N°177 décembre 2014 DÉVELOPPEMENT ET ADMINISTRATION SUR SYSTÈMES OPEN SOURCE

GNU LINUX MAGAZINE / FRANCE

CODE / TESTS
Adoptez le Behaviour Driven Development pour des tests lisibles
p.72

ACTU / PHP
Prenez connaissance des nouveautés de PHP 5.6 p.14

PYTHON / TURING
Découvrez le Brainfuck et écrivez un interpréteur Python pour ce langage p.52

ALGO / CRYPTO
DES, AES, RSA, chiffrement asymétrique, signature, preuve...

COMPRENEZ LA CRYPTOGRAPHIE !

La seule arme efficace pour protéger votre vie privée! p.30

ANDROID / GO
Exécutez du code Go sous Android p.60

REPÈRES / HISTOIRE
Suivez la transition des machines mécaniques aux machines électroniques p.20

SYSADMIN / SUPERVISION
Effectuez un monitoring réactif avec Riemann p.38

L 19275 - 177 - F: 7,90 € - RD

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

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 VHD 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

MY COUNTRY. MY CLOUD.*

Cloud Public | Cloud Privé | Cloud Hybride | Cloud Storage | Infogérance

GNU/Linux Magazine France
est édité par Les Éditions Diamond

LES ÉDITIONS DIAMOND

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

Directeur de publication : Arnaud Metzler
Chef des rédactions : Denis Bodor
Rédacteur en chef : Tristan Colombo
Secrétaire de rédaction : Fleur Brosseau
Réalisation graphique : Kathrin Scali & Carine Greppat
Responsable publicité : Valérie Fréchard,
Tél. : 03 67 10 00 27
v.frechard@ed-diamond.com
Service abonnement : Tél. : 03 67 10 00 20
Impression : pva, Druck und Medien-Dienstleistungen GmbH, Landau, Allemagne

Distribution France : (uniquement pour les dépositaires de presse)
MLP Réassort : Plate-forme de Saint-Barthélemy-d'Anjou.
Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier. Tél. : 04 74 82 63 04

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

Périodicité : Mensuel
Prix de vente : 7,90 €

MIXTE
Papier issu de sources responsables
FSC® C015136

Membre April
www.april.org

SUIVEZ-NOUS SUR :

https://www.facebook.com/editionsdiamond

@gnulinuxmag

LES ABONNEMENTS ET LES ANCIENS NUMÉROS SONT DISPONIBLES !

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

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

ÉDITORIAL

Edward Snowden. Ce nom vous dit quelque chose ? Forcément ! Depuis ses révélations sur les programmes d'écoute massive des gouvernements américains et britanniques, tout le monde connaît Edward Snowden. Et d'ailleurs, depuis le 6 juin 2013, date à laquelle les premières révélations sont parues dans la presse, les gens s'interrogent sur la manière de protéger leurs données... et en général ils en restent là. En effet il y a toujours essentiellement deux groupes : les « paranos » qui cryptaient déjà toutes leurs données bien avant les révélations de Snowden et les « naïfs », ceux qui vous diront « Moi, je n'ai rien à me reprocher », qui ne cryptaient rien et qui suivent toujours le même raisonnement. Il est bien plus simple de faire partie du second groupe. En effet, il est très pratique d'utiliser Gmail, Google Drive, Dropbox et autres sans se soucier de ce que deviennent les données. Pourtant ce n'est sans doute pas un hasard si tous ces services « gratuits » ne proposent pas une solution native simple pour chiffrer vos messages et fichiers avant de les envoyer sur les serveurs (et bien sûr pour pouvoir les déchiffrer tout aussi simplement). Ce n'est pas forcément la NSA qui exploitera vos données, mais des entreprises commerciales. Certes, ce n'est pas (encore) le cas, mais elles en ont le pouvoir. Réfléchissez aux profils que Google pourrait vendre en se basant sur les données accumulées par les recherches Web, les mails et la localisation des smartphones sous Android. Il est normal qu'une telle base de connaissances attire les agences gouvernementales et plus largement, même si Google s'en défend, les entreprises commerciales. Ces données peuvent être exploitées soudainement suite à différents événements : un changement de politique de Google, un piratage, un coup d'État dans un des pays hébergeant les serveurs, etc.

La prise de conscience du danger est la première étape et en cela Edward Snowden a servi d' « accélérateur ». L'étape suivante, logique, est de mettre en place une solution de chiffrement (comme GnuPG présentée dans le numéro de février de *GNU/Linux Magazine* ou plus rapidement dans le numéro de ce mois-ci de *Linux Pratique*). On pourrait s'arrêter là, mais vous comme moi nous sommes curieux, nous avons besoin de comprendre. La dernière étape est donc d'analyser la manière dont les données sont cryptées. Dans ce numéro, Léo Ducas-Binda vous explique justement les protocoles de base de la cryptographie avant de revenir le mois prochain avec un article sur le futur de la cryptographie et les réseaux euclidiens.

Et que cela ne vous fasse pas oublier les autres rubriques, avec des sujets très variés ce mois-ci : de nombreuses occasions d'apprendre, encore et toujours. Bonne lecture !

**Les codes sources de certains articles sont désormais disponibles sur GitHub : https://github.com/glmf.
Je pense que l'on apprend plus de choses en tapant le code, en commettant des erreurs que l'on cherche ensuite à corriger plutôt qu'en récupérant un code pour le bidouiller. Par contre, certains des codes présentés dans le magazine peuvent devenir de véritables projets et, en ce sens, il peut être intéressant de continuer à les faire vivre après la parution du magazine, que vous puissiez, vous lecteurs, vous les approprier et que nous partagions éventuellement ensemble vos avancées. Alors copiez, forkez, améliorez... et vive le logiciel libre !**

Tristan Colombo

AGENDA

PHPBenelux Conference 2015

Date : les 23 et 24 janvier 2015

Lieu : Anvers - Belgique (en anglais)

Site officiel : http://conference.phpbenelux.eu/2015

6ème conférence de l'association PHPBenelux. À l'heure où ces lignes sont écrites, le programme n'est pas encore connu, mais parmi les thèmes proposés lors de l'appel à contribution, on pouvait trouver :

- le futur de PHP ;
- la sécurité ;
- les tests (unitaires, fonctionnels, etc.) ;
- l'expérience utilisateur ;
- etc.

Chaque présentation durera une heure dont dix minutes de questions. Les trois dernières heures seront consacrées à un workshop.

PyCon 2015

Date : du 8 au 16 avril 2015

Lieu : Montréal - Canada (en anglais)

Site officiel : https://us.pycon.org/2015/

PyCon aura lieu en 2015 pour la première fois chez nos amis québécois... et ça se voit dès que l'on visite le site Web : les pages sont traduites en français ! Oui, vous avez bien lu : PyCon, événement emblématique de la communauté Python aura bien sûr lieu en anglais, mais les informations disponibles sur le site internet sont présentes à la fois en anglais et en français. Bravo ! Quand on pense que certaines conférences internationales se déroulant en France ne prennent même pas cette peine...

Refermons cette parenthèse de défense de la langue française pour nous intéresser à la conférence en elle-même. Elle se déroulera sur six jours avec différentes sessions : tutoriels, conférences, posters, « sprints » (sessions de développement intensif sur des projets open source), etc. Le programme n'est bien entendu pas encore disponible et l'appel à contribution est fermé. Pourquoi donc vous en parler maintenant, quatre mois avant l'ouverture de l'événement ? Simplement pour que vous ayez le temps de vous organiser si vous avez la chance de pouvoir faire le voyage et de participer à la plus grande conférence sur Python de l'année (et de pouvoir visiter le Québec par la même occasion...).

SOMMAIRE

GNU/LINUX MAGAZINE FRANCE
N° 177

Appel à contribution ScilabTEC 2015

Date : les 21 et 22 mai 2015

Deadline : 2 janvier 2015

Lieu : Paris - France (en anglais)

Site officiel : http://www.scilabtec.com/index.php/papers

La 7ème conférence internationale des utilisateurs de Scilab, le logiciel de calcul numérique, et de Xcos, l'éditeur graphique de modèles de systèmes dynamiques hybrides, aura lieu les 21 et 22 mai 2015.

L'appel à contributions a été lancé est restera ouvert jusqu'au 2 janvier 2015. Pour cette édition, les présentations attendues devront démontrer le potentiel de Scilab/Xcos pour résoudre des problèmes de la vie réelle avec une préférence pour les présentations d'applications industrielles et les présentations de nouveaux modules externes Scilab, le tout dans les principaux domaines du calcul numérique : automobile, aéronautique, espace, énergie, défense, télécommunications, biomédical, finance, transports, environnement, etc.

Vous avez donc jusqu'au 2 janvier pour soumettre vos propositions par mail à scilabtec@scilab-enterprises.com (les modèles de soumission sont disponibles sur le site officiel). ■

POSTGRESQL 9.4, ENCORE QUELQUES NOUVEAUTÉS

par ***Guillaume Lelarge*** *[Consultant technique chez Dalibo - Contributeur à pgAdmin]*

La dernière fois, nous avons discuté des nouveautés concernant la réplication et les fonctionnalités utilisateurs. Cette fois, nous ciblerons l'administration du serveur et sa supervision. Nous discuterons aussi des performances, pour finir sur les perspectives d'avenir.

1 Administration

1.1 Tablespaces

La version 9.4 améliore légèrement la façon dont un administrateur peut gérer les *tablespaces*.

La nouvelle fonctionnalité la plus importante concerne le déplacement de tous les objets d'un certain type. Avant la 9.4, il était possible de déplacer objet par objet d'un tablespace vers un autre. Il était aussi possible de déplacer tous les objets d'une base de données du tablespace par défaut vers un autre. Là, il s'agit de pouvoir déplacer tous les objets d'un certain type. Par exemple, si nous voulons déplacer toutes les tables du tablespace **petitdisque** vers le tablespace **grosdisque**, nous utiliserons la requête suivante :

```
ALTER TABLE ALL IN TABLESPACE petitdisque SET TABLESPACE grosdisque;
```

Cela fonctionne aussi pour les index et les vues matérialisées.

Il est possible d'utiliser deux options avec cette nouvelle version du **ALTER TABLE** :

- la clause **OWNED BY** permet de ne déplacer que les tables d'un certain propriétaire ;
- la clause **NOWAIT** permet d'annuler la requête s'il n'est pas possible d'obtenir immédiatement les verrous requis sur tous les objets concernés.

L'autre nouveauté concerne l'ordre **CREATE TABLESPACE**. Il permet de configurer directement les options à la création du tablespace :

```
CREATE TABLESPACE grosdisque
  LOCATION '/mnt/mongrosdique'
  WITH (random_page_cost = 1.5);
```

1.2 Configuration

Il y a trois types de nouveautés sur la configuration :

- un nouvel ordre SQL ;
- de nouveaux paramètres ;
- et de nouvelles valeurs par défaut.

Commençons par le plus excitant : le nouvel ordre SQL. La modification de la configuration passe par une modification du fichier **postgresql.conf**. Avant la version 9.4, il était nécessaire de se connecter sur le serveur via SSH, et de modifier le fichier de configuration en utilisant son éditeur de texte favori. Il était aussi possible d'utiliser l'extension **adminpack** pour le faire, mais c'était assez compliqué. La version 9.4 ajoute un ordre SQL appelé **ALTER SYSTEM** permettant de modifier le fichier de configuration à distance. Pour être plus précis, il ne modifie pas le fichier **postgresql.conf**, mais le fichier **postgresql.auto.conf**. Ce fichier ne contient que des configurations faites par la nouvelle instruction SQL. Par exemple, si on souhaite changer le paramétrage de **checkpoint_segments** pour le faire passer à **20**, voici ce qu'il faut faire :

```
pagila=# ALTER SYSTEM SET checkpoint_segments TO 20;
ALTER SYSTEM
```

Ce qui résulte en la modification du fichier **postgresql.auto.conf** :

```
$ ll $PGDATA/postgresql*.conf
-rw-------. 1 guillaume guillaume   111 Aug 29 15:03 /opt/
postgresql/datas/glmf/postgresql.auto.conf
-rw-------. 1 guillaume guillaume 21203 Aug 29 14:30 /opt/
postgresql/datas/glmf/postgresql.conf
$ cat /opt/postgresql/datas/glmf/postgresql.auto.conf
# Do not edit this file manually!
# It will be overwritten by ALTER SYSTEM command.
checkpoint_segments = '20'
```

Ce fichier est réécrit entièrement à chaque fois. Seules apparaissent les deux lignes de commentaire et les paramètres configurés de cette façon. Tout autre ajout est ignoré.

Ce fichier est toujours lu en dernier, il surcharge donc le contenu du fichier **postgresql.conf**.

Un gros avantage de l'utilisation de cette nouvelle commande est que les modifications réalisées de cette façon sont validées avant d'être enregistrées :

```
pagila=# ALTER SYSTEM SET wal_level TO 'toto';
ERROR:  invalid value for parameter "wal_level": "toto"
HINT:  Available values: minimal, archive, hot_standby, logical.
```

Si cela était fait directement dans le fichier de configuration, on ne se serait aperçu de l'erreur qu'au redémarrage de PostgreSQL, ce qui aurait causé une immobilisation du serveur plus importante.

Il est à noter que la configuration n'est pas prise en compte immédiatement. Il faut demander à PostgreSQL de recharger sa configuration, voire, dans certains cas, de redémarrer.

En ce qui concerne les nouveaux paramètres, ils sont au nombre de deux (si on ne tient pas compte des autres nouveaux paramètres présentés dans le reste de ce document) :

- **huge_pages**, pour activer la gestion des *huge pages* avec PostgreSQL ;
- **autovacuum_work_mem**, qui surcharge la valeur de **maintenance_work_mem** au niveau du processus **autovacuum**.

