dont la sortie est injectée verbatim dans le template. Ce qui est indispensable si le code Perl génère lui-même du HTML.

Le code suivant (en zone **DATA**) va utiliser le « layout » défini par la commande **mojo generate** et renvoyer une page HTML contenant un tableau avec les listes des produits disponibles.

```
__DATA__

@@ product_list.html.ep
% layout 'default' ;
% title 'Product list' ;
<h1>Produits</h1>
<table>
   <tr>
       <th>Produit</th>
       <th>Home page</th>
   </tr>
% while (my $product = $rs->next) {
%   my $name = $product-> name ;
   <tr>
       <td><a href="<%==$name%>"><%== $name %></td>
       <td><a href="<%== $product->home_page%>">home page</td>
   </tr>
% }
</table>
```

Notez que la première URL du tableau est construite sous la forme **<nom_du_produit>**. Il faudra définir une route dans le script Mojolicious pour répondre à cette adresse.

Pour tester, il faut relancer le mini-serveur (sauf si **morbo** s'en charge) et recharger la page web. Et là, horreur, cette page est moche : les informations sont là, mais en noir sur fond blanc.

Pour remédier à cette faute de goût, il suffit d'ajouter une feuille de styles dans le « layout » :

```
@@ layouts/default.html.ep
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link href="/css/my_style.css" rel="stylesheet" />
  </head>
  <body><%= content %></body>
</html>
```

Plutôt que d'ajouter une route dans la section **DATA**, on peut placer le fichier CSS dans le répertoire **public/css**, **Mojolicious::Lite** saura aller le chercher à cet endroit.

2.2 Mais c'était quoi cette horreur ?

Vous aurez sans doute remarqué que le template **product_list.html.ep** du paragraphe précédent n'est pas forcément agréable à lire : c'est un pot-pourri indigeste de balises HTML et de balises de template, persillé avec du code Perl.

C'est une limite des templates : quelques variables Perl de-ci de-là ne posent pas de problème, mais ça devient illisible dès que le traitement métier devient important par rapport au texte.

La philosophie de Perl étant d'avoir plus d'une façon de faire une chose **[TIMTOWTDI]**, on va utiliser pour les autres routes une façon alternative. Tous ces traitements complexes seront écrits principalement avec du code Perl et le code HTML sera généré avec le module qui va bien. Par exemple, le module **HTML::Tiny**.

2.3 Une petite digression sur HTML::Tiny

HTML::Tiny est un module bien pratique pour générer de l'HTML. Il fournit une méthode pour chaque balise HTML. Chaque méthode renverra une liste ou une chaîne, selon le contexte.

Par exemple, on peut avoir :

```
my $h = HTML::Tiny->new;
my $html_text = '';
$html_text .= $h->h1('Une petite digression...') ;
```

Les attributs d'une balise peuvent être passés dans un hash ref :

```
$html_text .= $h->div({class => 'perl_
module'},'HTML::Tiny' ) ;
```

Chaque paramètre va générer une balise :

```
say $h->td(qw/foo bar baz/) ;
# <td>foo</td><td>bar</td><td>baz</td>
```

On peut aussi avoir une balise pour une liste en passant une référence :

```
my @chanson = map {"$_ km à pied, ça use, ça
use... "} qw/un deux trois/ ;
my @li_en_vrac =  $h->li( @chanson ) ;
say $h->ul( \@li_en_vrac ) ;
```

Ce qui donne, après reformatage manuel :

```
<ul>
  <li>un km à pied, ça use, ça use... </li>
  <li>deux km à pied, ça use, ça use... </li>
  <li>trois km à pied, ça use, ça use... </li>
</ul>
```

Si une balise n'est pas implémentée dans **HTML::Tiny**, la méthode **tag** devra être utilisée :

```
say $h->tag('titi', qw/foo bar baz/),"\n";'
# <titi>foo</titi><titi>bar</titi><titi>baz</titi>
```

2.4 Générer la liste des versions

Le principe est le même que pour générer la liste de produits. Plutôt que d'utiliser les templates Mojolicious pour écrire la table, on va utiliser **HTML::Tiny**.

Tout d'abord, on crée un objet **HTML::Tiny** une fois pour toutes et on déclare la route. Celle-ci est un peu plus compliquée, car elle contient le nom du produit sous la forme **:name**.

```
my $h = HTML::Tiny->new;

get '/:name' => sub {
   my $self = shift;
   my $p_rs = $schema -> resultset('Product');
```

On récupère le nom du produit avec la méthode **param** et on recherche dans la base de données les infos pour les récupérer dans un objet **Schema::Product** :

```
   my $n = $self->param('name') ;
   my $p_obj = $p_rs->search({name => $n }) ->single ;
```

Et on construit le tableau HTML en faisant une itération sur les objets **Schema::ProductVersion** renvoyés par **$p_obj->product_versions->all** :

```
   my @rows ;
   foreach my $version_obj ($p_obj->product_versions-
>all) {
       my $version_as_string = $version_obj->version ;
```

Le lien vers la page dédiée à la version est créé avec la méthode **a**. Cette URL sera de la forme **/<nom_du_produit>/<version>** :

```
       my $v_link = $h->a({href => $version_as_
string }, $version_as_string) ;
```

Les lignes du tableau sont créées avec :

```
       my @data = $h->td( $v_link , $version_
obj->date->ymd ) ;
       push @rows, $h->tr( \@data ) ;
   }
```

Et enfin, on passe le tableau HTML sous forme de string au template **version_list** :

```
   $self->render ( version_list => rows =>
join("\n",@rows), name => $n ) ;
 } ;
```

Comme tout le boulot est fait avec **HTML::Tiny**, le template **version_list** devient beaucoup plus simple :

```
@@ version_list.html.ep
% layout 'default' ;
% title 'Product version list' ;
<h1>Versions du produit <%== $name %></h1>
<table>
    <tr>
        <th>Version</th>
        <th>Date</th>
    </tr>
    <%== $rows %>
</table>
```

Le script correspondant est disponible sur mon dépôt git **[GITHUBADOD]** sous **rapport.pl**.

3 Modification des données avec Mojolicious

Pour assurer le suivi de la qualification du produit, on va ajouter un formulaire web pour pouvoir modifier facilement le statut d'une version d'un produit. Pour cet article, ce statut est directement tiré des pratiques Debian, soit « unstable », « testing » ou « unstable ».

Le but est de créer un formulaire qui permettra de modifier le statut d'une version de produit.

D'abord, il faut une route pour lister les versions d'un produit. Chaque version contient un lien qui pointe vers une page d'édition. Cette route se contente de récupérer l'objet produit de la base pour le passer au template :

```
get '/:name' => sub {
   my $self = shift;

   # on récupère l'objet dans la base
   my $p_obj = $schema
      -> resultset('Product')
      -> find({name => $self->stash('name')}) ,

   # et on le passe au template
   $self->render (
       template => 'mojo_prod_v_list',
       p => $p_obj
   ) ;
} ;
```

Le template en question est assez simple :

```
@@mojo_prod_v_list.html.ep
% layout 'default' ;
% title 'Product versions' ;
<h1>Versions of product <%== $p->name %></h1>
<ul>
%  foreach my $r ($p->product_versions) {
%     my $v = $r->version ;
      <li><a href="<%== $p->name.'/'.$v %>"><%== $v %></a>
% }
</ul>
```

Ensuite, on crée la route pour éditer une version du produit. Comme le numéro de version contient un **.**, il faut utiliser **#version** au lieu de **:version**.

Complétez votre collection d'anciens numéros !

boutique.ed-diamond.com

numerique.ed-diamond.com

Les boutons sont construits ici pour simplifier le traitement en Perl dans le template :

```
get '/:name/#version' => sub {
   my $self = shift;
```

On récupère l'objet **version** du produit dans la base :

```
   my $pv = $schema->resultset('Product')
       -> find({name => $self->stash('name')})
       -> product_versions
       -> find ({version => $self->stash('version')}) ;
```

Ensuite, on crée les boutons radio actifs avec cette séquence un peu lourde (où **$h** est un objet **HTML::Tiny**). L'attribut **check** est calculé à la volée pour que le bouton affiche la valeur courante dans la base :

```
    my @radio_items = map {
         $h->label($_)
       . $h->input({
           type => "radio",
           name => "v_status",
           value => "$_",
           ($_ eq $pv->status) ? ( checked => 'checked') : ()
       });
    } qw/unstable testing stable/ ;
```

Enfin, on envoie le HTML pré-mâché au template :

```
   $self->render (
       template => 'mojo_template_mod',
       radio_b => \@radio_items,
       pv => $pv ,
   ) ;
};
```

Le template associé contient un formulaire. Pour se simplifier la vie, le bouton de sauvegarde va construire une URL avec l'id de la version plutôt qu'avec le couple « nom/version » :

```
@@mojo_template_mod.html.ep
% layout 'default' ;
% title 'Product status' ;
<h1>Product <%== $pv->product->name %> version <%== $pv->version %></h1>

<form name="save_data"
      action="/product_version_data_save/<%= $pv->id %>"
      method="post">
status :
   <%== join("\n", @$radio_b) ; %>
   <input type="submit"
          name="save-product-version"
          value="Save"/>
</form>
```

Et on obtient ce résultat, certes moche, mais fonctionnel :

Une fois le bouton « Save » pressé, le navigateur va envoyer une requête **POST** vers le serveur Mojolicious. Pour que cette requête puisse fonctionner, il faut la route associée.

Celle-ci va récupérer l'objet correspondant à la version modifiée et sauver le paramètre modifié dans la base.

```
post 'product_version_data_save/:vid' => sub {
   my $self = shift ;

   my $v_obj = $integ_schema
       -> resultset('ProductVersion')
       -> find({ id => $self->stash('vid')}) ;

   # on extrait l'information des paramètres du POST
   my $value = $self->param('v_status') ;
   # on sauve dans l'objet version
   $v_obj->status($value) ;
   # enfin, on envoie la donnée dans la base
   $v_obj->update ;

   # le gros du boulot est fait, il reste à informer l'utilisateur
   $self->render(
       'product_version_save_done',
       p => $v_obj->product,
       v => $v_obj->version
   );
} ;
```

Voici son template :

```
@@ product_version_save_done.html.ep
%title 'product '.$p->name . ' saved ' ;
%layout 'redirect', redirect_url => '/' ;
<p>Product <%= $p->name %> version <%== $v %> was saved</p>
<p><a href="/">go back to product list</a></p>
```

Au moins, ça fonctionne. Mais le code est déjà assez compliqué et les données ne sont pas vérifiées avant d'être envoyées dans la base (**modify-a-la-mojo.pl** dans **[GITHUBADOD]**).

Pour des formulaires plus compliqués que des boutons radio, il faut en plus s'assurer que les valeurs rentrées par l'utilisateur soient cohérentes. On peut toujours ajouter du code de validation dans les contrôleurs (c'est-à-dire dans les procédures associées aux routes), mais le mélange de la validation avec la génération des formulaires va compliquer la maintenance. L'idéal serait de pouvoir séparer la validation de la gestion des templates et des données.

Ben, justement, c'est ce que proposent les modules **HTML::FormHandler** et **HTML::FormHandler::Model::DBIC**.

4 Modification des données avec HTML::FormHandler

HTML::FormHandler est un module impressionnant qui permet de créer un formulaire HTML avec une déclaration de classe Perl en format Moose (ou presque). Cette classe est utilisée par **HTML::FormHandler** pour :

- Créer les formulaires HTML,
- Traiter et valider les paramètres envoyés par l'utilisateur avec une requête **POST**,
- Sauvegarder ces valeurs dans une base de données si **HTML::FormHandler::Model::DBIC** est aussi utilisé.

Je vous propose de faire un formulaire comprenant la fonction précédente (modification du statut) et l'ajout d'un log (pour montrer comment déclarer un champ avec une validation).

Tout d'abord, il faut créer une classe par formulaire. Cette classe doit utiliser **HTML::FormHandler::Moose** au lieu de Moose. Ça permet de déclarer les champs du formulaire avec **has_field**. Elle doit aussi hériter (déclaration avec **extends**) de **HTML::FormHandler::Model::DBIC** pour créer un formulaire dédié à une table de base de données. Comme cette classe va être incluse dans le fichier avec les routes de **Mojolicious::Lite**, on va l'englober entre deux accolades :

```
{
   package ProductVersionForm;
   use HTML::FormHandler::Moose;
   extends 'HTML::FormHandler::Model::DBIC';
   use namespace::autoclean;
```

Il faut ensuite déclarer quelle table va être utilisée pour sauvegarder les valeurs du formulaire. Il ne faut pas oublier le **+**, car cet attribut surcharge celui de la classe **HTML::FormHandler**.

```
   has '+item_class' => ( default => 'ProductVersion' );
```

Ensuite, on peut déclarer les champs du formulaire.

status est un champ à plusieurs valeurs possibles (type **select**), où une seule valeur doit être choisie. Le plus simple est d'en faire un ensemble de boutons de type radio avec **widget RadioGroup**. Les informations passées avec options sont utilisées telles quelles dans la balise **input**. Chaque statut possible doit avoir une déclaration **label** et **value** pour que le champ soit affiché correctement.

```
   has_field 'status' => (
      type => 'Select',
      widget => 'RadioGroup',
      options => [
         map { { label => $_ , value => $_ } ;} qw/unstable testing stable/
      ],
   );
```

Maintenant, on va s'occuper du champ **log**. Pour emb^W inciter les utilisateurs à motiver le changement de statut, on rend le champ obligatoire avec **required**. Le paramètre **apply** permet de déclarer des contraintes de validation arbitraire (voire farfelues dans cet exemple). **apply** spécifie une condition à appliquer (une « subref » qui renvoie vrai ou faux) et un message d'erreur.

```
   has_field 'log' => (
      # utiliser TextArea pour une zone d'édition plus grande
      type => 'Text',
      required => 1,
      apply => [
         {
            check => \&whatever,
            message => 'whatever rule was not satisfied'
         }
      ]
   );

   sub whatever {
      my ( $value, $field ) = @_;
      return $value =~ /whatever/ ? 1 : 0 ; # ok, c'est idiot
   }
```

Enfin, le dernier champ spécifie l'indispensable bouton pour envoyer le formulaire :

```
    has_field 'submit' => ( type => 'Submit', value => 'Save' );
```

Ces 2 dernières lignes sont des optimisations de performance :

```
    __PACKAGE__->meta->make_immutable;
    no HTML::FormHandler::Moose;
}
```

Maintenant, il faut revoir les contrôleurs. Les seuls qui changent sont ceux des routes **/:name/#version** et **/product_version_data_save/:vid**. Voici la première route qui va générer le formulaire :

```
get '/:name/#version' => sub {
    my $self = shift;

    my $product_version = $integ_schema
        ->resultset('Product')
        ->find( { name => $self->stash('name') } )
        ->product_versions
        ->find( { version => $self->stash('version') } );

    my $form = ProductVersionForm->new(
        action => "/product_version_data_save/".$product_version->id,
        item => $product_version,
    ) ;

    $self->render(
        template => 'formhandler_template_mod',
        my_form  => $form ,
        pv       => $pv,
    );
};
```

Il faut d'abord trouver dans la base l'objet version de produit (stocké dans **$product_version**) pour pouvoir afficher les valeurs courantes dans le formulaire. L'objet **$form** contient une instance de l'objet **HTML::FormHandler**. Le paramètre **action** spécifie où envoyer la requête **POST**. Le paramètre **item** doit contenir l'objet qui représente une ligne (« row ») dans la table **ProductVersion**. Enfin, l'appel à **render** utilise le template suivant :

```
@@formhandler_template_mod.html.ep
% layout 'default' ;
% title 'Product status' ;
<h1>Product <%== $pv->product->name %> version <%== $pv->version %></h1>
<%== $my_form->render%>
```

Celui-ci est très simple : il suffit d'appeler la méthode **render** sur l'objet formulaire pour créer tout le code HTML et produire cette page :

Et voici le contrôleur pour sauver les données :

```
post '/product_version_data_save/:vid' => sub {
    my $self = shift;

    my $form = ProductVersionForm->new;

    # ne pas oublier de traiter les paramètres
    $form->process(
        schema => $integ_schema ,
        item_id => $self->stash('vid') ,
        params => $self->req->params->to_hash
    ) ;

    if ($form->has_errors) {
        return $self->render(text => join("\n", $form->errors )) ;
    }

    $self->render(
        'product_version_save_done',
    );
};
```

Il faut aussi recréer l'objet **HTML::FormHandler** formulaire (dans **$form**). L'appel à **process** contient le schéma et l'id de l'objet version de produit passé en paramètre du **POST**. **HTML::FormHandler** n'a pas besoin de plus pour savoir comment sauver les autres paramètres du **POST** contenus dans **$self->req->params>**. Le template appelé est très simple :

```
@@ product_version_save_done.html.ep
%title 'product  saved ' ;
%layout 'redirect', redirect_url => '/' ;
<p>Product status was saved</p>
<p><a href="/">go back to product list</a></p>
```

Ce qui renvoie sur la page principale du serveur.

Ce web serveur est disponible dans **modify-a-la-formhandler.pl** sur le dépôt git **[GITHUBADOD]**.

Conclusion

Nous voici arrivés à la fin de la trilogie. Vous avez le minimum pour pouvoir créer une petite application web avec une base de données relationnelle. N'hésitez pas à utiliser les codes sur le dépôt git **[GITHUBADOD]**, ceux-ci sont disponibles en licence Artistic ou GPL-1+.

Dans votre projet, vous vous rendrez compte que ces articles ne montrent que le sommet de l'iceberg. Les modules **DBIX::Class**, **Mojolicious** et **HTML::FormHandler** ont beaucoup plus de possibilités. Il faudrait, non pas 3 articles pour les couvrir, mais 3 volumes. Heureusement, ces modules ont tous une documentation très complète sur CPAN. N'hésitez pas à la consulter ! ■

Remerciements

Les Mongueurs de Perl pour leur accueil et la relecture de cet article.

Liens

[GLMF138] « Développement web en Perl avec Mojolicious » GNU/Linux Magazine n° 138

[TIMTOWTDI] http://en.wikipedia.org/wiki/There%27s_more_than_one_way_to_do_it

[GITHUBADOD] https://github.com/dod38fr/glmf-article-dbix-class-web

DART : LA PLATEFORME ORIENTÉE WEB PAR GOOGLE

par Sylvain Saurel [Ingénieur d'études Java / Java EE]

Acteur majeur du Web depuis plus d'une décennie, Google ne cesse de travailler à proposer des solutions visant à rendre le Web plus rapide et plus riche. Toujours prompt à faire bouger les lignes, le géant de Mountain View est à la pointe de l'innovation et nous le prouve une fois de plus avec sa plateforme Dart qui vient de fêter sa version 1.0. Indissociable de JavaScript dont elle assure la compatibilité tout en comblant ses lacunes, Dart offre un environnement moderne et efficace pour réaliser des applications web riches. Plongée au cœur d'une technologie prometteuse combinant performance et productivité.

Révélée au grand public en octobre 2011, la plateforme Dart se base sur le langage éponyme. Bien souvent d'ailleurs, beaucoup pensent que Dart se limite à un langage, mais il constitue en réalité un véritable écosystème applicatif ayant pour but de fournir une alternative aux limites de JavaScript. Plutôt que de chercher à faire évoluer le langage JavaScript pour combler ses lacunes, les ingénieurs de chez Google ont opté pour une solution nouvelle avec Dart, tout en assurant la pleine compatibilité avec JavaScript puisque la plateforme propose un compilateur de code vers ce dernier.

À l'origine de la création de Dart, un constat assez simple. Véritable assembleur du Web, le langage JavaScript pose un certain nombre de problèmes tant au niveau de sa syntaxe (typage dynamique, langage prototypé, ...) que de son implémentation différente au sein des navigateurs. En outre, les performances ne sont pas toujours au rendez-vous du fait d'une mauvaise utilisation du langage. En effet, de par ses spécificités, le JavaScript nécessite une forte maîtrise de la part des développeurs pour construire des applications web à la fois rapides, sûres et riches. Trouver de tels développeurs est souvent ardu, ce qui amène à la réalisation d'applications web de piètre qualité. Enfin, les outils de développement autour de JavaScript ne sont pas de qualité suffisante, ce qui ralentit la productivité des développements.

Parmi les autres acteurs du Web ayant fait ces constats, Microsoft a pris une voie différente avec TypeScript, qui s'appuie sur la syntaxe JavaScript en proposant un sur-ensemble au langage. A contrario, Google a donc préféré partir sur un nouveau langage pour corriger au mieux les lacunes de JavaScript. C'est dans ce contexte qu'est né Dart qui se veut être, selon ses concepteurs, tout ce que JavaScript aurait dû être s'il avait été inventé aujourd'hui. En bref, il tente de garder un côté dynamique dans un cadre plus strict, pour offrir des outils de qualité permettant de réaliser des applications web complexes et pérennes.

1 Architecture

Un programme Dart est exécuté au sein d'une machine virtuelle dédiée utilisable en standalone, ou bien embarquée dans un navigateur exécutant directement le code Dart. Le SDK s'amène d'ailleurs avec Dartium, une implémentation spécifique de Chromium intégrant la machine virtuelle Dart. Excepté Chrome, qui prévoit d'intégrer cette machine virtuelle dans un avenir proche, la plupart des navigateurs ne supportent pas Dart. Pour contourner ce potentiel problème, le SDK propose un compilateur Dart vers JavaScript compilant un programme Dart en un JavaScript équivalent, compatible avec les normes ECMAScript 5 et HTML 5.

Le code produit fonctionne sur l'ensemble des navigateurs récents : Chrome, Safari, Firefox, IE 9 et IE 10, ainsi que les versions mobiles de Safari et de Chrome. Ceci offre l'avantage d'abstraire le développeur du travail de compatibilité inter navigateurs.

L'approche retenue pour l'exécution d'un programme Dart (figure 1) consiste à proposer le code Dart et son équivalent JavaScript, tout en testant à l'exécution la présence ou non d'une machine virtuelle Dart pour savoir quel code exécuter. Ce test se fait via le script **dart.js** mis à disposition par la plateforme.