La trace réalisée par le paramètre **log_lock_waits** contient plus d'informations, avec notamment le PID du processus qui détient le verrou bloquant et celui du processus en attente du verrou.

Quant aux nouvelles valeurs par défaut, elles concernent trois paramètres : **work_mem** passe de 1 Mi à 4 Mi ; **maintenance_work_mem** passe de 16 Mi à 64 Mi ; enfin, **effective_cache_size passe** de 128 Mi à 4 Gi.

Dernier point à noter, dans les unités de volumétrie, il est « enfin » possible d'indiquer TB pour téraoctets. Mais il y a peu de risque qu'on utilise rapidement l'unité TB pour les paramètres mémoire, malheureusement :-)

1.3 Restauration PITR

Lors d'une restauration PITR, il y avait quatre cibles finales de restauration :

- jusqu'à la dernière transaction (par défaut) ;
- jusqu'à une certaine date (**recovery_target = 'time'**) ;
- jusqu'à un certain identifiant de transaction (**recovery_target = 'xid'**) ;
- jusqu'à un certain point de restauration (**recovery_target = 'name'**).

Une cinquième cible est ajoutée en version 9.4 : **recovery_target = 'immediate'**. Son but est d'arrêter la restauration dès que cette dernière a atteint le point de cohérence établi par la fonction **pg_stop_backup()**.

Voici un exemple complet de restauration utilisant la nouvelle cible.

Commençons par modifier la configuration de l'instance pour activer l'archivage :

```
$ psql
psql (9.4beta2)
Type "help" for help.

postgres=# ALTER SYSTEM SET wal_level TO 'archive';
ALTER SYSTEM
postgres=# ALTER SYSTEM SET archive_mode TO on;
ALTER SYSTEM
postgres=# ALTER SYSTEM SET archive_command TO 'cp %p /opt/
postgresql/archives/r94/%f';
ALTER SYSTEM
postgres=# \q
```

Créons maintenant le répertoire d'archivage, et donnons les droits à l'utilisateur qui lance PostgreSQL :

```
$ sudo mkdir -p /opt/postgresql/archives/r94
[sudo] password for guillaume:
$ sudo chown guillaume:guillaume /opt/postgresql/archives/r94
```

Il ne reste plus qu'à redémarrer PostgreSQL pour prendre en compte les modifications :

```
$ /etc/init.d/postgresql restart
server stopped
Cluster glmf started.
```

Créons une nouvelle base avec l'outil **pgbench**, ce qui nous permettra de créer facilement un peu d'activité sur le serveur :

```
$ createdb benchs
$ pgbench -i -s10 benchs
NOTICE:  table "pgbench_history" does not exist, skipping
NOTICE:  table "pgbench_tellers" does not exist, skipping
NOTICE:  table "pgbench_accounts" does not exist, skipping
NOTICE:  table "pgbench_branches" does not exist, skipping
creating tables...
100000 of 1000000 tuples (10%) done (elapsed 0.08 s, remaining 0.70 s).
200000 of 1000000 tuples (20%) done (elapsed 0.56 s, remaining 2.24 s).
300000 of 1000000 tuples (30%) done (elapsed 1.13 s, remaining 2.64 s).
400000 of 1000000 tuples (40%) done (elapsed 1.42 s, remaining 2.12 s).
500000 of 1000000 tuples (50%) done (elapsed 1.82 s, remaining 1.82 s).
600000 of 1000000 tuples (60%) done (elapsed 2.46 s, remaining 1.64 s).
700000 of 1000000 tuples (70%) done (elapsed 2.83 s, remaining 1.21 s).
800000 of 1000000 tuples (80%) done (elapsed 3.82 s, remaining 0.95 s).
900000 of 1000000 tuples (90%) done (elapsed 5.17 s, remaining 0.57 s).
1000000 of 1000000 tuples (100%) done (elapsed 5.27 s, remaining 0.00 s).
vacuum...
set primary keys...
done.
$ psql
psql (9.4beta2)
Type "help" for help.

postgres=# \x
Expanded display (expanded) is on.
postgres=# SELECT * FROM pg_stat_archiver ;
 -[ RECORD 1 ]------+--------------------------
archived_count     | 12
last_archived_wal  | 000000010000000000000033
last_archived_time | 2014-08-29 15:07:19.25632+02
failed_count       | 0
last_failed_wal    |
last_failed_time   |
stats_reset        | 2014-08-29 14:31:36.866948+02

postgres=# \q
```

Grâce à la vue **pg_stat_archiver**, nous voyons que l'archivage fonctionne (il s'agit d'une nouvelle vue qui est expliquée un peu plus bas dans cet article). Lançons la sauvegarde des fichiers et créons une table pendant cette sauvegarde :

```
$ psql
psql (9.4beta2)
Type "help" for help.

postgres=# SELECT pg_start_backup('test 9.4', true);
 pg_start_backup
-----------------
 0/35000028
(1 row)

postgres=# \q
$ cd $PGDATA/..
$ cp -r glmf glmf2
$ psql
psql (9.4beta2)
Type "help" for help.

postgres=# CREATE TABLE table_pendant_sauvegarde(id integer);
CREATE TABLE
postgres=# SELECT pg_stop_backup();
NOTICE:  pg_stop_backup complete, all required WAL segments have
been archived
 pg_stop_backup
----------------
 0/35012238
(1 row)
```

Maintenant que la sauvegarde s'est terminée, créons une deuxième table et lançons de nouveau **pgbench** pour réaliser quelques nouvelles écritures.

```
postgres=# CREATE TABLE table_apres_sauvegarde(id integer);
CREATE TABLE
postgres=# \q
$ pgbench -c 10 -T 60 benchs
starting vacuum...end.
transaction type: TPC-B (sort of)
scaling factor: 10
query mode: simple
number of clients: 10
number of threads: 1
duration: 60 s
number of transactions actually processed: 10755
latency average: 55.788 ms
tps = 179.058065 (including connections establishing)
tps = 179.089032 (excluding connections establishing)
```

Enfin, créons une troisième table :

```
$ psql
psql (9.4beta2)
Type "help" for help.

postgres=# CREATE TABLE table_apres_pgbench(id integer);
CREATE TABLE
postgres=# \q
```

Au niveau du répertoire de copie, le fichier **postgresql.conf** a été modifié pour changer le numéro de port et le fichier **recovery.conf** a été modifié ainsi :

```
restore_command = 'cp /opt/postgresql/archives/r94/%f %p'
recovery_target = 'immediate'
```

Après avoir fait le ménage habituel (**rm postmaster.pid pg_xlog/***), le serveur de copie a été démarré. Voici ce qu'indique le fichier de traces :

```
LOG:  database system was interrupted; last known up at 2014-08-
29 15:08:24 CEST
LOG:  starting point-in-time recovery to earliest consistent point
LOG:  restored log file "000000010000000000000035" from archive
LOG:  redo starts at 0/35000090
LOG:  consistent recovery state reached at 0/35012238
LOG:  recovery stopping after reaching consistency
LOG:  redo done at 0/35012210
LOG:  last completed transaction was at log time 2014-08-29
15:09:59.71341+02
cp: cannot stat '/opt/postgresql/archives/r94/00000002.history':
No such file or directory
LOG:  selected new timeline ID: 2
cp: cannot stat '/opt/postgresql/archives/r94/00000001.history':
No such file or directory
LOG:  archive recovery complete
LOG:  database system is ready to accept connections
LOG:  autovacuum launcher started
```

PostgreSQL a rejoué tous les journaux (ici un seul) entre le **pg_start_backup()** et le **pg_stop_backup()**. On s'en rend compte plus facilement en regardant les tables existantes :

```
$ psql
psql (9.4beta2)
Type "help" for help.

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

Seules les données modifiées avant et pendant la sauvegarde ont été récupérées. Toutes les modifications après la sauvegarde ayant généré des journaux de transactions n'ont pas été prises en compte. Autrement dit, seule la table **table_pendant_sauvegarde** est présente. Les deux autres tables, **table_apres_sauvegarde** et **table_apres_pgbench**, n'ont pas été récréées. C'est le but recherché de la cible immédiate.

1.4 Background workers

Les *background workers* introduits en version 9.3 ont évolué assez fortement en 9.4. Ils ont maintenant la possibilité d'être lancés et arrêtés à tout moment. Ils ont aussi la possibilité d'allouer dynamiquement des segments de mémoire partagée.

Comme il est possible de lancer des background workers à la demande, il fallait quand même avoir la possibilité de limiter le nombre de processus de ce type. C'est chose faite, avec le paramètre **max_worker_processes**. Même si ce paramètre a principalement pour but d'éviter de lancer trop de background workers au fil de l'exécution du serveur, il limite aussi le nombre de background workers lancés traditionnellement au démarrage du serveur. Il conviendra donc de prendre en compte les processus statiques et dynamiques dans le paramétrage de cette variable.

2 Supervision

PostgreSQL a encore du chemin à faire dans le domaine de la supervision. La version 9.4 améliore cela en proposant une nouvelle vue statistique et en ajoutant quelques colonnes à des vues existantes.

2.1 Vue pg_stat_archiver

La nouvelle vue **pg_stat_archiver** a pour but d'indiquer le nombre de journaux de transactions dont l'archivage a réussi, le nombre de ceux dont l'archivage a échoué, ainsi que les moments où les derniers archivages réussis et échoués ont eu lieu.

```
postgres=# SELECT * FROM pg_stat_archiver ;
 -[ RECORD 1 ]------+------------------------------
 archived_count     | 8
 last_archived_wal  | 000000010000000000000008
 last_archived_time | 2014-06-19 18:20:12.167006+02
 failed_count       | 0
 last_failed_wal    |
 last_failed_time   |
 stats_reset        | 2014-06-19 18:17:44.585545+02
```

La requête suivante indique si le dernier archivage a réussi ou échoué et le journal concerné :

```
SELECT 'WAL ' ||
case when last_archived_time > last_failed_time then last_
archived_wal else last_failed_wal end ||
case when last_archived_time > last_failed_time then 'succès'
else 'échoué' end
AS message
FROM pg_stat_archiver ;
```

2.2 Nouvelles colonnes

La colonne **n_mod_since_analyze** a été ajoutée dans la vue **pg_stat_all_tables**. Cette colonne indique le nombre de lignes insérées, modifiées et supprimées depuis le dernier **ANALYZE** de la table. C'est très utile pour savoir si les statistiques sur les données sont plus ou moins à jour et si l'autovacuum va bientôt se déclencher sur cette table.

La vue statistique **pg_stat_activity** récupère deux nouvelles colonnes (**backend_xid** et **backend_xmin**). Le premier est l'identifiant de transaction courant pour cette session, alors que le second représente la vision de la base pour cette session.

La vue statistique **pg_stat_replication** a aussi récupéré la colonne **backend_xmin**. Par contre, la colonne **backend_xid** n'a pas d'intérêt, vu qu'un esclave ne peut pas faire de modifications suite à des ordres DML ou DDL, donc elle n'a pas été ajoutée.

3 Performances

3.1 pg_prewarm, pour préchauffer le cache

pg_prewarm est une nouvelle extension de PostgreSQL. Son but est de préchauffer le cache disque, notamment après un redémarrage.

Cette extension contient uniquement une procédure, appelée elle aussi **pg_prewarm**, qui lira la relation pour la mettre en cache, soit au niveau du système d'exploitation, soit au niveau de PostgreSQL. Elle prend en argument :

- le nom de la table ;
- le mode de lecture (**buffer** par défaut) ;
- le type de fichier (**main** par défaut) ;
- le premier bloc à lire (**NULL** par défaut, donc depuis le début) ;
- le dernier bloc à lire (**NULL** par défaut, donc jusqu'à la fin).

Prenons comme exemple la table **rental**. Le serveur PostgreSQL a été redémarré pour que le cache soit à froid (autrement dit vide).

Commençons par ajouter deux extensions (**pg_buffercache** pour connaître le contenu du cache et **pg_prewarm** pour la tester) :

```
pagila=# CREATE EXTENSION pg_buffercache;
CREATE EXTENSION
pagila=# CREATE EXTENSION pg_prewarm;
CREATE EXTENSION
```

Regardons maintenant le contenu du cache pour la table **rental** :

```
pagila=# SELECT count(bc.*)FROM pg_buffercache bc join pg_class
c on bc.relfilenode=c.relfilenode where c.relname='rental';
 count
-------
     0
(1 row)
```

La table **rental** contient 0 bloc en cache. Donc la table n'est pas en cache, ce qui est logique après un redémarrage du serveur PostgreSQL. Chargeons la table avec l'extension :

```
pagila=# select pg_prewarm('rental');
 pg_prewarm
------------
        150
(1 row)
pagila=# select count(bc.*) from pg_buffercache bc join pg_class
c on bc.relfilenode=c.relfilenode where c.relname='rental';
 count
-------
   150
(1 row)
```

150 blocs ont été placés en cache, ce que confirme l'extension **pg_buffercache**. Un **EXPLAIN (ANALYZE, BUFFERS)** nous le confirme aussi :

```
pagila=# EXPLAIN (ANALYZE, BUFFERS, COSTS off) select * from
rental;
                          QUERY PLAN
---------------------------------------------------------------
----
 Seq Scan on rental  (actual time=0.028..5.551 rows=16044
loops=1)
   Buffers: shared hit=150 dirtied=147
 Execution time: 7.618 ms
(4 rows)
```

Grâce à cette extension, il est possible de monter rapidement en mémoire les tables dont on a besoin. C'est surtout utile au démarrage de PostgreSQL, vu qu'à ce moment son cache est totalement vide, mais ça pourrait se révéler utile dans d'autres cas.

Même si cet outil n'est pas en soi un outil de sauvegarde et de restauration du cache de PostgreSQL, il est assez simple d'écrire un script le faisant en utilisant cette extension.

3.2 Nouveautés de la commande EXPLAIN

La commande **EXPLAIN** bénéficie aussi de quelques améliorations intéressantes.

Tout d'abord, elle affiche enfin le temps de planification. Ce dernier est généralement bien inférieur au temps d'exécution, mais dans le cas des requêtes complexes, il peut devenir prédominant. Son affichage dépend de l'option **COSTS** (donc activé par défaut). Voici un exemple sur une extraction simple de la table **staff** :