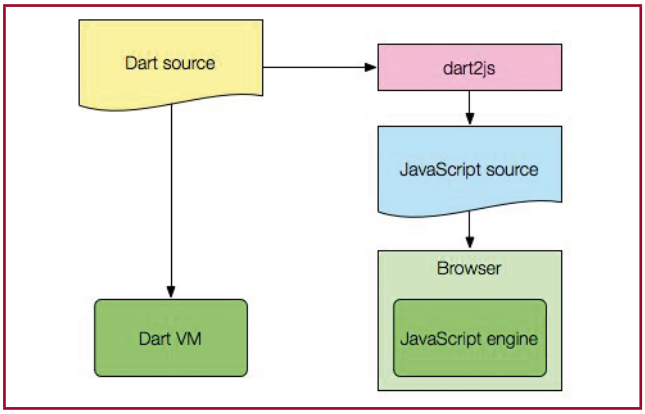


Fig. 1 : Possibilités d'exécution d'un code Dart

Le compilateur **dart2js** compile toutes les sources d'une application, y compris les bibliothèques utilisées, en un seul fichier JavaScript. Juste avant cette compilation, **dart2js** optimise le code source, afin de réduire sa taille, en éliminant le code mort via le *Tree Shaking*, à l'instar de ce que fait le framework GWT par exemple.

Le modèle d'exécution de Dart n'engage qu'un seul *thread*, mais autorise néanmoins un mécanisme de concurrence simple. En outre, la gestion des dépendances n'est pas oubliée, avec l'outil **pub** permettant de découvrir de nouvelles bibliothèques, de les installer et des les intégrer tout en gérant leur versioning. Enfin, la machine virtuelle possède 2 modes d'exécution. Plus rapide, le mode *production* est activé par défaut et ne lève pas d'avertissements sur le code comme peut le faire le mode *checked*.

2 SDK et Outils

Récupérable ici : http://www.dartlang.org/tools/sdk/, le SDK Dart met à disposition les outils nécessaires pour la création et le développement d'applications web de qualité (figure 2) proposant ainsi :

- La machine virtuelle Dart,
- Le compilateur **dart2js**,
- Le gestionnaire de dépendances **pub**,
- Le navigateur Dartium embarquant une machine virtuelle Dart,
- Des outils pour la qualité des projets Dart (le générateur de documentation **dartdoc** et l'analyseur statique **dartanalyzer**).

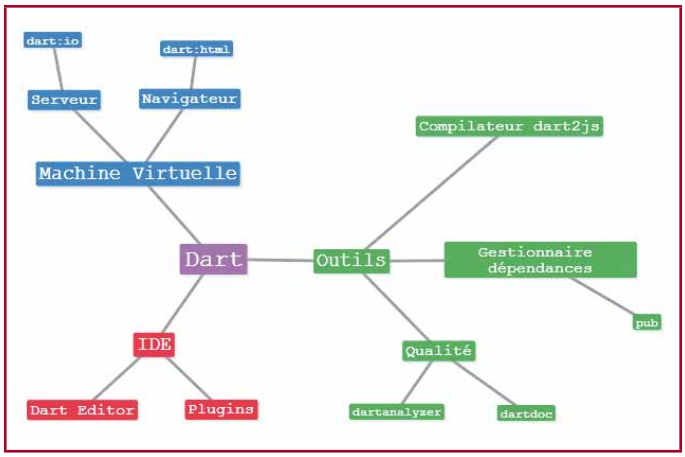


Fig. 2 : Plateforme Dart

À côté du SDK, Google met à disposition l'IDE Dart Editor. Basé sur Eclipse, il offre les outils de base pour développer des applications web avec Dart. On retrouve ainsi la complétion du code, l'analyse de code, la navigation au sein d'un projet, le débuggage, ainsi que l'exécution du code depuis l'IDE. En sus, des plugins pour Eclipse et IntelliJ sont proposés.

3 Bibliothèques

Le SDK Dart comporte près d'une vingtaine de bibliothèques standards couvrant les besoins de base dans la réalisation d'applications web, mais également celles pour le développement côté serveur parmi lesquelles on retrouve :

- **dart:core**, incluse par défaut, contient l'ensemble des types et des structures de données de base,
- **dart:html**, qui est la porte d'entrée pour accéder au DOM HTML 5,
- **dart:io** réservée au développement d'applications serveur avec le support des entrée/sortie et un serveur HTTP,
- **dart:math** pour les fonctions mathématiques basiques,
- **dart:json** pour travailler avec le format JSON,
- **dart:utf** pour la gestion de l'Unicode,
- **dart:mirros** propose des fonctionnalités de réflexion,
- **dart:isolate** supportant une concurrence simple et sûre,
- **dart:async** pour des opérations asynchrones avec notamment le concept des Futures.

En plus de ces composants standards, de nombreuses autres bibliothèques sont accessibles via le gestionnaire de dépendances **pub**. On peut ainsi citer *JavaScript Interop* pour utiliser du JavaScript directement au sein de Dart, *Web UI* pour construire des applications web basées sur les composants web et *Unit Test* pour l'écriture de tests unitaires.

4 Langage

Conçu pour pallier les défauts de JavaScript, Dart est un langage de script web dynamique pouvant être compilé en JavaScript ou exécuté au sein d'une machine virtuelle dédiée. Pleinement orienté objet, il repose sur des classes concrètes et abstraites avec un héritage simple. Dérivant du C, sa syntaxe se rapproche d'autres de ses descendants tels que Java et C#. Supportant également les interfaces et les génériques, il propose un système de typage optionnel ignoré à l'exécution et qui sert à déclencher des avertissements en phase de développement. D'autre part, Dart supporte les fonctions de haut niveau considérées comme des objets à part entière et donc, utilisables en tant qu'arguments en entrée de fonctions ou méthodes au sein d'une classe. En sus, le concept de *mixins* autorise des constructions puissantes. Un modèle de concurrence simple est supporté via les *Isolates* implémentés sous forme de *Web Workers* lors du passage en JavaScript. Au niveau des différences avec ce dernier, on peut citer l'absence des prototypes et l'impossibilité d'exécuter du code à la volée. Enfin, un programme Dart requiert la présence d'une fonction de haut niveau *main* appelée à son lancement.

De par sa conception, le langage Dart et plus encore la plateforme autorisent la construction d'applications web allant du simple script au projet complexe avec une grande efficacité. Ses concepts issus de langages répandus le rendent facile à prendre en main et le fait qu'il puisse être employé côté client et côté serveur lui confère des atouts considérables dans la perspective de réaliser des applications web via un seul langage.

5 Types

Tous les types Dart sont des objets dérivant de la classe **Object**. Le mot-clé **var** sert à déclarer une variable sans définir de type. Comme en Java, la méthode **toString()** d'un objet est appelée par défaut lorsque celui-ci est converti en **String** comme avec la fonction **print** :

```
var myObj = 2013;
// équivalent
print(myObj);
print(myObject.toString());
```

Dans l'idéal, l'absence de typage doit se limiter à des portions de code internes. Pour représenter des numériques, Dart propose le type général **num** et les implémentations spécialisées **int** et **double**. Bien qu'il s'agisse d'objets, il n'est pas nécessaire de les instancier via **new** :

```
num myNum = 12.4;
num myRounded = myNum.round();
int myInt = 12;
// possible
var myInt = 123;
var intAsHex = 0xD;
double myDouble = 12.4;
var doubleWithExp = 1.24e3;
```

Les objets numériques proposent les opérations de calculs **+-/*** qu'il est possible de surcharger, ainsi que des fonctions classiques comme **abs**, **ceil**, **floor** et **round**. Contrairement à JavaScript, Dart considère uniquement le littéral booléen **true** comme valeur vraie. Toutes les autres valeurs sont donc fausses. En mode checked, les opérations booléennes sont autorisées uniquement sur des variables de type **bool** :

```
bool myBool = true;
var myBool = true;
bool myFalse = false;
print(myBool != myFalse); // true
print(myBool && myFalse); // false
print(myBool || myFalse); // true
print("Cond is : " is String); // true
```

On remarque l'opérateur **is** pour tester si une instance est d'un type ou non. Élément indispensable, les chaînes de caractères sont implémentées au sein du type **String**. Leur principale spécificité concerne la concaténation, qui s'appuie sur l'interpolation et non l'opérateur **+** :

```
String hello = "Hello";
var name = "Sylvain";
var concat = "$hello $name";
print(concat); // Hello Sylvain
```

Les variables définies au sein d'une interpolation sont évaluées, ce qui autorise des opérations en se servant de **${...}** :

```
var test = "${1+2.5.round()}";
print(test); // 3
```

6 Structures de données

Au niveau structures de données, Dart n'a pas de type tableau, mais propose le type **List** instanciable à partir de la syntaxe raccourcie, ou via son constructeur :

```
List myList = []; // vide
var myList = [];
var myList = [1, 2, 3];
var myList = [1, "val 2", 3];
// constructeur
List myList = new List();
var myList = new List(10); // 10 elts null
var myList = new List.filled(10, "SS"); // 10 "SS"
```

L'ajout d'éléments se fait via la méthode **add()**, alors que l'accès à un élément de la liste passe par l'utilisation des crochets **[ ]**. En outre, il est bon de souligner que les listes sont indexées sur le zéro :

```
myList.add("Elt1");
myList.add("2");
print(myList[1]); // 2
myList[1] = "Elt2";
```

Le langage supportant les génériques, il est possible de typer une liste pour préciser le type d'élément qu'elle peut contenir. L'ajout d'un objet de type différent lèvera ensuite un avertissement de type :

```
List<String> myList = new List<String>();
myList.add(123); // avertissement 123 non String
myList = <String>["a", "b"];
```

Le parcours des éléments d'une liste peut se faire de manière classique en accédant aux éléments via leur index, soit en utilisant une boucle **for-in**, ou bien en appelant la méthode **forEach** de l'objet **List** :

```
for(var i = 0; I < myList.length; i++) {
 var val = myList[i];
}

for(var item in myList) {
 print(item);
}
myList.forEach((item) {
 print(item);
});
```

La bibliothèque standard propose également des implémentations spécialisées pour les collections HashSet, Queue et Set. D'autre part, le langage supporte les maps, qui sont des listes de couples clé/valeur, la création d'une map se faisant via la syntaxe raccourcie ou via son constructeur. Les génériques restent applicables et l'accession aux éléments d'une map se fait via les crochets **[ ]**. Enfin, le parcours d'une map se réalise via une boucle **for** ou via la méthode **forEach** :

```
Map m = {};
Map<String, String> map = {"A":"1", "B":"2", "C":"3"};
// ajout
map["D"] = "4";
print(map.containsKey("D")); // true

for (var keys in map.keys){
 print(keys);
}

for (var val in map.values){
 print(val);
}

map.forEach((k,v) {
 print("$k => $v");
});
```

7 Fonctions

Les fonctions peuvent être de haut niveau, ou bien définies au sein d'une classe, auquel cas on parle de méthodes. En sus, des facilités syntaxiques permettent de raccourcir l'écriture de fonctions :

```
sum(num val1, num val2) => val1 + val2;
printMsg(msg) => print("Msg : $msg"); // renvoie null
// équivalent
num sum(num val1, num val2) {
 return val1 + val2;
}
```

Le typage d'une fonction est optionnel tant au niveau des entrées que de sa sortie. Définir les types attendus reste néanmoins une bonne pratique permettant aux outils Dart de travailler efficacement. Les paramètres d'entrée peuvent être obligatoires ou optionnels. Dans tous les cas, les paramètres obligatoires sont définis en premier. Alors que ces derniers doivent être passés en respectant l'ordre de définition, les paramètres optionnels peuvent être définis associés à leur nom également :

```
void optParams(String name, [int version = 1, url]) { ... }
void optNamed(String name, {int version:1, url}) { ... }
optParams("name");
optParams("name", 1);
optNamed("name", url:"www.programmez.com");
```

Comme pour les autres types, une fonction peut être affectée à une variable, ce qui implique que l'on peut passer une fonction en entrée ou en sortie d'une autre fonction :

```
num getResult(num val1, num val2, Function calcFunc) {
 return calcFunc(val1, val2);
}
var add = (a,b) => a+b;
var sub = (a,b) => a-b;

var res1 = getResult(1,2,add); // 3
var res2 = getResult(10,5,sub); // 5
```

Afin de définir une fonction d'un type donné et d'éviter d'utiliser le type générique **Function**, il faut recourir au mot-clé **typedef** :

```
typedef CalcFunction(num value1, num value2);
```

Cette définition permet de lever des avertissements à l'exécution si le type de fonction passé en entrée ne correspond pas au type attendu.

8 Classes / Interfaces

Le modèle de classes retenu par Dart se rapproche grandement de ce qui existe en Java ou en C# notamment, ce qui permet aux développeurs venant de ces univers de ne pas être trop dépaysés. Une classe peut ainsi définir un constructeur, des méthodes, des propriétés, ainsi que des getters et setters. Ces derniers étant définis par défaut, mais pouvant être redéfinis le cas échéant :

```
class Person {
 // private
 String _firstName;
 String _lastName;
 int _age;
 // public
 String gender = MALE;
 static String MALE = "MALE";
 static String FEMALE = "FEMALE";

 Person(fname, lname, age, g) {
  _firstName = fname;
  _lastName = lname;
  _age = age;
```

```
  gender = g;
 }

 Person.fromJson(Map data) {
   // ...
 }

 int get age => _age;

 bool isMale() => this.gender == MALE;

 String toString() {
  return "$_firstName $_lastName - $_age";
 }
}
```

Le langage propose un héritage simple via le classique mot-clé **extends**, l'accès à l'objet parent se faisant via le mot-clé **super**. Les constructeurs ne sont pas hérités et le constructeur parent doit être appelé dans le bloc d'initialisation du constructeur :

```
class Employee extends Person {
 int _id;
 Employee(fname, lname, age, g) : super(fname, lname, age, g) { }
 Employee.fromJson(Map data) : super.fromJson(data) {
  // ...
 }
}
```

Pour définir une classe sans implémentation ou avec implémentation partielle, le langage propose les classes abstraites. Les classes en héritant devant définir les méthodes sans implémentation :

```
abstract class Person {
 // ...
 void fromJson(Map data);
}
class Employee extends Person {
 // ...
 void fromJson(Map data) {
 }
}
Person p = new Employee("Sylvain", "Saurel", 29, Person.MALE);
```

Le modèle de classes de Dart ne propose pas de mot-clé **interface**. Si le langage ne propose pas de définition explicite, c'est que chaque classe est en fait une interface. En effet, l'interface d'une classe est la représentation des membres publics de la classe. Quand une classe implémente une interface, elle doit fournir une implémentation pour tous les membres :

```
class Employee implements Person {
  // implémentation des membres publiques
 void fromJson(Map data);
}
```

Pour terminer, un mot sur les mixins que le langage propose depuis sa *milestone* 3. Ceux-ci permettent d'injecter des comportements au sein d'une classe sans recourir à l'héritage. L'ajout d'un comportement **Persistence** à la classe **Employee** s'effectuant de la sorte :

```
abstract class Persistence {
 void save(String filename) { }
 void load(String filename) { }
}

class Employee extends Person with Persistence {
 // ...
}
```

9 Premier programme

Un programme Dart contient obligatoirement une fonction de haut niveau **main** servant de point d'entrée lors de l'exécution du programme. Notre premier programme Dart met en œuvre les classes définies précédemment :

```
void main() {
 List<Person> persons = <Person>[];
 Person p = new Employee("Sylvain", "Saurel", 29, Person.MALE);
 persons.add(p);
 p = new Employee("Rachida", "Saurel", 26, Person.FEMALE);
 persons.add(p);

 persons.forEach((e) {
  print("$e - ${e.isMale() ? 'Male' : 'Female'}");
 });
}
```

L'exécution de ce code au sein de l'IDE Dart Editor produit la sortie présentée en figure 3.

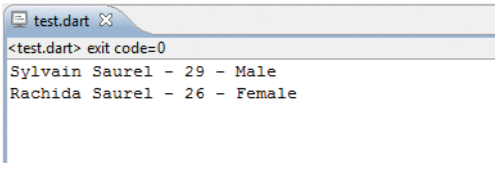


Fig. 3 : Premier programme Dart

Conclusion

Bien plus qu'un simple langage, Dart constitue une véritable plateforme de développement orientée Web. Proche du Java et du C#, sa syntaxe associée à un outillage de qualité favorise une prise en main rapide. Fonctionnant au sein d'une machine virtuelle dédiée, le code Dart est également compilable en JavaScript condition *sine qua non* à ses chances d'adoption par le plus grand nombre. Les nombreuses bibliothèques standards proposées, ainsi que le gestionnaire de dépendances intégré, attestent de la volonté de Google de faciliter le développement de projets complexes grâce à Dart. Enfin, son fonctionnement côté client et côté serveur offre de réelles perspectives pour réaliser des applications web avec un seul langage. C'est justement ce qui constituera l'objet de la seconde partie de cet article, au cours de laquelle nous étudierons la réalisation d'une application web basée sur Dart côté client et côté serveur. ■

DART : LA PLATEFORME ORIENTÉE WEB PAR GOOGLE (2ÈME PARTIE)

par Sylvain Saurel [Ingénieur d'études Java / Java EE]

La première partie de cet article consacré à Dart a permis de faire un tour d'horizon de la plateforme et de ses spécificités, tout en proposant une prise en main du langage. Plutôt théorique, cette introduction était le prérequis à la suite de l'article avec la mise en pratique sur un cas d'usage concret : la réalisation d'une application web utilisant Dart de bout en bout, côté client et côté serveur.

Alliant puissance et simplicité, Dart permet la réalisation d'applications web complexes avec un seul langage côté client et côté serveur, ouvrant des perspectives intéressantes pour le futur de Dart. L'IDE Dart Editor étant suffisamment avancé pour être productif en travaillant efficacement. En outre, le compilateur **dart2js** produit un code JavaScript optimisé et plus performant que le code écrit directement dans ce langage, avec une compatibilité inter navigateurs garantie. L'utilisation de ce dernier étant la seule solution viable actuellement pour le déploiement d'applications Dart, puisqu'aucun des navigateurs du marché n'embarque la machine virtuelle Dart en standard.

1 Cahier des charges

Le cahier des charges de l'application web que nous allons réaliser est assez simple. Il s'agira d'une application de type *todo list*, proposant une liste de tâches à réaliser. L'utilisateur pourra visualiser des tâches, en ajouter de nouvelles, ainsi qu'en supprimer, le tout au sein d'une page unique. En sus, les tâches à réaliser seront stockées de sorte qu'elles soient accessibles à plusieurs utilisateurs. Enfin, l'application devra être accessible via une URL au sein d'un navigateur.

2 Solution technique

Au niveau technique, le cahier des charges de l'application implique la présence d'un serveur HTTP pour servir les requêtes d'affichage, d'ajout et de suppression des tâches. Il sera implémenté au sein d'un script Dart utilisant la bibliothèque **dart:io** et lancé en ligne de commandes via l'exécutable **dart**. L'obligation de persister les tâches implique l'emploi d'une base de données côté serveur. Une solution souple et légère consiste à s'appuyer sur MongoDB, une base de données NoSQL orientée documents parfaitement scalable. En outre, elle présente l'insigne avantage d'avoir une bibliothèque cliente mise à disposition sur le dépôt http://pub.dartlang.org et d'utiliser JavaScript comme langage natif de requêtage. Récupérable dans un projet via le gestionnaire de dépendances **pub**, elle propose une API pour interagir facilement avec MongoDB. La communication client/serveur se fera en mode REST au format JSON.

Côté client, l'application se compose d'une seule page HTML associée à un script Dart au sein duquel seront réalisés les appels vers le serveur. Ces appels utiliseront l'objet **HttpRequest** du package **dart:html** implémenté via l'objet **XMLHttpRequest** lors du passage au JavaScript. L'interaction entre la page HTML et le code Dart mettra en évidence les possibilités de manipulation du DOM proposées en standard permettant de se passer de jQuery dans le domaine. Non obligatoire mais intéressant, le recours au package Web UI servira de support à la présentation des facilités de *templating* de Dart et des Web Components.

3 Base de données MongoDB

Actuellement en version 2.4, la base de données MongoDB est récupérable à cette adresse : http://www.mongodb.org/downloads. Afin de fonctionner, la

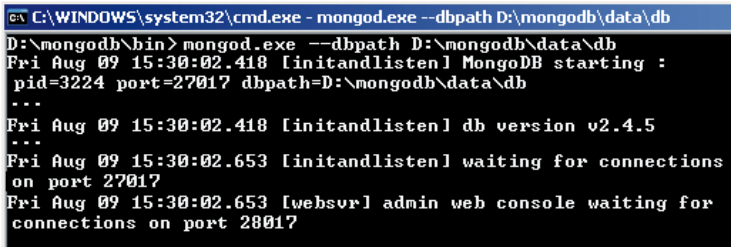


Fig. 1 : Lancement de MongoDB

base a besoin d'un dossier pour stocker ses fichiers. Par défaut, MongoDB pointe vers **C:\data\db** sous Windows et **/data/db** sous Linux. Il est possible toutefois de préciser au démarrage de MongoDB le répertoire de destination utilisé. Au sein du répertoire **bin**, on retrouve l'exécutable **mongodb** permettant le lancement de la base :

```
mongod --dbpath d:\mongodb\data\db
```

Une fois démarré, le serveur MongoDB est en écoute sur le port 27017 du localhost (figure 1).

4 Projet serveur

La réalisation de l'application se fait avec Dart Editor. La première étape est la création du projet contenant le script jouant le rôle de serveur HTTP. Pour ce faire, il faut aller dans le menu ***File > New Application*** et créer un projet de type « Command-line application » nommé **TodosServer**. La hiérarchie du projet ainsi créé (figure 2) montre la présence du descripteur de dépendances au format YAML nommé **pubspec.yaml** et du fichier **pubspec.lock** détaillant les dépendances effectives. Le répertoire **packages** liste les dépendances du projet avec leur version et leurs sources. Enfin, puisqu'il s'agit d'une application ligne de commandes, le script principal **todosserver.dart** se trouve au sein d'un dossier **bin**, mais cela n'est pas obligatoire.