```
pagila=# EXPLAIN SELECT * FROM staff ORDER BY first_name;
                          QUERY PLAN
---------------------------------------------------------------
 Sort  (cost=16.39..16.74 rows=140 width=531)
   Sort Key: first_name
   ->  Seq Scan on staff  (cost=0.00..11.40 rows=140 width=531)
 Planning time: 0.093 ms
(4 rows)
```

Une autre amélioration de la commande **EXPLAIN** concerne les nœuds Agg et Group (pour les agrégats et les regroupements). Les colonnes concernées par ces nœuds sont indiquées clairement :

```
pagila=# EXPLAIN SELECT substr(last_name, 1, 1), count(*) FROM
actor GROUP BY 1 ORDER BY 1;
                          QUERY PLAN
---------------------------------------------------------------
----
 Sort  (cost=11.20..11.50 rows=121 width=7)
   Sort Key: (substr((last_name)::text, 1, 1))
   ->  HashAggregate  (cost=5.50..7.01 rows=121 width=7)
         Group Key: substr((last_name)::text, 1, 1)
         ->  Seq Scan on actor  (cost=0.00..4.50 rows=200
width=7)
 Planning time: 42.844 ms
(6 rows)
```

La dernière nouveauté de la commande **EXPLAIN** concerne les parcours de Bitmap Index. Elle ajoute des informations supplémentaires sur les blocs exacts et à perte :

```
pagila=# EXPLAIN (ANALYZE, COSTS off)
  SELECT * FROM rental WHERE inventory_id BETWEEN 1000 AND 2000;
                                    QUERY PLAN
-------------------------------------------------------------------
---------------------------
 Bitmap Heap Scan on rental (actual time=0.864..2.346 rows=3545
loops=1)
   Recheck Cond: ((inventory_id >= 1000) AND (inventory_id <=
2000))
   Heap Blocks: exact=150
   ->  Bitmap Index Scan on idx_fk_inventory_id (actual
time=0.815..0.815 rows=3545 loops=1)
         Index Cond: ((inventory_id >= 1000) AND (inventory_id
<= 2000))
 Execution time: 2.724 ms
(6 rows)
```

La nouvelle ligne ici est « Heap Blocks: exact=150 ».

3.3 Divers

Il y a eu aussi différentes autres améliorations plutôt importantes pour les performances :

- la fonction **pg_switch_xlog()** a été améliorée pour remplir le reste du journal de transactions par des octets nuls, dans l'idée de permettre une meilleure compression des journaux en question ;
- la possibilité pour que plusieurs processus serveurs insèrent simultanément des données dans le cache disque des journaux de transactions, ce qui améliore les performances des écritures en parallèle ;
- la possibilité de n'enregistrer dans les journaux de transactions que la portion modifiée des lignes mises à jour ;
- la tentative de gel des tables lorsqu'elles sont réécrites par les opérations **CLUSTER** et **VACUUM FULL**, ce qui évite d'avoir à faire cette opération plus tard ;
- une grosse réduction des verrous utilisés sur certaines commandes **ALTER TABLE**.

4 Régressions

Dans les régressions de cette version, il faut noter la suppression du support de l'authentification krb5. Il est conseillé d'utiliser à la place l'authentification GSSAPI.

La commande **pg_upgrade** utilisait l'option **-u** (u minuscule) pour spécifier l'utilisation de connexion, alors que tous les autres outils provenant de la distribution PostgreSQL utilisent l'option **-U** (u majuscule). Pour des soucis de cohérence, la commande **pg_upgrade** utilise maintenant l'option **-U**.

La commande **DISCARD ALL** ne tenait pas compte des séquences. Cet oubli a été réparé avec la version 9.4.

5 Et la suite ?

Deux *commit fests* de la version 9.5 ont déjà eu lieu depuis le passage en bêta de la version 9.4. Dans les patchs déjà intégrés, notons la nouvelle instruction **IMPORT FOREIGN SCHEMA**, permettant d'importer un schéma complet provenant d'un autre SGBD dans le contexte SQL/med, la possibilité de transformer une table non journalisée en table journalisée et *vice versa* (il s'agissait à la base d'un projet du *Google Summer Of Code* 2014). Sur les patches en cours, notons des travaux sur les index, sur les tris partiels, sur les triggers sur événement, sur les droits sur les lignes des tables, sur l'ajout d'une clause **LIMIT** aux instructions **DELETE** et **UPDATE**, etc.

Cependant, les projets les plus importants ne sont pas là.

EnterpriseDB travaille très fortement sur l'ajout de la parallélisation de certaines opérations dans PostgreSQL. Tous les travaux sur les background workers ont été faits avec cette idée en tête.

2ndQuadrant continue son implication dans la réplication logique. L'infrastructure est là, il manque la partie visible pour les utilisateurs. Elle devrait voir le jour pour la 9.5, voire la 9.6.

Conclusion

En conclusion de cette présentation de la version 9.4, il est intéressant de noter que le développement de PostgreSQL ne faiblit pas. Même au niveau réplication, les idées continuent à fuser et terminent généralement avec un patch sur les sources de PostgreSQL.

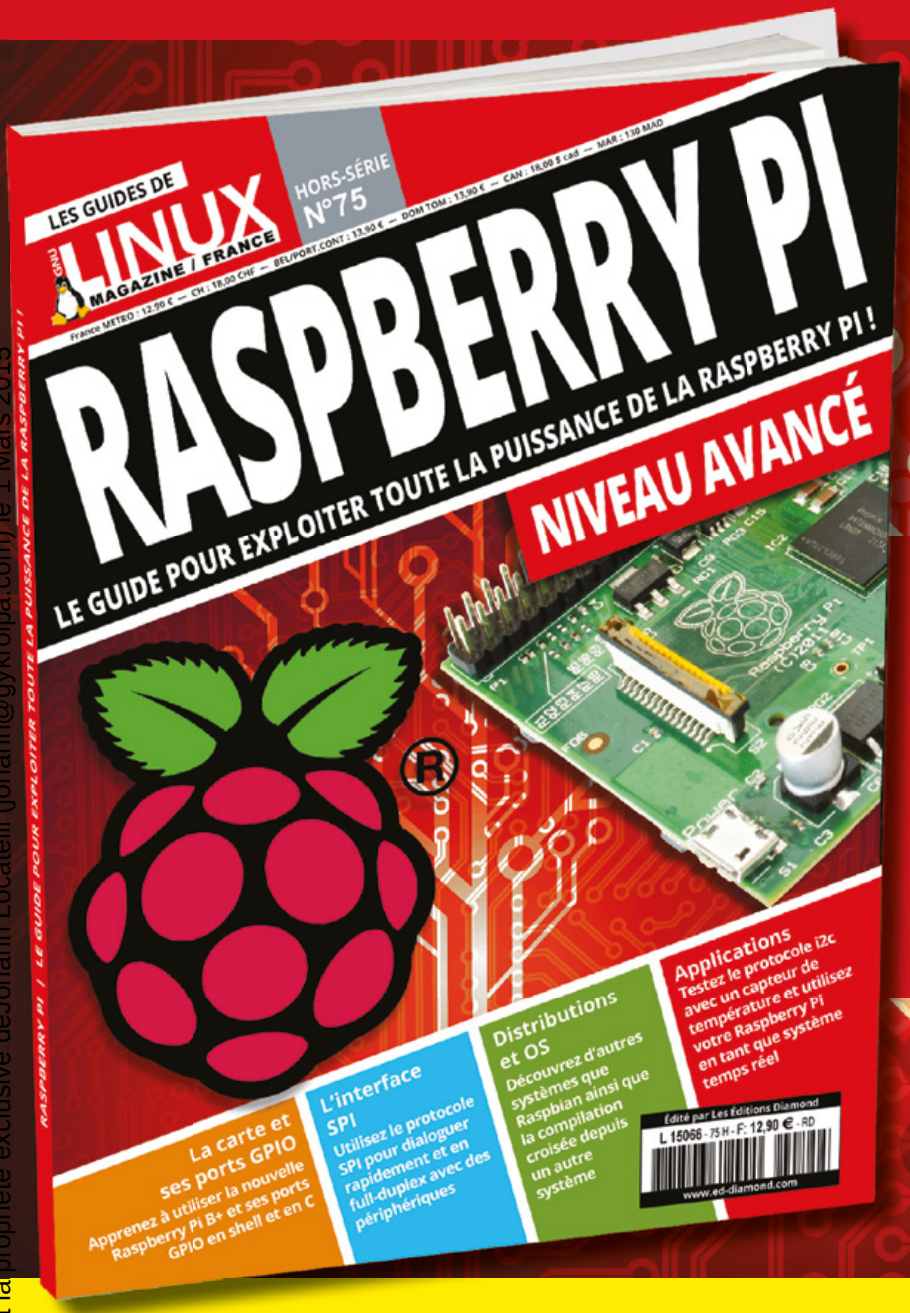
Le développement de la prochaine version a commencé fortement et de nombreuses discussions montrent la direction du projet. Au jour de la rédaction de cet article, une nouvelle discussion a été lancée par Alvarro Herrera sur le partitionnement, discussion qui semble bien partie. Avec un peu de chance, beaucoup de réflexions, de sueur et d'arguments, et un peu de code (quand même), peut-être aura-t-on un partitionnement bien plus efficace à tous points de vue dans PostgreSQL... ■

NE MANQUEZ PAS GNU/LINUX MAGAZINE HORS-SÉRIE N°75 !

DOSSIER SPÉCIAL :

RASPBERRY PI

NIVEAU AVANCÉ !



LE GUIDE POUR EXPLOITER TOUTE LA PUISSANCE DE LA RASPBERRY PI !

DISPONIBLE CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

www.ed-diamond.com

NOUVEAUTÉS DE PHP 5.6

par ***Stéphane Mourey*** *[Taohacker]*

On a beaucoup parlé sur la Toile ces derniers temps de Hack et de PHP NG, devenu futur PHP 7, qui promettent de renouveler PHP en profondeur. Mais cet enthousiasme ne doit pas faire oublier l'évolution « normale » de PHP, et la version 5.6, devenue la version recommandée à la toute fin du mois d'août, apporte son lot de nouveautés. Voyons de quoi il retourne.

Bon, disons-le tout de suite, il ne s'agit pas d'une « grande » version de PHP. Les nouveautés apportées, si elles sont sans doute utiles et importantes à connaître, ne vont pas non plus bouleverser votre façon de programmer – sauf à découvrir PHPDBG, que nous n'aborderons pas ici. Mais un peu de calme dans le monde très agité de PHP n'est pas non plus pour déplaire à l'occasion.

1 Constantes scalaires

Vous pouvez maintenant utiliser des expressions scalaires lorsque vous définissez une constante. Auparavant, il fallait utiliser une valeur statique, un nombre ou une chaîne de caractère. Désormais, vous pouvez par exemple écrire :

```
<?php
const UN = 1;
const DEUX = UN + UN;
const RESULTAT_DEUX_PLUS_DEUX =
"Deux plus deux égale : ".DEUX+DEUX;
```

Naturellement, cette expression doit fournir un résultat constant, sans quoi votre constante ne serait pas constante. Quel intérêt alors ? Pourquoi ne pas plutôt calculer soi-même le résultat une bonne fois pour toutes ? Certes, cela économiserait une ou deux instructions.

Le premier intérêt que j'y vois est d'abord *documentaire* : au-delà du résultat apparaît la logique de construction de la constante, son sens apparaît ainsi bien mieux, et le code devient bien plus facile à comprendre sur ces points.

Là où cela a le plus de sens, c'est dans la définition de valeurs par défaut d'arguments de méthodes :

```
<?php
class test {
  const MAVALEUR = 1;
  public function maMethode($a = MAVALEUR + 1) {}
  // avant PHP 5.6, nous aurions écrit :
  // public function maMethode($a = 2) {}
}
```

Dans cet exemple, nous voyons bien que la nouvelle syntaxe permet de montrer le lien entre la valeur par défaut de **`$a`** et la constante, ce qui ne pouvait apparaître auparavant.

Le second intérêt est de pouvoir définir des constantes en fonction d'autres constantes, ce qui permet, lorsque l'une des ces constantes vient à varier tout de même, de n'avoir à en redéfinir qu'une seule.