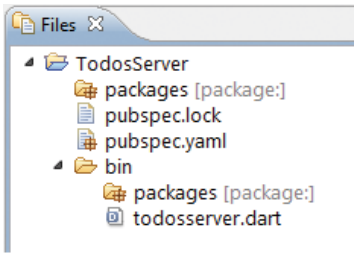


Fig. 2 : Hiérarchie du projet créé

5 Connexion à MongoDB

Les échanges avec la base MongoDB s'effectuent au travers du client **mongo_dart** présent sur le dépôt central Dart. L'ajout de cette bibliothèque se fait en ajoutant la dépendance suivante au fichier **pubspec.yaml** du projet :

```
mongo_dart: 0.1.27
```

L'IDE se charge alors de récupérer la bibliothèque avec ses dépendances et des les intégrer au projet via la commande **pub install**. Pour le serveur, nous créons une classe **TodosServer** initialisant la connexion à la base au sein de son constructeur, via la classe **Db** de **mongo_dart** :

```
import 'package:mongo_dart/mongo_dart.dart';

class TodosServer {
 DbCollection _todos;
 static const HOST = "127.0.0.1";
 static const PORT = 8080;
 static const TODOS_PATH = "/todos";
 static const ADD_PATH = "/addTodo";
 static const DELETE_PATH = "/deleteTodos";

 TodosServer() {
  Db db = new Db("mongodb://$HOST/todos-server");
  db.open().then((c) {
   _todos = db.collection("todos");
  });
 }
}
```

On remarque tout d'abord l'import du fichier Dart principal de **mongo_dart** grâce au mot-clé **import**. En sus, la bibliothèque propose une API basée sur des appels chaînés, ce qui permet d'ouvrir la connexion à la base pour ensuite créer la collection MongoDB stockant les todos.

6 Création du serveur

En proposant la classe **HttpServer** au sein du package **dart:io**, Dart facilite la création du serveur HTTP. La méthode **bind** de cet objet prend en entrée une adresse et un port à partir desquels elle se met en écoute des requêtes entrantes. La suite du travail consiste à utiliser la méthode **listen** du serveur pour gérer ces requêtes. Dans notre cas, une requête de type **GET** sur le chemin **/todos** renverra l'ensemble des tâches, alors qu'une requête **POST** sur le chemin **/addTodo** ajoutera une nouvelle tâche côté serveur et qu'une requête **POST** sur **/deleteTodos** permettra de supprimer des tâches. Ce comportement est réalisé au sein de la méthode **start** de **TodosServer** :

```
void start() {
 HttpServer.bind(HOST, PORT).then((server) {
  server.listen((HttpRequest request) {
   switch (request.method) {
    case "GET":
     handleGet(request);
     break;
    case "POST":
     handlePost(request);
     break;
    case "OPTIONS":
     handleOptions(request);
     break;
    default: defaultHandler(request);
   }
  },
  onError: printError);

  print("Ecoute GET / POST sur
http://$HOST:$PORT");
 },
 onError: printError);
}
```

7 Requêtes HTTP

Déléguée dans les méthodes **handleGet** et **handlePost**, la gestion des requêtes ne pose pas de problème, à cela

près qu'il faut vérifier les chemins des URL requêtées pour les dispatcher correctement et effectuer les actions associées. La récupération des tâches en **GET** sur **/todos** a la forme suivante :

```
void handleGet(HttpRequest req) {
 HttpResponse res = req.response;

 if (req.uri.path == TODOS_PATH) {
  addCorsHeaders(res);
  res.headers.add(HttpHeaders.CONTENT_TYPE, "application/json");
  returnAllTodos(res);
 } else {
  res.statusCode = HttpStatus.METHOD_NOT_ALLOWED;
  res.close();
 }
}
```

Si le chemin requêté est correct, on délègue le renvoi des todos au sein de la méthode **returnAllTodos**. Dans le cas contraire, on renvoie un code HTTP 405 indiquant que la requête n'est pas autorisée. Enfin, on note la méthode **addCorsHeaders** chargée d'ajouter les headers nécessaires pour rendre le serveur accessible à des pages qu'il ne sert pas.

Au niveau des requêtes **POST** d'ajout et de suppression, le code est similaire, avec cependant la nécessité de convertir les informations reçues puisque la méthode **listen** propose un accès à la requête via une liste d'entiers :

```
void handlePost(HttpRequest req) {
 HttpResponse res = req.response;

 if (req.uri.path == ADD_PATH  ||  req.uri.path == DELETE_PATH) {
  addCorsHeaders(res);
  req.listen((List<int> buffer) {
   if (req.uri.path == ADD_PATH) {
   addTodo(buffer, res);
   }else {
   deleteTodos(buffer, res);
   }
  },
   onError: printError);
 } else {
  returnNotAllowed(res);
 }
}
```

8 Gestion des Todos

La modélisation des tâches se fait au sein d'un objet **Todo** possédant les propriétés **id** et **value**. Cet objet devant être partagé entre la partie serveur et la partie client, il est préférable de créer une bibliothèque partagée. Cela se fait en créant un projet et en précisant au sein du script qu'il s'agit d'une bibliothèque, comme suit :

```
library shared_lib;

class Todo {
 int id;
 String value;

 Todo(id, value) {
  this.id = id;
  this.value = value;
 }

 Todo.fromJson(Map data) {
  id = data['id'];
  value = data['value'];
 }

 String toJson() {
  return '{"id" : $id, "value" : "$value"}';
 }
}
```

On remarque le constructeur **Todo.fromJson** pour construire un **Todo** à partir d'une map JSON, ainsi que la méthode **toJson** pour exporter l'objet au format JSON.

Au niveau du serveur, il reste à importer cette bibliothèque et à gérer les interactions avec les todos. La déclaration de la dépendance au sein du fichier **pubspec.yaml** se fait comme suit :

```
 shared_lib:
   path: ..\shared
```

Pour l'ajout d'un nouveau todo, il faut importer le package **dart:json** afin de manipuler les données JSON. Ensuite, l'implémentation est la suivante :

```
import 'package:shared_lib/Todo.dart';
import 'dart:json' as JSON;
// ...
void addTodo(List<int> buffer, HttpResponse res) {
 String strTodo = JSON.parse(new String.fromCharCodes(buffer));
 Map m = JSON.parse(strTodo);
 _todos.insert(m).then((_) => returnAllTodos(res));
}
```

Une fois le todo ajouté, on renvoie au client la liste des todos à jour en réponse à la requête entrante. L'implémentation des méthodes de récupération et de suppression des todos se fait sur le même principe, en ayant recours aux méthodes appropriées du client MongoDB.

9 Projet client

La partie client de l'application doit permettre d'afficher les todos, d'en créer, mais également d'en supprimer. Elle fait interagir une page HTML avec un script Dart spécifique. Au niveau de l'IDE, on crée un nouveau projet de type application web utilisant Web UI. Par défaut, le projet utilise les dépendances des packages **browser**, nécessaire pour avoir le fichier **dart.js**, et **web_ui**. Il reste alors à ajouter la dépendance à la bibliothèque définissant l'objet **Todo**. La hiérarchie du projet montre ainsi la présence du dossier **web** amené à

contenir les fichiers de l'application côté client. En outre, on remarque la présence d'un fichier **build.dart** à la racine du projet. Généré à la création du projet, il est modifiable et définit la configuration du projet web. Ici, il sert simplement à préciser la page web de démarrage de l'application :

```
import 'dart:io';
import 'package:web_ui/component_build.dart';

void main() {
 build(new Options().arguments, ['web/todos.html']);
}
```

Enfin, un dossier **out** est également présent et contiendra les éléments compilés du projet web.

10 Web UI

La bibliothèque Web UI fournit un ensemble de fonctionnalités facilitant la création d'applications web modernes. Reprenant des idées de frameworks tels que AngularJS ou Backbone.js, elle offre le support des Web Components, des templates, du data binding uni- et bidirectionnel, ainsi qu'un modèle événementiel simplifié. L'utilisation de Web UI n'est pas obligatoire, mais ce package offre un réel gain de productivité. Ici, nous créons un composant **TodoComponent** chargé d'afficher un élément **Todo** et de permettre sa sélection en vue d'une suppression. Au niveau Dart, ce composant est minimaliste :

```
import 'package:web_ui/web_ui.dart';
import 'package:shared_lib/Todo.dart';

class TodoComponent extends WebComponent {
 Todo todo;
}
```

Pour gérer l'affichage du composant, un fichier HTML **todo_component.html** est associé :

```
<html>
 <body>
  <element name="x-todo-item" constructor="TodoComponent" extends="li">
   <template>
    <li><input type="checkbox" bind-checked="todo.delete"> {{ todo.value }}</li>
   </template>
   <script type="application/dart" src="todo_component.dart"></script>
  </element>
 </body>
</html>
```

La balise **element** définit un élément **x-todo-item** associé au **TodoComponent**. À l'intérieur de celle-ci, la balise **template** définit le code HTML du Web Component. On remarque ici le binding bidirectionnel avec la propriété **delete** du **Todo** ajoutée pour gérer la suppression d'une tâche. Enfin, les doubles crochets servent à afficher une variable issue d'un script Dart.

11 Page principale

Nommée **todos.html**, la page principale de l'application affiche les todos issus du serveur en s'appuyant sur le **TodoComponent**. Ce dernier permettant de gérer également la sélection des todos à supprimer :

```
<html>
 <head>
  <title>Todos</title>
  <link rel="import" href="todo_component.html">
 </head>
 <body>
  <h1>Todos</h1>
  <input id="id-todo" type="text" placeholder="id">
  <input id="value-todo" type="text" placeholder="value">
  <button on-click="addTodo()">Add Todo</button>
  <ul id="todos">
  <template iterate="todo in todos">
    <x-todo-item todo="{{ todo }}"></x-todo-item>
   </template>
  </ul>
  <button on-click="deleteTodos()">Delete Selected</button>
  <script type="application/dart" src="todos.dart"></script>
  <script type="text/javascript" src="packages/browser/dart.js"></script>
 </body>
</html>
```

L'injection du composant **todo** se fait dans la page HTML via la balise **link** et son attribut **rel** positionné à **import**. La gestion des évènements est réalisée en ajoutant l'attribut **on-{event}** sur l'élément dont on souhaite écouter un évènement. Pour l'ajout du **todo**, on ajoute ainsi **on-click** sur la balise **button** en déclarant le nom de la fonction appelée du côté du script **todos.dart**.

D'autre part, l'utilisation du composant passe par la balise **template** permettant d'itérer sur les tâches de la variable **todos**. Pour chaque tâche, on appelle notre composant via **x-todo-item** en passant en entrée l'instance de **todo**. Enfin, on note l'inclusion du script Dart de type **application/dart**. L'ajout du script **dart.js** servant quant à lui à faire le switch entre le code Dart et le code JavaScript à l'exécution, dans le cas où le navigateur n'embarque pas de machine virtuelle dédiée.