```
<?php
class test {
  const TOTO = 1;
  const TATA = TOTO +1;
  const TITI = TOTO * 3 + TATA;
}
```

Bien que les constantes ne doivent pas varier durant l'exécution, on sait bien que cela peut arriver d'une version à l'autre du code. Ici, la redéfinition de **`TOTO`** entraînera automatiquement celles de **`TATA`** et **`TITI`**, sans que j'aie besoin de les corriger. Cela permet de garantir la cohérence des constantes entre elles, et permet ainsi de diminuer l'utilisation de variables et de fonctions qui auraient justement eu ce rôle. Ainsi, auparavant, j'aurais peut-être adopté une stratégie différente, telle que celle-ci :

```
<?php
class test {
  const TOTO = 1;
  public fonction getTata(){
    return TOTO+1;
  }
  public function getTiti(){
    return self::TOTO * 3 + $this->getTata();
  }
}
```

Il est évident que cette stratégie est beaucoup moins économique que celle permise par les constantes scalaires.

2 L'opérateur ...

Mon innovation préférée de ce millésime, l'opérateur **`...`**, permet deux choses : tout d'abord de se passer de la fonction **`func_get_args`** dans les fonctions et d'autre part, de passer les éléments d'un tableau comme une suite d'arguments à une fonction sans avoir besoin de les déplier.

Commençons par son utilisation dans la définition d'une fonction. PHP permet d'utiliser des fonctions ayant un nombre variable d'arguments. Auparavant, les arguments attendus de la fonction étaient définis dans sa déclaration, et les arguments surnuméraires pouvaient être récupérés par la fonction **`func_get_args`**, qui retourne un tableau de tous les arguments passés à la fonction.

Cette technique présentait au moins deux inconvénients importants : tout d'abord, le fait que la fonction pouvait accepter des arguments supplémentaires n'était pas documenté dans la signature de la fonction, seul un examen de son code interne pouvait le révéler, ce qui est problématique ; d'autre part, comme **func_get_args()** renvoie tous les arguments, il fallait faire ensuite le tri entre ceux déjà reçus dans des variables grâce à la déclaration de la fonction et les arguments supplémentaires, d'où plus de travail pour le développeur.

Maintenant, il est possible d'utiliser cette syntaxe pour déclarer une fonction :

```
<?php
function maFonction($arg1, $arg2 =
"pipo", ...$args) {}
```

Ici, **$arg1** est un argument attendu et obligatoire, **$arg2** est un argument optionnel avec une valeur par défaut, et **$args** est un tableau contenant l'ensemble des arguments supplémentaires.

Cette syntaxe est donc particulièrement pratique lorsqu'il s'agit d'appliquer un traitement paramétré sur un nombre d'éléments indéterminés et non structurés dans un tableau.

Oui, mais justement, si mes éléments sont déjà dans un tableau, il faudra que je les en sorte alors ?

Non, car nous arrivons maintenant au deuxième cas d'utilisation de notre nouvel opérateur. Prenons donc un tableau, contenant les arguments à passer à une fonction. Auparavant, j'aurais écrit :

```
<?php
$mesArgs = [1, 2, 3];
maFonction($mesArgs[0], $mesArgs[1],
$mesArgs[2]);
```

Ce code est un peu lourd, mais surtout, il aurait été pénible de le maintenir si le nombre d'éléments du tableau était venu à changer, encore plus si ce nombre devenait variable. Aujourd'hui, j'obtiens le même résultat en écrivant :

```
<?php
$mesArgs = [1, 2, 3];
maFonction(...$mesArgs);
```

Ce qui est quand même beaucoup plus élégant. Et si j'ai un ou plusieurs arguments qui ne sont pas dans mon tableau, je peux les passer avant de cette façon :

```
<?php
$mesArgs = [1, 2, 3];
maFonction("monPremierArgument",
"monSecondArgument", ...$mesArgs);
```

L'un dans l'autre, l'opérateur **...** va soulager le développeur PHP d'un exercice récurrent et pénible : la mise en tableau et le dépliage de tableau d'arguments !

3 Import de fonctions et de constantes

Il était déjà possible d'importer une classe d'un espace de nom à un autre en lui attribuant un alias via l'instruction **use**. Cette instruction a été généralisée et permet maintenant l'import de constantes et de fonctions.