12 Chargement des todos

Côté client, la gestion des échanges avec le serveur se fait au sein du script **todos.dart**. Le chargement des todos conduit à la mise en œuvre de la classe **HttpRequest** du package **dart:html**, ainsi que des méthodes du package **dart:json** pour parser les données au format JSON échangées :

```
void loadTodos() {
 HttpRequest.getString("http://$host/todos").then((txt) {
  todos.clear();
  List tmp = JSON.parse(txt);

  for (var item in tmp) {
```

```
   Map m = JSON.parse(item);
   Todo todo = new Todo.fromJson(m);
   todos.add(todo);
  }

  dispatch();
 });
}
```

De scope global, l'objet todos est utilisé dans la page HTML pour itérer sur les tâches. L'appel à la méthode **dispatch** sert à mettre à jour l'IHM une fois les tâches récupérées depuis le serveur. En effet, le binding des données entre IHM et code est automatique dans le cadre d'évènements générés depuis l'IHM. Ici, la méthode **loadTodos** est appelée au sein du main ce qui implique le recours à dispatch.

13 Création des todos

L'ajout d'un todo impose de récupérer les informations saisies dans la page HTML. En lieu et place des sélecteurs jQuery, Dart propose la méthode **query** dans le package **dart:html**, réalisant le même travail. L'API ne s'arrête pas là, en offrant d'autres fonctionnalités puissantes qu'il est bon de connaître. Cela nous donne donc :

```
void addTodo() {
 var id = query("#id-todo");
 var input = query("#value-todo");
 var request = new HttpRequest();
  request.onReadyStateChange.listen((txt) {
   if (request.readyState == HttpRequest.DONE &&
    (request.status == 200 || request.status == 0)) {
     displayTodos(txt.currentTarget.responseText);
   }
  });

 request.open("POST", "http://$host/addTodo", async : false);
 request.send(JSON.stringify(todo));
 id.value = "";
 input.value = "";
}
```

L'envoi de la requête **POST** au serveur se fait en ouvrant la connexion et en envoyant le contenu du todo. On écoute le retour de la requête pour récupérer la liste des todos à jour depuis le serveur. La requête de suppression se fait de la même manière que l'ajout, en envoyant une requête au serveur sur **/deleteTodos** avec en entrée la liste des todos à supprimer au format JSON.

Fig. 3 : Todo côté client

Fig. 4 : Todo côté serveur

14 Exécution

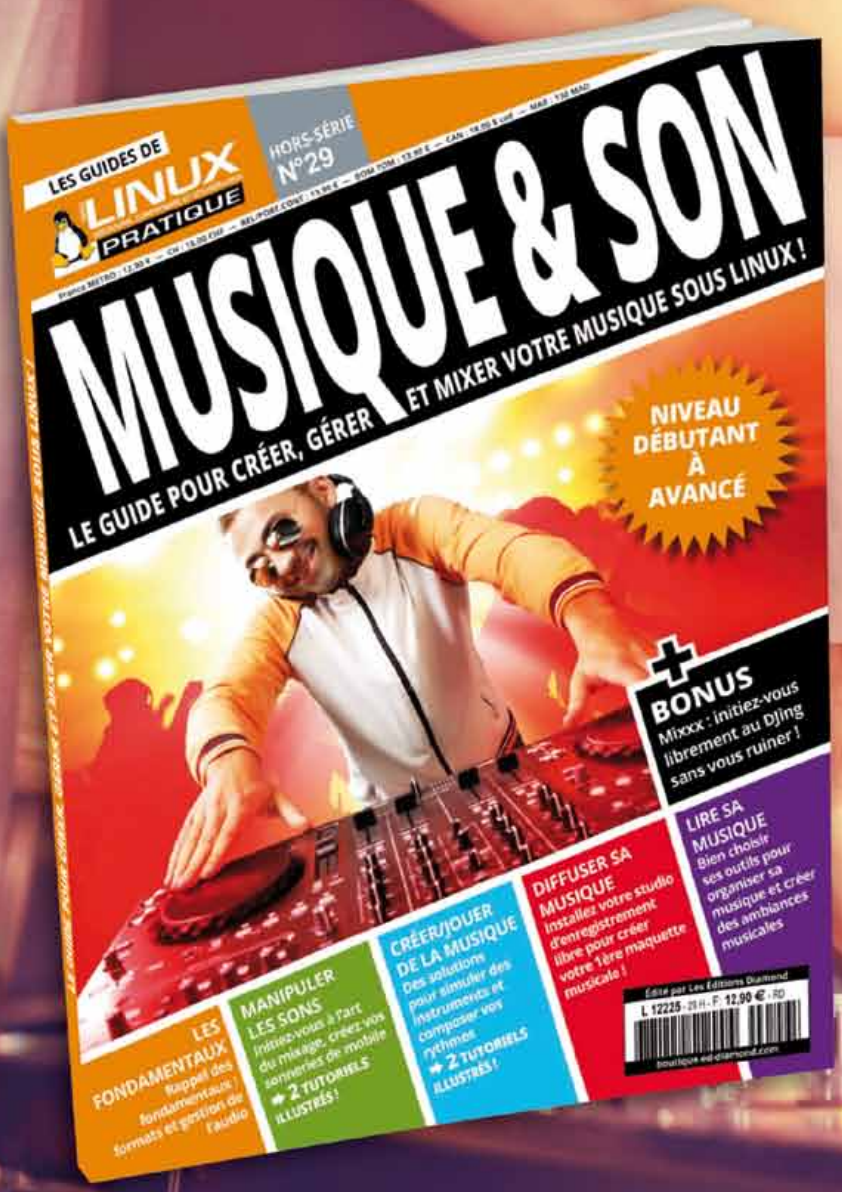
Le serveur et le client en place, nous procédons aux tests de l'application web. Pour ce faire, nous démarrons le serveur, via le script **todosserver.dart**, qui se met en attente de requêtes **GET** et **POST** sur http://127.0.0.1:8080/. Nous testons ensuite la partie cliente en allant sur le fichier **todos.html** dans l'IDE, puis en faisant un clic droit et en choisissant ***Run in Dartium***, ce qui lance la page dans le navigateur Dartium. Quelques ajouts et suppressions montrent que l'application web est opérationnelle et répond bien à son cahier des charges côté client (figure 3) et côté serveur (figure 4).

Notre application web fonctionne parfaitement au sein d'une machine virtuelle Dart. Pour la déployer en production, il faut utiliser le code embarquant l'équivalent JavaScript compilé via **dart2js**. Ce code est généré automatiquement par l'IDE au sein du dossier **out** dont le contenu constitue le code client de l'application utilisable via n'importe quel navigateur.

Conclusion

Encore récente, la plateforme Dart s'annonce comme une solution prometteuse pour le futur. L'application réalisée au cours de cet article aura mis en lumière la puissance et la simplicité de la plateforme. Pouvoir réaliser une application web de bout en bout avec un unique langage est un luxe qui séduira de nombreux développeurs. Le seul point d'interrogation concernant l'avenir de la plateforme provient du peu de chances de voir la machine virtuelle intégrée par d'autres navigateurs que Chrome. Néanmoins, avec plus de 30% de parts de marché, ce dernier constitue une base plus que suffisante pour s'y intéresser, d'autant plus que le compilateur **dart2js** permet d'obtenir un code source compatible avec l'ensemble des navigateurs récents, avec des performances meilleures qu'un code directement écrit en JavaScript. ■

MUSIQUE & SON



LE GUIDE POUR CRÉER, GÉRER ET MIXER VOTRE MUSIQUE SOUS LINUX !

LINUX PRATIQUE HORS-SÉRIE N°29

ACTUELLEMENT DISPONIBLE !

CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

boutique.ed-diamond.com

À LA DÉCOUVERTE DU SYSTÈME DE BUILD REDO

par Thierry Gayet

Bien que GNU/Make soit devenu un standard, redo a été développé comme une alternative devant combler les lacunes de l'outil initial. Nous allons voir comment l'utiliser.

Introduction

Développé en scripts Bash et Python, **redo** est un concurrent de plus à GNU/Make basé sur une conception de Daniel J. Bernstein (aka djb). Une réécriture en langage C est en cours pour le rendre encore plus performant. L'idée de **redo** a été de réécrire le fonctionnement de **make** en seulement 250 lignes de script shell. Une version *light* (partie minimale de **redo**) a même été réalisée en vérifiant les dépendances et ce, en seulement 150 lignes de code, soit environ 3 kilo-octets.

Le nom « redo » vient du fait que dans la communauté, 75 versions différentes de **make** coexistent sous la même appellation et aussi pour faire la même chose. Donc, au lieu de reprendre le même nom et de s'ajouter à la longue liste, le nom « redo » s'est imposé, ce qui le rend plus visible face aux autres.

Cela en fait donc un mini système de build, avec gestion de dépendances qui peut être inclus dans les projets eux-mêmes. Il est distribué sous licence LGPLv2 ; la version minimale étant quant à elle donnée au domaine public.

1 Obtenir redo

Pour pouvoir commencer avec l'utilisation de **redo**, il faut récupérer le script principal que l'on installera dans **/opt** :

```
$ cd /opt
$ su
$ git clone https://github.com/apenwarr/redo.git
Cloning into 'redo'...
remote: Reusing existing pack: 2001, done.
remote: Total 2001 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (2001/2001), 529.67 KiB | 303 KiB/s, done.
Resolving deltas: 100% (838/838), done.
$ chown -R tgayet : redo/
$ cd redo
```

On peut y observer un mélange de script Bash et Python. Aujourd'hui, l'implémentation de **redo** est redirigée via un lien symbolique vers le script **redo.py** :

```
$ ls -l `find . -maxdepth 1 -type l -print`
lrwxrwxrwx 1 denis denis  7 févr.  5 15:43 ./redo -> redo.py
lrwxrwxrwx 1 denis denis 14 févr.  5 15:43 ./redo-always -> redo-always.py
lrwxrwxrwx 1 denis denis 16 févr.  5 15:43 ./redo-ifchange -> redo-ifchange.py
lrwxrwxrwx 1 denis denis 16 févr.  5 15:43 ./redo-ifcreate -> redo-ifcreate.py
lrwxrwxrwx 1 denis denis 11 févr.  5 15:43 ./redo-ood -> redo-ood.py
lrwxrwxrwx 1 denis denis 15 févr.  5 15:43 ./redo-sources -> redo-sources.py
lrwxrwxrwx 1 denis denis 13 févr.  5 15:43 ./redo-stamp -> redo-stamp.py
lrwxrwxrwx 1 denis denis 15 févr.  5 15:43 ./redo-targets -> redo-targets.py
lrwxrwxrwx 1 denis denis 16 févr.  5 15:43 ./redo-unlocked -> redo-unlocked.py
```

Il en va de même pour les autres scripts. Tous ces scripts doivent être mis dans le chemin courant, en ajoutant la ligne suivante dans votre **~/.bashrc** :

```
export PATH="${PATH}:/opt/redo"
```

2 Premiers pas avec redo

La théorie derrière **redo** est presque magique : grosso modo, il peut faire tout ce que **make** peut faire. Seule la mise en œuvre est beaucoup plus simple, car la syntaxe est plus propre et vous pouvez rendre les choses plus souples sans recourir à des hacks immondes.