```
<?php
namespace GMLF;
use function Taophp\maFonction as TaophpMaFonction;
use const Taophp\maConstante as TaophpMaConstante;
```

Les deux dernières lignes de cet extrait de code importent **maFonction()** et la constante **maConstante** depuis l'espace de nom **Taophp** dans l'espace de nom **GMLF**, en les renommant. Notez que l'usage de ces imports diffère de celui des classes, dans la mesure où il est nécessaire de faire suivre le mot-clé **use** d'un autre indiquant ce que l'on veut importer, constante ou fonction.

4 Intégration de PHPDBG

PHPDBG est un débogueur pour PHP. Cet outil, qui a vécu sa vie avant d'être intégré dans PHP, mérite un article entier à lui tout seul. Nous en reparlerons donc. Ceux qui ne peuvent pas attendre pourront se faire les dents sur la documentation officielle : http://phpdbg.com/docs.

5 Amélioration concernant la sécurité et la cryptographie

N'étant pas expert, je ne m'étendrai pas sur ces améliorations qui, au fond, regardent plus l'administrateur de site que le développeur. Signalons-les tout de même pour faire mentir ceux qui affirment que PHP est un gruyère et que ses concepteurs ne se préoccupent pas de la sécurité.

Ainsi est apparue la fonction **hash_equals**, qui permet de vérifier la conformité d'une chaîne avec une signature préalablement calculée en utilisant **crypt()**, mais en utilisant un temps constant. En effet, la fonction **crypt** est vulnérable à des attaques temporelles (ou *timing attack*, voir http://fr.wikipedia.org/wiki/Attaque_temporelle pour plus de détails), qui se basent sur le temps de traitement d'un cryptage pour essayer de le casser. Notons en passant qu'il est recommandé d'utiliser les fonctions **password_hash** et **password_verify**, introduites dans la version 5.5 et qui sont dès l'origine insensibles aux attaques temporelles.

Notons également l'amélioration du support de SSH/TLS. Ces améliorations, nombreuses et pointues, font l'objet d'une page dédiée : http://fr2.php.net/manual/fr/migration56.openssl.php.

Conclusion

Voilà pour les innovations qui m'ont paru les plus marquantes. Il en reste peut-être quelques autres qui vous intéresseront, comme un nouvel opérateur pour les puissances ou l'acceptation de fichiers de plus de 2 Gi en *upload*. Retrouvez la liste complète sur le site officiel à cette adresse : http://fr2.php.net/manual/fr/migration56.new-features.php. ■

FOUTAISES PARLEMENTAIRES – PETITES QUESTIONS ENTRE AMIS

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

Si dans l'hémicycle, la majorité des discussions durant le mois d'octobre ont concerné le projet de loi de financement de la sécurité sociale, le numérique n'a pas pour autant été oublié, comme en atteste certaines questions écrites au Gouvernement. Petit florilège.

1 Sylviane Bulteau et les petites annonces

1.1 Ce qu'elle a dit

Dans une question écrite en date du 21 octobre 2014, la députée socialiste de la deuxième circonscription de Vendée interroge le Ministre des Finances sur la pratique déloyale des sites de petites annonces immobilières, qui porteraient préjudice aux agents immobiliers ainsi qu'à l'État.

1.2 Ce qu'il en est réellement

Qu'on se le dise : les petites annonces existent en France depuis 1631. Initialement, elles servaient à financer les publications. Avec l'essor de la presse puis du Web, les petites annonces et leurs supports se sont multipliés : électroménager, automobiles, emploi, rencontres amoureuses et sexuelles et biens immobiliers. Le désormais célèbre hebdomadaire *De Particulier à Particulier* existe depuis 1975 et s'est décliné en version Web afin que des personnes puissent vendre directement leurs biens sans intermédiaires. Il semble donc étonnant que la députée se soit émue d'une situation finalement très ancienne.

Elle invoque notamment deux arguments : la concurrence déloyale envers les agents immobiliers et la perte de recettes fiscales pour l'État. En effet, lorsqu'un agent réalise une transaction, il est assujetti à la taxe sur la valeur ajoutée, notre fameuse TVA, recette qui échapperait à l'État si la transaction est réalisée entre particuliers.

Cette affirmation serait parfaitement exacte si elle n'omettait pas la plus-value immobilière. En effet, généralement, les biens immobiliers prennent de la valeur avec le temps, permettant à certains propriétaires de réaliser une plus-value et c'est cette fameuse plus-value qui fait l'objet d'un correctif, sous forme de taxes pour l'État, même s'il existe quelques exonérations fiscales.

Enfin, même entre particuliers, les ventes de biens immobiliers sont obligatoirement réalisées avec l'aide d'un notaire puisque la loi l'impose. Or, le recours à un notaire n'est pas gratuit et les honoraires d'un notaire sont également soumis au paiement de 20% de TVA. L'argument de la perte de recette pour l'État est donc fallacieux.

Concernant la concurrence déloyale envers les agents immobiliers, là encore, l'argument ne tient pas. En effet, les agents immobiliers peuvent être salariés en CDI avec un salaire fixe et des commissions sur les ventes ou être en autoentreprise et, de façon un peu imagée, courir derrière les commissions qu'ils peuvent réaliser sur les ventes. C'est donc à eux de trouver les clients et les biens, dans les meilleures conditions possibles, pour leur compte et pour une agence, ce qui s'apparente à un CDI déguisé et, dans la mesure où les charges patronales échappent à l'URSSAF, à du travail dissimulé. D'ailleurs, il est souvent amu-

sant de constater que sur la plupart des sites d'annonces entre particuliers, que ce soit *PAP*, *Le bon coin* ou autre, on trouve beaucoup (trop) d'annonces déposées par des agents immobiliers.

1.3 Ce qu'elle sous-entend

On peut trouver curieux qu'une parlementaire socialiste jette un pavé dans la mare concernant le marché immobilier et le sort des agents immobiliers au moment où le pouvoir d'achat des Français est au plus bas, qu'il existe une réelle crise du logement, notamment en Ile-De-France et que l'actuelle Ministre de l'Économie, des Finances et du Numérique planche sur la déréglementation de certaines professions notamment les avocats et les notaires, afin de casser les monopoles. Or, en demandant au Ministre de réglementer un soi-disant nouvel usage, dont on a démontré qu'il n'avait rien de nouveau, on constate le décalage au sein d'un même parti sur la question des monopoles et une certaine méconnaissance des fameux sites d'annonces. Enfin, on peut trouver curieux qu'une assistante sociale en détachement, administratrice dans un organisme de logements sociaux en Vendée ne comprenne pas que des particuliers préfèrent se passer d'agents immobiliers, notamment pour éviter d'avoir à perdre 10% de la vente au profit d'un intermédiaire. Phénomène encore plus curieux, un autre député, Jacques Cresta a posé une question similaire une semaine après, venant en soutien à la députée socialiste.

2 Les avocats en ligne de Joël Giraud

2.1 Ce qu'il a dit

Dans une question écrite à la Secrétaire d'État, auprès du ministre de l'économie, de l'industrie et du numérique, chargée du numérique le 21 octobre 2014, le député radical Joël Giraud de la deuxième circonscription des Hautes-Alpes soulève une question épineuse : celle des sites de conseils juridiques en ligne. Sur cette question, il convient, non pas de soulever d'éventuelles incohérences dans le propos du député, mais d'expliquer pourquoi il existe aujourd'hui un vrai débat sur cette question.

2.2 Ce qu'il en est

Depuis environ 5 ans, un certain nombre de sites Web se sont ouverts, proposant à des internautes de fournir des conseils juridiques, voire des moyens de recours et/ou des plaidoiries, clefs en main, pour une somme moindre que s'ils passaient par des avocats. Naïvement, on pourrait se dire que c'est une bonne idée dans les cas simples comme le divorce par consentement mutuel, les demandes d'exonération gracieuses auprès des services fiscaux ou les questions relatives aux infractions au Code de la route. Plutôt que de chercher, de se déplacer et de payer une consultation, on dépose directement sa question à travers un formulaire adéquat et en échange d'une somme modique, on obtient rapidement une réponse.

Or, comme l'a rappelé le Conseil National des Barreaux dans une lettre adressée à Arnaud Montebourg en date du 11 juillet 2014, les avocats disposent d'un monopole : celui de représenter, d'assister et de défendre les justiciables, devant certaines juridictions. Il faut raisonner par opposition : un juriste peut conseiller des justiciables, mais il ne peut les représenter, les assister et les défendre devant certaines juridictions. Certaines juridictions comme les Prud'hommes admettent que ces fonctions soient remplies par une autre catégorie de personnes. Mais devant les juridictions civiles et pénales, le recours à un avocat est obligatoire, sauf à se représenter soi-même.

En donnant des « kits » de défense devant des juridictions à des justiciables, ces sites violent ce fameux monopole, ce qui est un premier problème. Le deuxième problème concerne la déontologie. Pendant la formation d'avocat, la déontologie est enseignée et fait l'objet d'un examen. Lorsque les avocats prêtent serment, ils jurent de respecter cette déontologie et si, au cours de leur carrière, ils violent les règles de la profession, ils sont passibles de sanctions disciplinaires pouvant aller jusqu'à la radiation de l'ordre. Un simple juriste n'est pas soumis aux mêmes règles. De la même façon, un avocat a un droit assez limité quant au recours à la publicité. Il peut bien sûr, avoir un blog, anonyme ou non, des comptes sur les réseaux sociaux, mais cela ne doit pas lui servir à gagner directement une clientèle et il ne doit pas s'en servir pour exercer sa mission de conseil. Là encore, la publicité est très strictement encadrée et dans certains cas, doit faire l'objet d'une autorisation de l'Ordre des Avocats. Enfin, il faut savoir que les avocats, lorsqu'ils ne respectent pas l'obligation de diligence, de dévouement, de sérieux, de disponibilité, de prudence, d'indépendance, de dignité et de délicatesse peuvent faire l'objet de sanctions. De façon

plus imagée, si vous avez été mal conseillé par un avocat, vous pouvez l'attaquer et mettre en jeu, non seulement sa responsabilité civile, mais aussi sa responsabilité disciplinaire. Enfin et cela relève plus de l'argument de bon sens qu'autre chose, si pour être avocat, il faut faire un minimum de huit ans d'études, ce n'est pas uniquement pour surpeupler les facultés.

On peut comprendre l'argument qui consiste à dire que l'accès à un avocat peut être difficile, notamment sur le plan financier, mais il est des professions qu'il faut libéraliser à bon escient et même les affaires qu'on imagine simples à résoudre comme le divorce par consentement mutuel peuvent s'avérer compliquées.

3 La culture de Fleur Pellerin

Le mois d'octobre 2014 ne sera pas à marquer d'une pierre blanche pour l'actuelle Ministre de la Culture et de la Communication. Entre son aveu concernant le Prix Nobel de Littérature et son envie de taxer les cartouches d'encre, la rue de Valois n'a pas brillé. Mais c'est son bras de fer avec l'institution la plus détestée de France qui va nous intéresser.

3.1 Une histoire de budget

Qu'on se le dise, l'Hadopi est l'institution la plus détestée de France par les internautes, victime d'un péché originel que nous devons conjointement à madame Albanel et à l'ancien Président de la République, Nicolas Sarkozy. Depuis 2012, les jours de l'institution sont incertains et si François Hollande avait promis de régler le sort de cette autorité, à l'heure actuelle, il n'en est rien. Difficile de supprimer une institution, tant sur le plan politique qu'institutionnel. On ne supprime pas des institutions en France : on les fait fusionner avec d'autres, on les vide de leurs substances, mais on ne les supprime pas. Comment se débarrasser de cette épine ? Fleur Pellerin semblait avoir trouvé la solution en réduisant le budget de l'institution à 6 millions d'Euros. Dit comme ça, la somme paraît gigantesque. En réalité, c'est une goutte d'eau dans le budget de l'État. Mais ces six millions représentent la quasi-totalité du budget de fonctionnement de la réponse graduée, dont le coût est de 5,6 millions d'Euros.

Apprenant cela, l'institution a publiquement annoncé son intention de faire fonctionner a minima la réponse graduée afin de ne pas assécher le fonctionnement de ses autres missions, à savoir le développement de l'offre légale, la régulation des DRM et l'étude des usages. Raisonnement logique de la part de l'institution puisque le développement de l'offre légale et la régulation sont dans la loi. Ne l'entendant pas de cette oreille, la ministre a essayé de rappeler à l'ordre l'autorité en lui disant qu'elle devait continuer à faire tourner la machine de la réponse graduée même si cela se fait au détriment des autres missions.

3.2 Bras de fer

Si sous l'égide de Frédéric Mitterrand, l'institution essayait de rester la plus lisse possible, à partir de juin 2012, un vent de révolte a soufflé à l'Hadopi. Pour parler vulgairement, foutu pour foutu, faisons enfin ce qui nous plaît. Une cellule de recherche entièrement dédiée aux usages a été créée, le site PUR a (enfin !) été supprimé pour laisser la place à l'offre légale avec son outil de référencement des contenus indisponibles légalement, le chantier des licences libres a été abordé et le fameux rapport intermédiaire sur la rémunération proportionnelle du partage a été publié. Fameux, car il a fait grincer des dents les ayants-droits. Il faut dire que dans l'esprit de certains, l'Hadopi devait être leur joujou bien docile. Avec l'annonce de la réduction du budget, l'ensemble du personnel de l'Hadopi a fait savoir qu'il se mettrait en grève si le budget n'était pas augmenté de 2,5 millions, ce qui aurait pour conséquence de stopper la réponse graduée. À l'heure où ces lignes sont écrites, le sort de l'institution est encore incertain.

3.3 Un jeu dangereux

Après l'élection de François Hollande, il y avait une guerre ouverte entre Fleur Pellerin et Aurélie Filippetti pour le ministère de la Culture et de la Communication et quand le portefeuille de la rue de Valois a changé de mains, on pouvait s'attendre à des approches plus innovantes, plus en adéquation avec les attentes, à la fois du monde de la culture et des internautes. En souhaitant préserver à tout prix la réponse graduée, Fleur Pellerin semble surtout vouloir faire plaisir à certains ayants-droits. Or, cette solution ne satisfait personne : ni les internautes, ni les ayants-droits, ni les salariés. ■

NE MANQUEZ PAS LINUX PRATIQUE N°86 !

FOCUS :

AUTOMATISEZ L'ADMINISTRATION DE VOS SERVEURS...

...EN TOUTE SIMPLICITÉ AVEC RUNDECK !

DISPONIBLE CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR : www.ed-diamond.com

UNE HISTOIRE DE L'INFORMATIQUE #3 - RECHERCHES ET DÉVELOPPEMENTS

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

À l'orée de la Seconde Guerre mondiale, la pression s'accentue dans le monde industriel pour détenir des moyens d'automatiser les calculs. La Seconde Guerre mondiale va rendre ceux-ci vitaux et forcer les militaires à prendre le taureau par les cornes. L'informatique moderne est sur le point de naître.

Les progrès de la science ne sont pas linéaires. Ceci est aussi vrai pour l'informatique où beaucoup de chercheurs se sont égarés dans une voie de garage qui leur paraissait pourtant prometteuse : le calcul analogique.

1 De nombreuses pistes de recherche

1.1. Intégrateur différentiel

Dès le début des années 1930, de nombreux universitaires avaient eu l'occasion de jouer avec l'analyseur différentiel. Ce calculateur analogique a été inventé par le frère de Lord Kevin, James Thomson, et vraiment réalisé de façon concrète par Venevar Bush. Il est basé sur la rotation de volants, que vous pouvez peut-être voir, chez vous, sur votre vieux compteur EDF à disque.

Le calculateur différentiel a une précision maximale de 6 bits (2% de marge d'erreur en moyenne), ce qui est très vite limitant. De plus, il oblige à exprimer toutes les opérations utiles sous forme d'équations différentielles. En effet, le calculateur différentiel est par construction capable de résoudre des opérations d'intégration et certaines équations différentielles. Ainsi, la multiplication se calcule par :

$$u \cdot v = \int u \, dv + \int v \, du$$

De plus, le calcul sur un tel calculateur est basé sur la vitesse de rotation des roues, ce qui implique de très grandes difficultés à améliorer la vitesse d'exécution des opérations. Ne parlons même pas de la programmabilité de ces machines, qui est inexistante.

La problématique de choisir la piste analogique et digitale avait des partisans de l'un ou l'autre camp. Si la piste digitale utilisait des machines à relais électromécaniques, les vitesses d'exécutions étant au final à peu près similaires. Par contre, un interrupteur électronique (un tube à vide) pouvait à l'époque fonctionner à 5µs, ce qui était 2000 fois plus rapide qu'un relais mécanique. Il était donc possible, mais non facile, d'envisager un calculateur extrêmement rapide. Néanmoins, les ingénieurs de l'époque étaient extrêmement réservés sur la fiabilité des tubes. Pour la plupart d'entre eux, réunir plus d'une centaine de tubes impliquerait que l'un d'entre eux serait forcément toujours en panne, à tour de rôle, rendant la machine

inutilisable - ce qui, en regard de la fiabilité désastreuse des tubes à vide de l'époque était une intuition justifiée.

Pour ces raisons, le concept de machine analogique pour le calcul va longtemps influencer la communauté scientifique pour laquelle la piste digitale ne restera qu'une des deux alternatives jusqu'à la fin des années 1950. En France, il faudra attendre les années 1960 pour que l'on comprenne que cette approche des calculateurs est une voie de garage. C'est l'une des principales raisons du retard initial de notre pays en recherche informatique.

1.2. Le MARK I

Créé par Howard Aiken entre 1937 et 1944, et financé par IBM, l'*Automatic Sequence Controlled Calculator* ou Harvard Mark I est le pendant américain, version industrielle, mais plus complexe, de la machine de Conrad Zuse. Elle était capable d'exécuter des instructions arithmétiques codées sur une carte perforée.

Howard Aiken a conceptualisé la machine à peu près à la même époque que Zuse, preuve que les idées de Babbage avaient fait leur chemin et mûri avec la disponibilité des relais électriques. C'est d'ailleurs la lecture d'une présentation de la machine analytique par le fils de Babbage, 50 ans plus tôt, qui lancera Aiken sur la voie.

À la différence de la machine de Zuse et Babbage, le Mark I est décimal, et l'architecture plus complexe : elle possède pas moins de 72 accumulateurs différents dont la plupart sont capables de faire des additions, certains des soustractions, des incrémentations, des valeurs absolues et... un seul une multiplication. Chaque instruction indiquait les deux registres sources, le registre destination et l'opération à effectuer, mais celle-ci dépendait des registres impliqués. Ainsi, il n'y avait pas de séparation claire entre mémoire et exécution des calculs.

Le premier programmeur de cette machine fut John Von Neumann, le 29 mars 1944, qui voulait tester mathématiquement le détonateur nucléaire basé sur une implosion (petite explosion sur les parois provoquant une implosion au centre). Retenez ce nom et cette date, cela aura son importance dans la suite.

1.3. Attanasoff Berry Computer

L'Attanasoff Berry Computer (ABC), du nom de ses deux auteurs a été conçu vers 1937 au Iowa State College. L'ABC est une machine semi-mécanique, munie de quelques tubes à vide et basée sur un principe - évident aujourd'hui - de séparation entre la mémoire et le calculateur, qui est comme on le verra, la base de l'architecture Von Neumann. Attanasoff eut l'idée de construire son calculateur en utilisant des tables d'additions et de soustractions - une sorte de mémoire morte. Du fait de la taille rapidement exponentielle des tables en base 10, il choisit l'approche binaire.

Bien qu'assez similaire au Zuse I sur de nombreux points, il était en retard sur celui-ci : le Zuse possédait déjà la notion de mémoire programme, avec une suite d'instructions à exécuter et était totalement automatique. L'ABC devait être utilisé par un opérateur qui manipulait les interrupteurs un à un pour conduire l'opération. En termes modernes, l'unité d'exécution du CPU qui coordonne les différents éléments logiques de la machine n'était pas automatique. L'opérateur devait conduire lui-même les opérations de base du CPU : *fetch* (aller chercher l'instruction), *decode* (démultiplexer l'instruction) *execute*, *transfer* (transférer à la mémoire). De plus, construit avec un budget très limité (5000$ de l'époque, soit 10 000€ d'aujourd'hui), l'ABC était ralenti par les parties mécaniques de son architecture, quand bien même les unités électroniques pouvaient fonctionner à grande vitesse. En l'espèce, son horloge de base était de 60hz (fréquence du courant alternatif aux États-Unis d'Amérique).

L'ABC n'a jamais été réellement terminé : Attanasoff, devant les problèmes de fiabilité et ses travaux plus utiles à l'effort de guerre à ses yeux, abandonna ceux-ci tout en attendant patiemment que son employeur se décide à breveter son invention, ce qui ne se fera jamais. En attendant la procédure de brevetage, il restera très circonspect quant à léguer ses recherches à John Mauckly. Néanmoins, ce dernier, qui travailla par la suite sur l'ENIAC, put en retirer de nombreux concepts stimulants.

Bien que cette machine ne donna pas grand-chose, l'ABC fut une avancée conceptuelle intéressante, en ce qu'elle institue la séparation entre les étages d'exécution d'un CPU moderne.

2 L'ENIAC : une fausse piste... très fructueuse

Le 7 décembre 1941, l'armée impériale japonaise attaque par surprise la base américaine de Pearl Harbor au milieu du Pacifique. Roosevelt ne voit plus aucun obstacle, au sein de l'opinion publique américaine alors franchement isolationniste, à déclarer la guerre au Japon et partant, son allié germanique. L'administration américaine organise alors l'orientation de la très puissante industrie américaine à

atteindre une production d'armement maximale. La guerre contre le Japon va très vite se concrétiser avec la bataille aéronavale de Midway, du 5 au 7 juin 1942. Et c'est là que les ordinateurs furent nécessaires... Car qui dit bataille navale, dit artillerie, lancer de projectile, et donc trajectoire...

Pour aider l'artilleur en pleine bataille, on lui met à disposition des tables de tir, lui indiquant l'angle horizontal de tir (le gisement) et l'élévation (angle vertical de tir) de son canon. Ces tables de tirs se présentent sous la forme de livres, dotés d'index multicritères. Ces deux paramètres permettent de calculer la trajectoire de tir de l'obus en fonction de son poids et de sa poussée initiale, qui est définie à l'avance. On peut ainsi atteindre n'importe quel point dans un disque allant jusqu'aux limites de portée de tir.

À la suite de la Première Guerre mondiale, où l'imprécision des lancers d'obus a pu être testée en long, en large, et en travers (des soldats, entre autres), les modèles mathématiques de calculs de trajectoire ont été largement améliorés afin de prendre en compte les paramètres environnementaux. Car notre environnement terrestre n'a rien à voir avec l'Espace où règne le vide : des paramètres supplémentaires comme la pression en fonction de l'altitude, la température, le sens du vent, la vitesse de celui-ci, le taux d'humidité ont une importance capitale dans l'évaluation d'un tir sur plusieurs kilomètres.

Lors de la bataille entre les gigantesques Bismark et le King George V le 27 mai 1941, les cuirassés se canonnaient jusqu'à 12km de distance ! Un croiseur étant large de quelques dizaines de mètres, la précision recherchée était extrêmement difficile à obtenir. Le Bismark fut, à cause de cela, très vite à court d'obus, la plupart étant tombés dans l'eau, et se saborda pour éviter sa capture. Les tables de calculs revêtent donc une importance absolument cruciale pour l'armée - un obus tombant dans l'eau est un projectile perdu, et en quatre heures de bataille on se retrouve vite à court de munitions.

Bien évidemment, ces modèles plus réalistes compliquaient fortement les calculs : une seule trajectoire avec des paramètres fixés nécessitait environ 750 multiplications pour une précision de 1/10000. Un humain muni d'une calculatrice mécanique alors largement répandue mettait 12 heures à calculer une seule trajectoire. En comparaison, le calculateur mécanique Mark I réalisait la même tâche en 2 heures (facteur 6).

Un livret de table de tirs comportait entre 2000 et 4000 trajectoires uniques... Et c'était des dizaines de tables de tir de ce type qu'il fallait fournir, pour différents canons et différentes tailles de pièces d'artillerie ! Les responsables de l'armée, et en particulier du Laboratoire de Balistique qui était chargé de calculer les tables de tir, se rendent compte que même une armée de calculateurs humains (2000 en 1940) munis d'outils mécaniques va mettre des années à venir à bout de ce travail titanesque, et donc trop tard. De même, une centaine de très onéreux et massifs Mark I suffirait à peine à la tâche. Devant le goulet d'étranglement qu'ils voyaient devant eux et l'échec relatif de modèles nécessitant moins de calculs, les responsables du bureau de la Balistique réagirent comme leurs ancêtres du bureau des statistiques en 1890 : il leur fallait une machine capable d'abattre tous les calculs.

2.1. Lancement du projet

Le Ballistic Research Laboratory a été créé en 1935 à Aberdeen par Hermann Zorning. Ce laboratoire est chargé de toutes les recherches concernant l'artillerie. Cela peut autant concerner des modèles de contrôle statistiques de la qualité des munitions que les fameuses tables de tir. C'est le Lieutenant Paul Gillon, Executive Officer du BRL, qui va impulser l'industrialisation des calculs des tables de tir. Gillon est très favorable à l'approche digitale, malgré le contexte scientifique nettement favorable à l'approche analogique. Dès la fin des années 30, le BRL a été en discussion régulière avec IBM afin d'automatiser une bonne partie des processus de calcul, et ce, par l'utilisation de machines à carte perforée, tabulatrices, multiplieurs, etc.

À cette époque, les machines travaillant sur des cartes perforées étaient capables de totaliser les nombres d'une pile de cartes perforées, d'autres machines étaient capables de multiplier des colonnes, etc. Ce genre d'outil convenait pour de la comptabilité, mais devenait peu utilisable pour des calculs mathématiques complexes fortement dépendants entre eux. Parallèlement, un programme de formation mixte (rare à l'époque), l'ESMWT, est mis sur pied pour former des spécialistes du calcul balistique et de l'utilisation de machines mécaniques. Bien évidemment, les femmes se partageront les postes plus subalternes, mais auront l'honneur d'être les premières « programmeurs ».

À l'entrée en guerre des États-Unis en 1941, le BRL se voit doté de fonds encore plus conséquents (bien qu'avec la montée en tension des années précédentes, ceux-ci avaient été fortement augmentés) qui vont lui permettre de constituer une équipe scientifique de premier plan. Herman Goldstine, jeune docteur et professeur de mathématiques à l'université de Chicago, spécialiste de la balistique, est désigné pour prendre la direction des opérations à la Moore School (Philadelphie) le 30 septembre 1942, à l'occasion de sa mobilisation

DÉCOUVREZ NOS NOUVELLES OFFRES D'ABONNEMENTS !

PRO OU PARTICULIER = CONNECTEZ-VOUS SUR :

www.ed-diamond.com

LES COUPLAGES PAR SUPPORT :

VERSION
PAPIER
Retrouvez votre magazine favori en papier dans votre boîte à lettres !

VERSION
PDF
Envie de lire votre magazine sur votre tablette ou votre ordinateur ?

ACCÈS À LA
BASE DOCUMENTAIRE
Effectuez des recherches dans la majorité des articles parus, qui seront disponibles avec un décalage de 6 mois après leur parution en magazine.

Sélectionnez votre offre dans la grille au verso et renvoyez ce document complet à l'adresse ci-dessous !

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.

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.

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 :

.....

.....

VOICI TOUTES LES OFFRES COUPLÉES AVEC GNU/LINUX MAGAZINE !

POUR LE PARTICULIER ET LE PROFESSIONNEL ...

Prix TTC en Euros / France Métropolitaine

CHOISISSEZ VOTRE OFFRE !

Prix en Euros / France Métropolitaine

Offre	SUPPORT / ABONNEMENT	PAPIER Réf	PAPIER Tarif TTC	PAPIER + PDF (PDF 1 lecteur) Réf	PAPIER + PDF (PDF 1 lecteur) Tarif TTC	PAPIER + BASE DOCUMENTAIRE (1 connexion BD) Réf	PAPIER + BASE DOCUMENTAIRE (1 connexion BD) Tarif TTC	PAPIER + PDF + BASE DOCUMENTAIRE (PDF 1 lecteur + 1 connexion BD) Réf	PAPIER + PDF + BASE DOCUMENTAIRE (PDF 1 lecteur + 1 connexion BD) Tarif TTC
LM	11n° GLMF	LM1	65,-	LM12	95,-	LM13	144,-	LM123	179,-
LM+	11n° GLMF + 6n° HS	LM+1	118,-	LM+12	178,-	LM+13	197,-	LM+123	262,-
LES COUPLAGES « LINUX »									
A	11n° GLMF + 6n° LP	A1	95,-	A12	142,-	A13	218,-	A123	270,-
B	11n° GLMF + 6n° MISC	B1	100,-	B12	147,-	B13	228,-	B123	280,-
C	11n° GLMF + 6n° LP + 6n° MISC	C1	135,-	C12	199,-	C13	300,-	C123	376,-
B+	11n° GLMF + 6n° HS + 6n° MISC + 2n° HS	B+1	172,-	A+12	252,-	A+13	307,-	A+123	385,-
A+	11n° GLMF + 6n° HS + 6n° LP + 3n° HS	B+1	182,-	B+12	267,-	B+13	305,-	B+123	395,-
C+	11n° GLMF + 6n° HS + 6n° LP + 3n° HS + 6n° MISC + 2n° HS	C+1	236,-	C+12	346,-	C+13	408,-	C+123	523,-
LES COUPLAGES « EMBARQUÉ »									
F	11n° GLMF + 6n° HK* + 4n° OS	F1	125,-	F12	184,-	F13	229,-*	F123	293,-*
F+	11n° GLMF + 6n° HS + 6n° HK* + 4n° OS	F+1	183,-	F+12	267,-	F+13	287,-*	F+123	376,-*
LES COUPLAGES « GÉNÉRAUX »									
H	11n° GLMF + 6n° HK* + 6n° LP + 6n° MISC + 4n° OS	H1	200,-	H12	293,-	H13	397,-*	H123	495,-*
H+	11n° GLMF + 6n° HS + 6n° HK* + 6n° LP + 3n° HS + 6n° MISC + 2n° HS + 4n° OS	H+1	301,-	H+12	440,-	H+13	498,-*	H+123	642,-*

Les abréviations des offres sont les suivantes : LM = GNU/Linux Magazine France | HS = Hors-Série | LP = Linux Pratique | OS = Open Silicium | HC = Hackable

* HK : Attention : La base Documentaire de Hackable n'est pas incluse dans l'offre.

N'hésitez pas à consulter les détails des offres ci-dessus sur : www.ed-diamond.com !

dans l'armée. La Moore School est une école d'ingénieurs associée au BRL qui lui sous-traite de nombreux travaux, en particulier la fabrication des calculateurs différentiels.

Fin 1942, John Mauchly, professeur à l'Ursinus College, une école d'ingénieurs qui travaille depuis quelques années sur les concepts des calculateurs digitaux, est appelé à participer au projet. Mauchly, comme on l'a vu, s'est beaucoup intéressé à l'ABC, partiellement électronique. À cause des problèmes de brevets, Mauchly dut aller rencontrer Atanasoff en juin 1941 afin d'en savoir un peu plus sur l'ABC. C'est lors de cette visite que Mauchly va être définitivement convaincu que les calculs doivent être réalisés à partir d'un circuit électronique et non mécanique. Après moult réflexions, Mauchly s'appuyant entre autres sur l'ABC a écrit un mémoire synthétisant ses idées sur la conception d'un calculateur électronique et digital. Il est alors clairement devenu un partisan de l'approche tout électronique, pensant que l'approche mécanique est trop lente et trop peu fiable. Il va vite s'associer à Eckert, un brillant étudiant en électronique de 23 ans, devenu assez vite un des meilleurs spécialistes de l'électronique digitale.

En mars 1943, Goldstine propose à Gillon de collaborer plus étroitement avec la Moore School et Mauchly (fraîchement embauché à la Moore School) pour développer des recherches sur le calcul électronique. Brainerd prépare un rapport envoyé à Gillon le 2 avril 1943 décrivant le fonctionnement et les capacités de l'ENIAC (*Electronic Numerical Integrator Analyser and Computer*), ainsi qu'une estimation des coûts à 150 000$. Il y est prévu les principaux éléments architecturaux et le projet est vite accepté par le comité directeur du BRL.

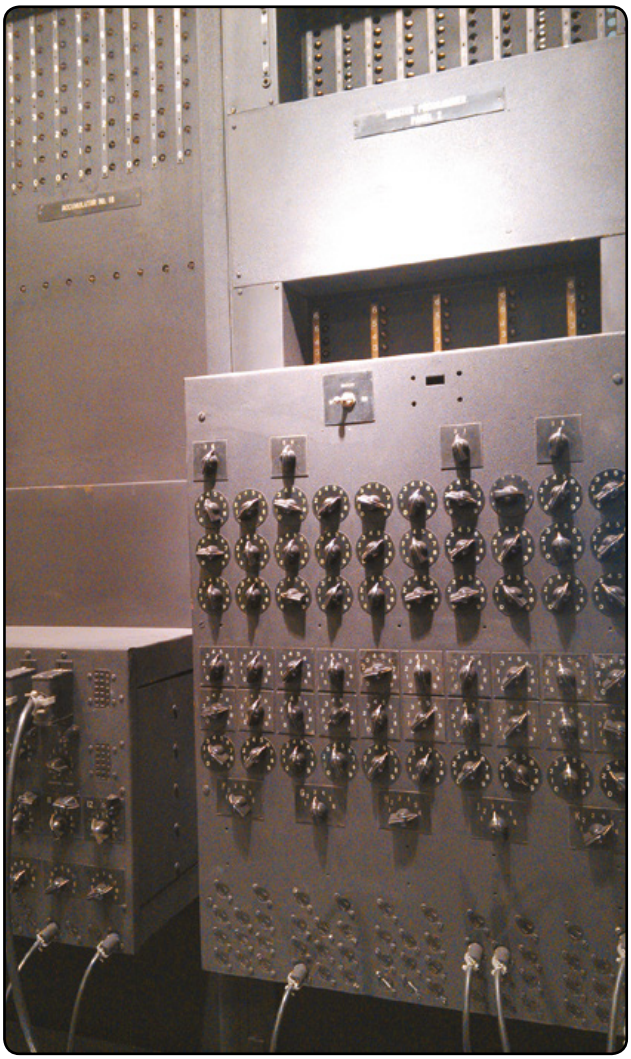


Fig. 1 : L'ENIAC à l'Université de Pennsylvanie (Philadelphie).

2.2. Conception de l'ENIAC

Lorsqu'il est décidé (et budgété) de construire l'ENIAC, en 1943, leurs concepteurs n'ont aucun exemple électronique, et les quelques exemples mécaniques sont au mieux partiellement numériques. La quasi-totalité des machines existantes, et fonctionnelles, étaient analogiques. Pire, on n'a aucune expérience de l'utilisation de l'électronique pour des calculs binaires... Tout était à inventer : flip-flop, additionneurs, portes logiques de base, etc.

L'ENIAC reprend de l'ABC :

- L'usage des tubes en mode binaire dans la conception électronique de la machine, mais pas logique, comme on le verra ;
- L'utilisation des tubes comme interrupteur pour le contrôle des opérations de la machine ;
- La notion de contrôle (partiellement humain) séquentiel des opérations ;
- Le fait de calculer des équations différentielles par voie digitale par intégration numérique.

2.3. Architecture de l'ENIAC

Pour un informaticien moderne, pour lequel la notion de processeur exécutant des instructions se trouvant en mémoire tombe sous le sens, l'ENIAC a une conception pour le moins… tordue. En plus d'être un monstre (17 468 tubes !), l'ENIAC est difficile à programmer et n'a quasiment pas de mémoire puisqu'il ne possède réellement

que 20 mots de mémoire vive, correspondant à ses 20 accumulateurs. Cette machine, dont les restes sont exposés à l'Université de Pennsylvanie (voir figure 1), est conçue pour réaliser des calculs ayant une forte dépendance entre eux ou alors de réaliser des calculs par « paquets », comme les tabulatrices. Encore une fois, le bain culturel a encore fortement influencé les architectes : l'ENIAC est une machine décimale. Pour stocker ces nombres décimaux, une série de 10 flipflop permet de stocker un « bit » décimal. Les nombres sont stockés en complément à 10, avec un bit (un vrai, pour le coup) de signe.

L'ENIAC est construit autour de 20 accumulateurs ou unités de soustraction/addition. Chacun des 20 accumulateurs peut stocker un nombre décimal positif ou négatif (avec un maximum de 10 chiffres). Un nombre arrivant dans un accumulateur est toujours ajouté au nombre présent - d'où le nom d'accumulateur. La soustraction est réalisée par transmission d'un nombre « négativé » à un accumulateur. Cela dit, l'accumulateur de l'ENIAC est un peu plus qu'un additionneur : il est capable de répéter n fois une addition, en se renvoyant à lui-même un signal. Les nombres sont transmis par des trains de pulsations : un par « bit » en décimale, e nombre de pulsations représentant le nombre entre 0 et 9.

L'ENIAC est aussi doté d'un multiplicateur simple, ainsi que d'un diviseur et d'un module de calcul de racine carrée. Les constantes étaient stockées dans des systèmes à relais, mis à jour avec des cartes perforées. Cette unité est appelée le « Function Table ». Une imprimante peut être utilisée en tant que dispositif de sortie pour les résultats numériques.

L'horloge de l'ENIAC est capable de délivrer - pour l'ensemble de la machine - 10 types de pulsations. Les autres unités peuvent utiliser ces impulsions comme référence ou comme entrée, de sorte que l'ensemble de la machine reste synchronisé. L'*initiating unit*, d'autre part, permet à l'opérateur de démarrer ou arrêter la machine. Le *Master Programmer*, un générateur à impulsion, peut être utilisé pour implémenter des boucles, ou pour commencer deux calculs différents en parallèle dans deux portions de la machine.

Il existe deux bus dans l'ENIAC : le bus de commande est utilisé pour envoyer des impulsions d'entrée pour démarrer les unités et le bus de données est utilisé pour transférer les nombres d'une unité à l'autre. Une unité qui termine un calcul envoie une impulsion de sortie, à travers le bus de commande, à l'unité supérieure, afin de commencer le calcul suivant. Mais comment l'ENIAC était-il programmé ? Essentiellement par câblage… Concevoir un programme pour l'ENIAC consistait à dessiner un circuit de transmission des données entre accumulateurs avec lequel les nombres vont circuler pour être additionnés, soustraits, multipliés, divisés, renvoyés un certain nombre de fois entre les différentes unités de calcul et finir dans l'imprimante.

Conclusion

Le Zuse 1, dont l'histoire montrera qu'il avait tout de suite vu juste, était inconnu à l'époque par les équipes du Laboratoire de Balistique, et une version à tube d'un ordinateur de Conrad Zuse eut été bien plus efficace et moins onéreuse que l'ENIAC. Il eut sans doute été prêt en 1 an au lieu de 3.

S'il est médiatiquement le premier ordinateur « électronique », l'ENIAC est avant tout la simulation d'une machine décimale à roue dentée, une sorte de Pascaline électronique. Architecturalement, son impact est resté limité à démontrer ce qu'il ne fallait pas faire. Néanmoins, il a prouvé la viabilité de l'approche digitale et électronique : seuls des outils électroniques peuvent produire des milliers d'étapes par seconde, automatiser ceux-ci, et donc ouvrir la voie à une automatisation complète des calculs. Autre enseignement important, malgré le taux de pannes non négligeable des machines à tubes, la masse de calculs abattus par seconde est tellement énorme que le taux de fiabilité est de très loin meilleur que leurs homologues mécaniques. Mais la maîtrise technologique de cet outil pose de nombreux problèmes : les lampes radio, conçues pour l'amplification analogique de signaux, ne sont pas très adaptées à la commutation à haute fréquence. De plus, elles sont très énergivores, très peu fiables et dangereuses, fonctionnant à quelques centaines de volts avec un ampérage conséquent.

Néanmoins, à une époque où la culture des électroniciens était quasi exclusivement analogique, l'ENIAC a permis de diffuser dans la culture technicienne les briques électroniques nécessaires à l'émergence de l'électronique digitale. Ce savoir sera fondamental par la suite : portes binaires, flip-flop, etc. L'ENIAC a été « mal conçu », car basé sur une manière déjà industrialisée de calculer en fonction des limites existantes à son époque. Les concepteurs ont surtout cherché à automatiser des tâches humaines telles qu'elles étaient distribuées, alors qu'il fallait reprendre le problème à sa base : comment faire un calcul à la main, et comment l'automatiser, façon recette. C'est ce travail que va réaliser Von Neumann, comme nous le verrons dans le prochain article. ■

PROFESSIONNELS !

DÉCOUVREZ NOS NOUVELLES OFFRES D'ABONNEMENTS ...

1 ABONNEMENT PDF (1-5 LECTEUR(S)) SOUSCRIT = 1 ABONNEMENT PAPIER OFFERT !

OFFRE VALABLE PAR PALIER DE LECTEURS

PDF COLLECTIFS		PROFESSIONNELS					
		1 - 5 lecteurs		6 - 10 lecteurs		11 - 25 lecteurs	
OFFRE	ABONNEMENT	Réf	Tarif TTC	Réf	Tarif TTC	Réf	Tarif TTC
PROLM	11n° GLMF	☐ PRO LM2	260,-	☐ PRO LM2	520,-	☐ PRO LM2	1040,-
PROLM+	11n° GLMF + 6n° HS	☐ PRO LM+2	472,-	☐ LM+2	944,-	☐ PRO LM+2	1888,-

Prix TTC en Euros / France Métropolitaine

PROFESSIONNELS : N'HÉSITEZ PAS À NOUS CONTACTER POUR UN DEVIS PERSONNALISÉ PAR E-MAIL : abopro@ed-diamond.com OU PAR TÉLÉPHONE : 03 67 10 00 20

1 ACCÈS COLLECTIF (1-5 CONNEXION(S)) SOUSCRIT = 1 ABONNEMENT PAPIER OFFERT !

OFFRE VALABLE PAR PALIER DE CONNEXIONS

ACCÈS COLLECTIFS BASE DOCU		PROFESSIONNELS					
		1 - 5 connexion(s)		6 - 10 connexions		11 - 25 connexions	
OFFRE	RÉABONNEMENT	Réf	Tarif TTC	Réf	Tarif TTC	Réf	Tarif TTC
PROLM+	GLMF + HS	☐ PRO LM+3	267,-	☐ PRO LM+3	534,-	☐ PRO LM+3	1068,-
PROA+	GLMF + HS + LP + HS	☐ PRO A+3	297,-	☐ PRO A+3	594,-	☐ PRO A+3	1188,-
PROH+	GLMF + HS + LP + HS + MISC + HS + OS	☐ PRO H+3	447,-	☐ PRO H+3	894,-	☐ PRO H+3	1788,-

Prix TTC en Euros / France Métropolitaine

Les abréviations des offres sont les suivantes : LM = GNU/Linux Magazine France | HS = Hors-Série | LP = Linux Pratique | OS = Open Silicium

...EN VOUS CONNECTANT À L'ESPACE DÉDIÉ AUX PROFESSIONNELS SUR :

www.ed-diamond.com

LES FAILLES DE CHIFFREMENT

par ***Tristan Colombo***

Dans ce numéro où nous parlons de cryptographie, nous ne pouvions pas ne pas revenir sur les deux plus grandes failles de l'année 2014.

De nombreuses failles ont été découvertes cette année et de nombreuses le seront encore sans doute l'année prochaine. L'important n'est pas réellement que ces failles existent, mais surtout qu'elles soient découvertes et corrigées le plus rapidement possible. Dernièrement, les failles **Poodle [1]** et **SRTP [2]** ont été rendues publiques, mais j'ai choisi de m'intéresser aux deux failles les plus importantes de cette année dont vous avez forcément entendu parler : les failles **Heartbleed** et **Goto fail**.

1 Heartbleed

Tout comme pour Poodle et SRTP, Heartbleed est une faille détectée dans OpenSSL et annoncée le 7 avril 2014. Cette faille du « saignement du cœur » ou « cœur saignant » a même eu droit à son petit logo (voir figure 1). Mais que provoque exactement cette faille ?

OpenSSL (SSL pour *Secure Socket Layer*) est une bibliothèque de cryptographie qui permet de gérer des échanges cryptés client/serveur.

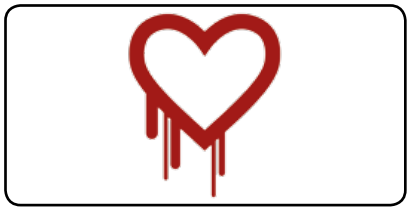


Fig. 1: Logo de la faille Heartbleed

La faille a été introduite lors de la mise à jour d'une extension nommée **Heartbeat** (battement de cœur), d'où le jeu de mots avec Heartbleed. Le problème provient simplement de l'oubli de la vérification de la longueur d'un message : ce sont bien souvent les erreurs les plus triviales qui provoquent les plus grands dommages. Cette faille ne provient pas d'une erreur fondamentale dans la définition d'un protocole, mais seulement d'une erreur d'implémentation, d'une étourderie.

Tout part donc de l'extension Heartbeat qui permet l'utilisation d'une connexion persistante (*keep-alive*) : le client fait une demande `heartbeat_request` et le serveur renvoie une réponse `heartbeat_response`. Un message est construit en utilisant une structure spécifiant le type de message (requête ou réponse), le message utile (*payload*), sa taille et des données aléatoires sur un minimum de 16 octets **[3]** :