Loin des automatismes des templates offerts par Autotools, CMake ou autre, **redo** est vraiment orienté comme un système minimal mais néanmoins fonctionnel. Pour l'exemple, nous allons utiliser les codes sources suivants. Premièrement, nous allons avoir besoin d'une fonction d'entrée **main()** qui sera définie dans le fichier **main.c** :

```
#include <stdio.h>

#include "extension.h"

int main()
{
  printf(module);

  return 0;
}
```

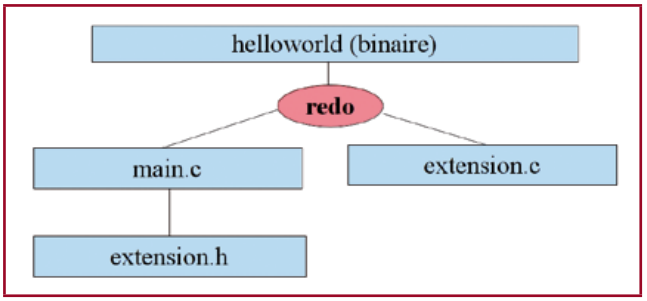
Nous aurons également besoin d'un *header* **extension.h**, qui définira une déclaration externe au **main** :

```
extern char *module;
```

Enfin, pour montrer un exemple concret de programmation modulaire, nous allons définir un second fichier source **extension.c** :

```
char *module = "hello, world!\n";
```

Les trois fichiers seront utilisés lors de la compilation :



Commençons par regarder la version de **redo** dont nous disposons :

```
$ redo --version
0.11
```

Tout d'abord, créons un fichier **default.o.do**, qui définira le comportement de la cible par défaut pour l'étape de compilation :

```
redo-ifchange $2.c
gcc -MD -MF $2.d -c -o $3 $2.c
read DEPS <$2.d
redo-ifchange ${DEPS#*:}
```

Ensuite, définissons le fichier **helloworld.do**, qui définira comment créer la cible principale, en l'occurrence notre binaire **helloworld**. Il décrit, en plus des dépendances, l'étape de l'édition de liens :

```
DEPS="main.o extension.o"
redo-ifchange $DEPS
gcc -o $3 $DEPS
```

Le nom du fichier **helloworld** servira comme nom du binaire à générer ; il faut donc ajuster le nom du fichier et donc de la cible en fonction de ce qui est à construire.

Nous pouvons voir les dépendances du binaire via les deux fichiers **main.o** et **extension.o**. La règle par défaut est apportée dans le fichier **default.o.do**.

À ce stade, nous devons avoir les fichiers suivants :

```
$ ls
default.o.do    extension.c    extension.h
helloworld.do   main.c
```

L'exécution donnera le résultat suivant :

```
$ redo helloworld
redo  helloworld
redo  main.o
main.c: In function 'main':
main.c:7:2: attention : le format n'est pas
   une chaîne littérale et pas d'argument
   de format [-Wformat-security]
redo  extension.o
```

Une fois le processus lancé, des fichiers de dépendance d'extension **.d** apparaissent :

```
$ ls
clean.do     default.o.do    extension.c    extension.d
extension.h  extension.o     helloworld     helloworld.do
main.c       main.d          main.o
```

Nous pouvons consulter ces deux fichiers de dépendance, qui sont utilisés par **redo** lors de la gestion de dépendances :

```
$ cat extension.d
extension.o.redo2.tmp: extension.c
$ cat main.d
main.o.redo2.tmp: main.c /usr/include/stdio.h /usr/include/features.h \
 /usr/include/x86_64-linux-gnu/bits/predefs.h \
 /usr/include/x86_64-linux-gnu/sys/cdefs.h \
 /usr/include/x86_64-linux-gnu/bits/wordsize.h \
 /usr/include/x86_64-linux-gnu/gnu/stubs.h \
 /usr/include/x86_64-linux-gnu/gnu/stubs-64.h \
 /usr/lib/gcc/x86_64-linux-gnu/4.6/include/stddef.h \
 /usr/include/x86_64-linux-gnu/bits/types.h \
 /usr/include/x86_64-linux-gnu/bits/typesizes.h /usr/include/libio.h \
 /usr/include/_G_config.h /usr/include/wchar.h \
 /usr/lib/gcc/x86_64-linux-gnu/4.6/include/stdarg.h \
 /usr/include/x86_64-linux-gnu/bits/stdio_lim.h \
 /usr/include/x86_64-linux-gnu/bits/sys_errlist.h extension.h
```

La modification du *timestamp* sur le header **extension.h** pour simuler sa modification régénérera ce qui a bougé :

```
$ touch extension.h
$ redo helloworld
redo  helloworld
redo  main.o
main.c: In function 'main':
main.c:7:2: attention : le format n'est pas une chaîne littérale et
pas d'argument de format [-Wformat-security]
```

En modifiant le timestamp du fichier source **extension.c**, il en va de même, mais sur le fichier concerné :

```
$ touch extension.c
$ redo helloworld
redo  helloworld
redo  extension.o
```

Cela nous aura bien généré le binaire **helloworld** comme attendu :

```
$ ls -al helloworld
-rwxrwxr-x 1 tgayet tgayet 8453 févr.  1 12:45 helloworld
$ file helloworld
helloworld: ELF 64-bit LSB executable, x86-64,
  version 1 (SYSV), dynamically linked
  (uses shared libs), for GNU/Linux 2.6.24,
  BuildID[sha1]=0xade40be9e76eddf06173a5286339890faa0a821e,
  not stripped
```

Cependant, la cible **helloworld** est appelée explicitement, mais n'est pas associée à une cible par défaut. L'appel direct à **redo** sans paramètre nous affichera donc le message suivant :

```
$ redo
redo  no rule to make 'all'
```

Nous pouvons remédier à cela en créant un fichier **all.do** qui servira de cible par défaut :

```
redo-ifchange helloworld
redo helloworld
```

Ce qui permettra désormais d'appeler **redo** directement, qui à son tour redirigera vers la cible **helloworld** :

```
$ redo
redo  all
redo  helloworld
redo  main.o
main.c: In function 'main':
main.c:7:2: attention : le format n'est pas une
  chaîne littérale et pas d'argument de
  format [-Wformat-security]
redo  extension.o
redo  helloworld
```

Pour plus de commodité, nous aurons aussi besoin de définir un fichier **clean.do** de façon à permettre le nettoyage de l'environnement. Ce fichier contiendra :

```
rm -fR *.o
rm -fR *.d
rm -fR helloworld
```

Son exécution sera simple :

```
$ redo clean
redo  clean
```

Sans la définition de la cible par défaut **default.o.do** que nous venons de créer, nous aurions eu le message d'erreur suivant :

```
$ redo helloworld.do
redo  helloworld.do: exists and not marked as generated; not redoing.
```

3 Les variables $1, $2 et $3

Redo utilise dans les fichiers **.do** des variables **$1**, **$2** ou **$3** qui nécessitent une clarification :

Nom de la variable	Description
$1	nom du fichier cible
$2	nom de base de la cible, sans son extension
$3	nom d'un fichier temporaire qui sera renommé le fichier cible (seulement si le fichier **.do** ne renvoie pas d'erreurs)

Une remarque vient à l'usage de **redo** : pourquoi ne pas avoir nommé les variables telles que **$FILE**, **$EXT**, et **$OUT** au lieu de **$1**, **$2** ou **$3** ?

Cela semble tentant et facile, mais malheureusement cela est un inconvénient via la compatibilité descendante en lien avec la conception originale de **redo** effectuée par l'auteur initial djb. En effet, les noms plus longs ne sont pas nécessairement mieux. Apprendre la signification des trois variables ne prend pas longtemps et n'oubliez pas que **make** ou **perl** ont eu aussi d'étranges noms de variables à un seul caractère pendant une longue période (exemple : les **makefile** **$@** ou **$^** dans les **makefiles** GNU).

4 Au sujet des flags de compilation et de link

Le fichier **default.o.do** est à la base des règles de **redo**. Il peut disposer de plusieurs variantes, comme par exemple, en incluant la définition des flags utilisés lors de la compilation du code C ou C++.

```
redo-ifchange $2.c

$CC $CPPFLAGS $CFLAGS -MD -MF $2.d -c -o $3 $2.c

read DEPS <$2.d
redo-ifchange ${DEPS#*:}
```

Attention : pour l'exemple précédent, **gcc** a été remplacé par une variable d'environnement **$CC**, ce qui permettra de fonctionner à la fois en compilation native, mais aussi en cross-compilation. Donc, pour que cela fonctionne en compilation native, il faut au préalable exporter la variable **CC** dans son environnement :

```
$ export CC=gcc
```

Quant aux variables **CFLAGS** ou **CPPFLAGS**, en Bash, une variable non définie est vue comme une chaîne vide, ce qui ne nous posera pas de problème à l'exécution.

Pour l'étape de l'édition de liens, on pourra également modifier le fichier de la cible **helloworld.do** du fichier qui définit l'édition de liens par :

```
$CC $LDFLAGS -o $3 $objs $libs $LDLIBS
```

Là aussi, la définition de **CC** dans l'export nous sera utile pour l'édition de liens native, car sinon, nous n'invoquerons pas le bon **LD**.

5 Exemples d'autres cibles

Pour chaque cible, **all**, **clean**, **documentation**, **install**, **test**, etc., un fichier avec l'extension **.do** peut être ajouté au projet. En voici quelques exemples pour illustration.

5.1 documentation.do, pour la génération de documentation

```
deps=$(find -name '*.[ch]')
redo-ifchange $deps

if ! (which doxygen &>/dev/null); then
    echo 1>&2 arrêt de la génération de la documentation ; merci d'installer doxygen
    exit 0
fi
doxygen >doxygen.out
```

Sera exécuté via la commande **redo documentation** ou **make documentation** via le wrapper **make**.

5.2 ctag.do, pour l'indexation du code

```
deps=$(find -name '*.[ch]')
redo-ifchange $deps

if ! (which ctags &>/dev/null); then
    echo 1>&2 arret de l'indexation ; merci
d'installer ctags
    exit 0
fi
ctags -f- --exclude="output" --exclude="html" -R
```

Sera exécuté via la commande **redo ctag** ou **make ctags** via le wrapper **make**.

5.3 install.do, pour l'installation des fichiers

```
#!/bin/bash
exec >&2
redo-ifchange all

INSTALL_DIR=${INSTALL_DIR:-output/generic}
mkdir -p $INSTALL_DIR

files_to_copy="
   helloworld
"
cp -au $files_to_copy "$INSTALL_DIR"
```

Sera exécuté via la commande **redo install** ou **make install**.

À noter l'usage de **exec >&2**, qui permet de sécuriser l'usage de certains scripts en lien avec **redo**.

6 Makefile pour contrôler redo

Comme nous l'avons vu, tout se fait en invoquant le script **redo**. **redo all** lorsqu'il est sans argument, ou bien **redo clean** pour nettoyer l'environnement de développement, ou enfin **redo helloworld** pour spécifier la génération d'une cible.