```
struct {
   HeartbeatMessageType type;
  uint16 payload_length;
   opaque payload[HeartbeatMessage.payload_length];
   opaque padding[padding_length];
} HeartbeatMessage;
```

Lors de la réception de la requête, le serveur réserve une zone mémoire permettant de stocker cette structure : un octet pour le type, deux octets pour la taille du message (*payload*), `n` octets correspondant à la taille précédente et `m` octets correspondant à la taille des données aléatoires (*padding*). Lors de la réponse, une structure de la même taille est créée dans laquelle le type est assigné à `heartbeat_response` et le message (*payload*) est recopié. Si la requête spécifie une taille de message (jusqu'à 64 Ki), mais pas de message, c'est le contenu d'une zone aléatoire de la mémoire qui sera renvoyé : la faille Heartbleed !

Cette faille a-t-elle été introduite pour le compte de la NSA, a-t-elle été exploitée par cette dernière **[4]** ? On ne le saura sans doute jamais, mais dans le climat de doutes créé par les révélations d'Edward Snowden, une petite polémique a vu le jour au moment de la révélation de cette faille. Heartbleed a beaucoup fait parler d'elle, car elle a rendu vulnérables de nombreuses machines pendant une durée relativement longue sur des communications cryptées par chiffrement SSL/TLS (confidentialité des navigateurs Web, gestionnaires de mail, messageries instantanées ou encore VPN). De plus, l'exploitation de cette faille ne laisse aucune trace dans les logs.

2 Goto fail

L'implémentation des protocoles SSL/TLS sur iOS contenait une faille pour les versions antérieures à iOS 6.1.6 et iOS 7.0.6. Cette faille

permettant de visualiser et éventuellement de modifier les données échangées lors d'une connexion SSL a été annoncée en février 2014.

On peut voir sur le code source **[5]** où se situe l'erreur et d'où vient son nom :

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa,
SSLBuffer signedParams,
                                 uint8_t *signature, UInt16
signatureLen)
{
    OSStatus        err;
    ...

    if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;

        err = sslRawVerify(ctx,
                           ctx->peerPubKey,
                           dataToSign,              /* plaintext */
                           dataToSignLen,      /* plaintext length */
                           signature,
                           signatureLen);
        if(err) {
                sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
                    "returned %d\n", (int)err);
                goto fail;
        }

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;

}
```