Pour interconnecter les GNU/Make avec **redo**, un petit **makefile** est fourni. Il sert juste de passerelle entre les deux systèmes :

```
all:

Makefile:
  @

%: FORCE
  +./redo $@

.PHONY: FORCE
```

Cela permettra d'invoquer les mêmes cibles passées à **make** pour **redo** :

```
$ make clean
redo clean
redo clean
$ make
redo all
redo all
redo helloworld
redo main.o
main.c: In function 'main':
main.c:7:2: attention : le format n'est pas
une chaîne littérale et pas d'argument de
format [-Wformat-security]
redo extension.o
redo helloworld
```

7 Paramètres utilisables avec redo

L'outil **redo** possède un certain nombre de paramètres que l'on peut utiliser pour détailler les étapes internes. En général, ils ne sont pas utiles, sauf en cas de problème dans la chaîne de build (voir tableau ci-dessous).

À partir de notre exemple précédent qui donnait le résultat suivant, nous allons voir que nous pouvons avoir plus d'informations suivant les paramètres :

```
$ redo clean && redo all
redo  clean
redo  all
redo  helloworld
redo  main.o
redo  extension.o
```

En mode **verbose**, nous obtenons la sortie suivante avec des informations détaillées sur les étapes intermédiaires à **redo** :

```
$ redo clean --verbose ; redo all --verbose

redo  clean
* sh -ev clean.do clean clean clean.redo2.tmp
rm -fR *.o
rm -fR *.d
rm -fR helloworld

find -depth '(' -name '.do_built*' -o -name '*.did' ')'
-exec rm -rf "{}" \;
redo  clean (done)


redo  all
* sh -ev all.do all all all.redo2.tmp
redo-ifchange helloworld
redo    helloworld
* sh -ev helloworld.do helloworld helloworld helloworld.
redo2.tmp
DEPS="main.o extension.o"
redo-ifchange $DEPS

redo      main.o
* sh -ev default.o.do main.o main main.o.redo2.tmp
redo-ifchange $2.c

$CC $CPPFLAGS $CFLAGS -MD -MF $2.d -c -o $3 $2.c

read DEPS <$2.d
redo-ifchange ${DEPS#*:}
redo      main.o (done)

redo      extension.o
* sh -ev default.o.do extension.o extension
extension.o.redo2.tmp
redo-ifchange $2.c

$CC $CPPFLAGS $CFLAGS -MD -MF $2.d -c -o $3 $2.c

read DEPS <$2.d
```

Nom du paramètre	Description
-j n **--jobs=n**	Nombre de threads à lancer pour paralléliser la génération
-d **--debug**	Affiche des infos de debug au niveau des vérifications de dépendance
-v **--verbose**	Affiche toutes les informations lues dans les fichiers de dépendance **.d** et les fichiers **.do**
-x **--xtrace**	Affiche les commandes telles qu'elles sont exécutées réellement
-k **--keep-going**	Continue aussi longtemps que possible, même si une cible échoue
--shuffle	Rend l'ordre du build aléatoire pour trouver des bugs de dépendance
--debug-locks	Affiche les messages sur les fichiers de verrouillage (utile pour le debug de **redo**)
--debug-pids	Affiche le *process id* (aka PID) dans les traces (utile pour le debug)
--version	Affiche la version courante du script **redo** et sort

```
redo-ifchange ${DEPS#*:}
redo      extension.o (done)

LDLIBS="$LDLIBS -lpthread"
$CC -o $3 $DEPS $objs $LOADLIBES $LDLIBS
redo    helloworld (done)

redo helloworld

redo    helloworld
* sh -ev helloworld.do helloworld helloworld helloworld.
redo2.tmp
DEPS="main.o extension.o"
redo-ifchange $DEPS
LDLIBS="$LDLIBS -lpthread"
$CC -o $3 $DEPS $objs $LOADLIBES $LDLIBS
redo    helloworld (done)

redo  all (done)
```

Le paramètre **debug** montre davantage d'informations sur les dépendances :

```
$ redo -debug clean && redo --debug all
redo  clean
redo  clean (done)

redo  all
redo:   ?helloworld
redo:   -- DIRTY (missing)
redo    helloworld
redo:     ?main.o
redo:     -- DIRTY (missing)
redo      main.o
redo:       ?main.c
redo:       ?main.c
redo:       -- CLEAN (checked)
redo:       ?../../../../../../usr/include/stdio.h
redo:       ?../../../../../../usr/include/features.h
redo:       ?../../../../../../usr/include/bits/predefs.h
redo:       -- DIRTY (never built)
redo:       ?../../../../../../usr/include/sys/cdefs.h
redo:       -- DIRTY (never built)
redo:       ?../../../../../../usr/include/bits/wordsize.h
redo:       -- DIRTY (never built)
redo:       ?../../../../../../usr/include/gnu/stubs.h
redo:       -- DIRTY (never built)
[...]
redo:       ?../../../../../../usr/include/bits/stdio.h
redo:     ?extension.o
redo:       ?default.o.do
redo:       -- CLEAN (checked)
redo:       ?extension.c
redo:       -- CLEAN (checked)
redo    helloworld (done)

redo  all (done)
```

Pour la sortie avec le paramètre **xtrace**, nous pouvons voir les commandes telles que **redo** les exécute :

```
$ redo clean --xtrace && redo all --xtrace

redo  clean
* sh -ex clean.do clean clean clean.redo2.tmp
+ rm -fR '*.o'
+ rm -fR '*.d'
+ rm -fR helloworld
+ find -depth '(' -name '.do_built*' -o -name
  '*.did' ')' -exec rm -rf '{}' ';'
redo  clean (done)
```

```
redo  all
* sh -ex all.do all all all.redo2.tmp
+ redo-ifchange helloworld
[...]
redo    helloworld (done)

redo  all (done)
```

8 Utilisation de redo en cross-compilation

Pour la cross-compilation, le plus simple est de définir un script de configuration **setenv-arm-rpi.sh**, qui devra être sourcé avant l'étape de génération. Exemple avec la *toolchain* ARM pour la Raspberry Pi :

```
#
# Définition pour le support ARM pour la Raspberry Pi
#

export ARM_TOOLCHAIN_PATH="/home/tgayet/x-tools/arm-
rpi-linux-gnueabi/bin"
export PREFIX="arm-rpi-linux-gnueabi"

# ------------------------------------------------

# Définit les outils de la chaîne de cross-compilation
export CC="${ARM_TOOLCHAIN_PATH}/${PREFIX}-gcc"
export CPP="${ARM_TOOLCHAIN_PATH}/${PREFIX}-cpp"
export AR="${ARM_TOOLCHAIN_PATH}/${PREFIX}-ar"

# Met à jour les flags de compilation
export CFLAGS="${CFLAGS} --sysroot=${SYSROOT}
-I${SYSROOT}/usr/include -I${ANDROID_PREFIX}/include
-I${DEV_PREFIX}/android/bionic"
export CPPFLAGS="${CFLAGS}"
export LDFLAGS="${LDFLAGS} -L${SYSROOT}/usr/lib
-L${ANDROID_PREFIX}/lib"
if test ${DEBUG}; then
    # Pour le mode debug, inclut les symboles de debug
    export CFLAGS="$CFLAGS -g -Wall -Wfatal-errors
-Werror -Wno-unused"
else
    # Pour le mode release, enlève les symboles de
debug et optimise
    export CFLAGS="$CFLAGS -O2"
    export CPPFLAGS="$CPPFLAGS -DNDEBUG"
fi

# Met à jour le prompt pour renseigner de
l'architecture courante
label="[CC ARM]"
export ORIG_PS1=${ORIG_PS1:-$PS1}
export PS1="%F{yellow}$label%f$ORIG_PS1"
```

Le résultat est concluant et a bien été généré pour l'architecture ARM :

```
$ source setenv-arm-rpi.sh
$ redo clean ; redo all
redo  clean
redo  all
redo    helloworld
```

```
redo      main.o
redo      extension.o
redo    helloworld
```

Le code généré est bien de l'ARM, comme demandé :

```
$ file *|grep ARM
extension.o:          ELF 32-bit LSB relocatable,
    ARM, version 1 (SYSV), not stripped
helloworld:           ELF 32-bit LSB executable,
    ARM, version 1 (SYSV), dynamically linked
    (uses shared libs), for GNU/Linux 3.0.39,
    not stripped
main.o:               ELF 32-bit LSB relocatable,
    ARM, version 1 (SYSV), not stripped
```

9 Scripts de pilotage

Tout comme pour l'exemple des **makefiles**, il est possible d'utiliser des scripts Bash qui amélioreront l'usage de **redo**. Cela peut être intéressant d'ajouter un paramètre **--with-arm-arch** ou **--with-mipss-arch** de façon à sourcer une toolchain en remplacement de celle native avant de lancer un build.

Conclusion

Vous voilà éclairé sur l'usage de **redo**, que vous pourrez croiser dans certains packages open source, ou utiliser au sein de vos développements. Quoi qu'il en soit, vous avez désormais une corde de plus à votre arc. ■

Liens

- https://github.com/apenwarr/redo
- http://grosskurth.ca/papers/mmath-thesis.pdf
- http://cr.yp.to/redo.html
- http://cr.yp.to/redo/atomic.html
- http://cr.yp.to/redo/honest-script.html
- http://cr.yp.to/redo/honest-nonfile.html
- http://cr.yp.to/redo/honest-crossdir.html
- http://miller.emu.id.au/pmiller/books/rmch/
- http://aegis.sourceforge.net/auug97.pdf
- http://www.xs4all.nl/~evbergen/nonrecursive-make.html
- http://cr.yp.to/data.html

Quelle interopérabilité entre mes différents fournisseurs Cloud ?

Avec Aruba Cloud,

vous avez l'assurance de ne pas être prisonnier d'un fournisseur. Nos services sont intégrés au **driver DeltaCloud** et compatibles **S3**. De plus, vous pouvez utiliser des formats standards d'images de machines virtuelles, **avec VHS et VMDK**, ainsi que des modèles personnalisés provenant éventuellement d'autres sources.

3 hyperviseurs

6 datacenters en Europe

APIs et connecteurs

70+ templates

Contrôle des coûts

Nous avons choisi Aruba Cloud car nous bénéficions d'un haut niveau de performance, à des coûts contrôlés et surtout car ils sont à dimension humaine, comme nous. Xavier Dufour - Directeur R&D - ITMP

Contactez-nous! 0810 710 300 www.arubacloud.fr

aruba CLOUD

Cloud Public | Cloud Privé | Cloud Hybride | Cloud Storage | Infogérance

MY COUNTRY. MY CLOUD.*

EXIGEZ LA VRAIE SOLUTION LIBRE ET GRATUITE DE MESSAGERIE COLLABORATIVE,

OBM.ORG !

ET MÉFIEZ-VOUS DES IMITATIONS !

OBM.org est toujours la solution de messagerie des grandes administrations françaises : Ministère de l'Économie et des Finances, Ministère de l'Intérieur, Gendarmerie, Ministère de la Culture et de la Communication et Ministère de l'Agriculture, de l'Agroalimentaire et de la Forêt.

C'est la solution utilisée par la moitié des administrations françaises !

LINAGORA

www.linagora.com www.obm.org