Non, il ne s'agit pas d'un extrait de la copie d'un mauvais étudiant de Licence : vous avez bien vu l'indentation du code et la grossière erreur de copier/coller. Si l'on réindente correctement le code, on voit bien que tout ce qui suit le **goto fail** en rouge ne sera jamais exécuté :

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa,
SSLBuffer signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
    OSStatus        err;
    ...

    if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
    goto fail; # Saute directement jusqu'à l'instruction
SSLFreeBuffer(...);
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        ...

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;

}
```

Si tous les tests précédents le **goto fail** en rouge renvoient la valeur **0**, alors **err** contiendra la valeur **0** et comme les tests suivant le **goto fail** (toujours en rouge) ne sont jamais exécutés, la vérification sera toujours positive et tous les certificats seront acceptés.

Comme quoi, ça peut paraître idiot et fastidieux d'indenter correctement son code, d'ajouter des accolades, mais ça permet d'éviter des erreurs...

Conclusion

Deux grosses failles dans des bibliothèques de chiffrement pour deux petites erreurs d'implémentation : on peut avoir la meilleure méthode de chiffrement qui soit, quand il y a une erreur d'implémentation, aussi minuscule soit-elle, tout tombe à plat... Dans certains domaines, les conséquences ne sont pas toujours très graves. Si vous passez à travers un mur dans un jeu, ça sera gênant pour le plaisir que vous éprouverez à jouer, mais ça n'aura pas de conséquence plus importante. En chiffrement, donner l'accès à des données confidentielles posera tout de suite beaucoup plus de problèmes ! ■

Références

[1] Faille OpenSSL Poodle : https://www.openssl.org/~bodo/ssl-poodle.pdf

[2] Faille OpenSSL SRTP : https://www.openssl.org/news/secadv_20141015.txt

[3] Définition de l'extension OpenSSL Heartbeat : https://tools.ietf.org/html/draft-ietf-tls-dtls-heartbeat-04

[4] Article du New-York Times sur l'exploitation de Heartbleed : http://www.nytimes.com/2014/04/13/us/politics/obama-lets-nsa-exploit-some-internet-flaws-officials-wsay.html

[5] Code source contenant la faille Goto fail : http://opensource.apple.com/source/Security/Security-55471/libsecurity_ssl/lib/sslKeyExchange.c

DÉMOCRATISER LA CRYPTOGRAPHIE

par ***Léo Ducas-Binda***
[Ancien élève de l'École Normale Supérieure, Chercheur en Cryptographie à University of California, San Diego
Co-auteur du prototype open source BLISS, signatures digitales fondées sur les Réseaux Euclidiens]

De la Scytale à Enigma, la Cryptologie est restée pendant longtemps un art secret, réservé aux militaires et diplomates. Elle devient une science dans les années soixante, et cette ouverture permet la naissance de nombreuses idées nouvelles. Mais beaucoup reste à faire pour la mise en pratique et la démocratisation de la cryptographie moderne.

Enigma est sans doute la méthode la plus emblématique de la cryptographie, pour son rôle historique durant la Seconde Guerre mondiale et c'est la cryptanalyse du chiffre Enigma qui motivera Turing et son équipe de mathématiciens à réaliser les premiers ordinateurs. C'est pour la cryptologie que naît l'informatique : d'abord la machine mécanique, puis la machine à transistor.

L'informatique trouvera très vite des applications commerciales, motivant des progrès théoriques comme l'invention des premiers langages de programmation (ex. FORTRAN, 1954), et des avancées pratiques avec la miniaturisation des machines et des circuits. Mais la cryptographie restera confidentielle jusqu'aux années 70, chaque état développant secrètement ses propres méthodes pour une utilisation interne.

La création du réseau ARPANET viendra bouleverser ce statu quo : il devient nécessaire de standardiser les méthodes de chiffrement à une échelle internationale. Un appel à candidatures est lancé, et attirera l'intérêt de scientifiques universitaires, mais aussi d'entreprises comme IBM. La Cryptographie s'ouvre et se transforme en discipline scientifique. Des théories longtemps considérées pures, comme l'arithmétique modulaire, se voient appliquées avec des retombées politiques et économiques majeures.

L'ouverture de la cryptographie permettra à de nouvelles idées d'apparaître, en particulier le domaine de la cryptographie dite asymétrique ou comment communiquer de façon chiffrée sans partage préalable d'une clef secrète. Naîtra aussi un combat pour la libre utilisation de la cryptographie, avec des logiciels open source comme PGP et OpenSSH. Mais il faudra attendre l'année 2004 pour voir l'usage privé de la cryptographie légalisé en France. L'actualité, avec des affaires comme celle de Lavabit **[1]**, montre tout de même que la confidentialité des communications face aux injonctions étatiques n'est pas encore un droit véritablement acquis.

1 Un premier Standard, le DES

Jusqu'à la fin des années 60, les techniques cryptographiques restent secrètes, développées indépendamment dans chaque nation pour les applications diplomatiques et militaires. Avec l'arrivée des réseaux informatisés dans les entreprises, et bientôt ARPANET, le bureau états-unien des standards lance un appel à candidatures pour la standardisation en 1973. L'entreprise IBM proposera alors l'algorithme de chiffrement par bloc Lucifer, développé deux ans plus tôt, notamment par Horst Feistel. Quelques modifications furent apportées, et le DES, Data Encryption Standard, fut adopté en 1976. Cela marque le début de la recherche académique en cryptographie.

La contribution majeure de Feistel sera souvent réutilisée par la suite. L'idée derrière sa construction est qu'il est facile de concevoir une fonction qui a l'air aléatoire, mais beaucoup plus compliquée si l'on ajoute la contrainte que cette fonction doit être efficacement réversible. Il propose ainsi une transformation qui augmente une petite fonction quelconque en une plus grosse fonction réversible, à l'aide de quelques XOR (voir figure 1).

Il est aujourd'hui avéré que la NSA a influencé le NIST dans les modifications apportées a Lucifer pour définir DES **[2]**. L'une des modifications internes du schéma a augmenté la sécurité du DES, empêchant des techniques de cryptanalyse qui n'ont été découvertes par le milieu universitaire que plus d'une décennie plus tard. Mais l'autre modification a fait baisser la sécurité du schéma, et ce de façon peu subtile... la taille de clef a été réduite de 64 bits à 56 bits, réduisant ainsi le coup d'une attaque exhaustive par un facteur 2^8 = 256. Les esprits mesquins pourront essayer d'en déduire un encadrement de la puissance de calcul de la NSA en 1976.

Le DES sera remplacé par le triple-DES, puis en 2001 par l'AES qui reste le standard le plus répandu aujourd'hui. Des instructions AES sont implémentées en dur sur les processeurs modernes, permettant de chiffrer les données à des vitesses comparables au flux du bus RAM.

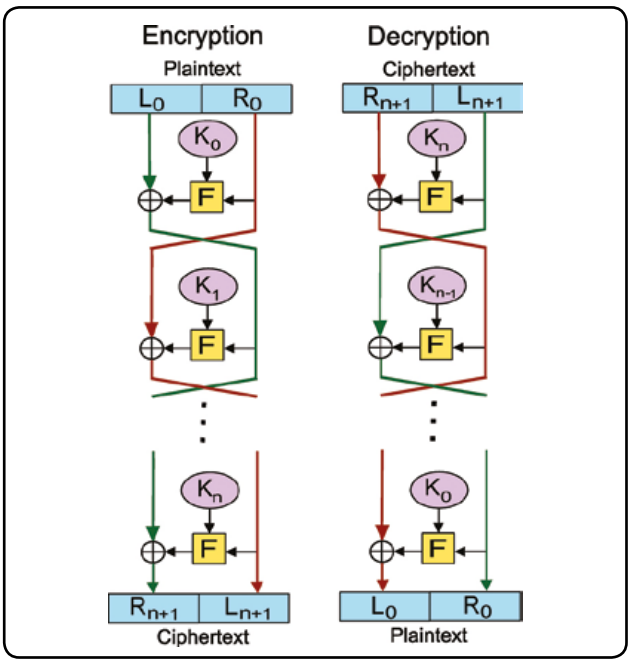


Fig. 1 : La transformation de Feistel : F est une petite fonction pseudo-aléatoire (quelconque), et la fonction totale est aussi pseudo-aléatoire, mais réversible. Image sous licence CC BY-SA 3.0, utilisateur Amirki, Wikimedia commons.

Quelques acteurs

Le GCHQ (Quartier général gouvernemental britannique des télécommunications)

Agence en charge de l'espionnage et du contre-espionnage. Célèbre pour la cryptanalyse d'Enigma durant la Seconde Guerre mondiale, ayant accueilli les fondateurs de l'informatique comme Alan Turing. Les récentes fuites concernant la NSA révèlent la collaboration du GHCQ avec la NSA sur des programmes de surveillance massive comme UPSTREAM **[3]**.

Le NIST (Bureau national états-unien des standards technologiques)

Agence Nationale américaine, du département du Commerce. Travaillant de concert avec l'industrie pour promouvoir le développement économique et technologique. Notamment en charge de la mise en place de standards pour les fonctions cryptographiques. Son indépendance a été mise en doute récemment avec l'affaire des *backdoors* dans le standard Dual EC-DRBG **[4]**.

La NSA (Agence nationale de sécurité états-unienne)

Agence en charge de l'espionnage et du contre-espionnage informatique. La fuite de documents classifiés en 2013 a soulevé une forte controverse sur la légalité et la légitimité de leur méthode de surveillance électronique **[3]**, incluant la collaboration (sous pression judiciaire) de nombreuses entreprises.

Lavabit

Société fondée en 2004, fournisseur de services de courriels chiffrés. En 2013, sous pression judiciaire la société décide de fermer son service plutôt que de livrer les clefs maîtresses de leurs serveurs **[1]**.

L'EFF (Electronic Frontier Foundation)

Fondation à but non lucratif ayant pour but de défendre la liberté d'expression sur internet. Outre son implication judiciaire et politique dans de nombreuses affaires, l'EFF se lance aussi dans des missions techniques, avec des projets comme le DES-cracker, ou de calcul distribué de nombres premiers.

L'IACR (Association internationale des chercheurs en cryptologie)

Elle organise les conférences majeures du domaine. Elle a en particulier négocié auprès des éditeurs de journaux que les cryptographes gardent presque tous les droits sur leurs publications. En mai 2013, elle s'exprime sur les affaires récentes avec la « résolution de Copenhague » **[5]**.

L'ANSSI (Agence nationale pour la sécurité des systèmes d'informations)

L'agence a pour mission la surveillance et la protection de réseaux sensibles, notamment des réseaux interministériels. Elle joue aussi un rôle de conseil, établit des standards et délivre des labels. Enfin, elle est censée informer régulièrement le public sur les menaces provenant des réseaux informatiques.

La CNIL (Commission nationale de l'informatique et des libertés)

Commission indépendante, munie d'un pouvoir de contrôle et de sanction, dont la mission est de veiller à la protection du droit à la vie privée et des libertés individuelles et publiques dans le cadre des réseaux et fichiers informatiques.

Alice et Bob

Protagonistes canoniques des aventures cryptographiques. Le choix des noms reflète simplement l'ordre alphabétique. Ainsi, si plus de parties sont en jeu, on invitera aussi Charly, Denise, Eve, Frank... Alice et Bob sont généralement les deux parties légitimes voulant communiquer. On réserve le prénom Eve pour un attaquant écoutant les communications, en raison de la consonance en anglais avec le terme Eavesdropper **[6]**.

2 La crypto asymétrique connue et utilisée

Toutes les techniques de chiffrement inventées jusqu'ici nécessitent que les parties voulant communiquer de façon confidentielle partagent au préalable une clef secrète ; cette limitation semblait naturelle à bon nombre de cryptologues. En 1970, James H. Ellis affirme dans une note interne au GHCQ qu'il est concevable d'effectuer des communications confidentielles sans secret partagé au préalable ; ouvrant la voie de la cryptographie dite à clef publique ou asymétrique.

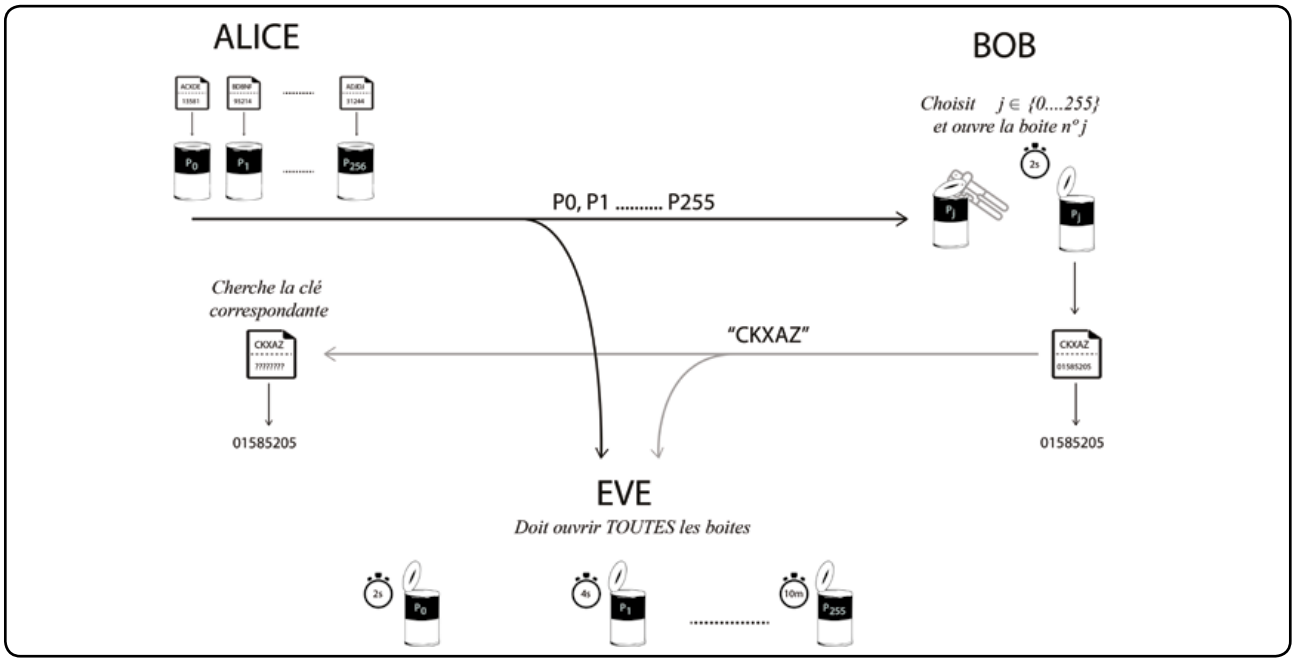
2.1. L'échange de clefs

Le problème principal de la cryptographie symétrique est logistique : avant de pouvoir communiquer secrètement les deux parties doivent connaître à l'avance un secret commun, la clef.

Plutôt que de s'attaquer directement au chiffrement, Merkle s'affaire à un problème plus simple. Deux parties communiquent à travers un calcul non confidentiel, mais veulent s'accorder sur un secret commun aléatoire. Théoriquement ce n'est pas possible, car toutes les informations connues par les deux parties sont aussi connues par l'attaquant. Mais Merkle veut faire en sorte que l'attaquant doive faire plus de calculs que les deux parties légitimes. On retrouvera bien plus tard son idée de puzzle dans les « preuves de travail » que constituent les *blockchains* de Bitcoin.

Prenons n'importe quel chiffrement, par exemple l'AES, mais rendons-le cassable en réduisant la taille des clefs à 32 bits (en complétant avec des 0) : cela prendrait environ deux secondes à casser sur une machine moderne. Alice a ainsi 256 petites clefs de 32 bits `k[0]` … `k[255]`, et chiffre des paires indices/grandes clefs aléatoires (`I[0]`, `K[0]`)... (`I[255]`, `K[255]`) qu'elle envoie à Bob, son correspondant : chaque message chiffré est un « puzzle » résoluble en y mettant un certain effort de calcul. Bob, choisi l'un de ces puzzles (disons le puzzle numéro `j`) et le résout, retrouvant ainsi `I[j]` et `K[j]`. Bob renvoie à Alice alors la valeur de `I[j]`, et ils peuvent utiliser la grande clef commune `K[j]`. Un attaquant voulant retrouver cette même clef devra essayer de résoudre les 256 puzzles (ce qui lui prendra environ 10 minutes) pour retrouver quel puzzle contient `I[j]` et en déduire `K[j]`: l'attaquant a dû fournir un plus gros effort que les participants légitimes. CQFD. Le protocole de Merkle est résumé en Figure 2.

Néanmoins, cette solution est loin d'être idéale, on voudrait un beaucoup plus gros écart entre l'effort de calcul des participants légitimes et l'attaquant. Il ne faudra attendre que deux ans avant que cette idée soit améliorée pour obtenir enfin un écart exponentiel entre le temps de calcul nécessaire aux parties légitimes et l'attaquant, avec le protocole d'échange de clefs de Bailey W. Diffie, Martin E. Hellman.



Crédit : Pratik Shah.

Fig. 2 : Le protocole de Merkle. Ici les puzzles sont représentés par des boîtes de conserve : c'est plus facile à ouvrir qu'un coffre fort, mais ça requiert un effort. Les indices I[0] … I[255] sont ici des chaînes de caractères alphabétiques, et les grandes clefs K[0]...K[255] des chaînes numériques.

L'idée novatrice est de s'appuyer sur une structure mathématique algébrique ; jusqu'alors toute forme de structure était évitée en cryptographie de peur qu'elle ne mène à des cryptanalyses. Cette structure algébrique permet en effet des attaques plus rapides que la recherche exhaustive, mais l'essentiel est que ces attaques restent exponentielles.

Soit **p** un nombre premier, et **g < p** un entier quelconque (ou presque), établi à l'avance, mais ne nécessitant pas d'être secret. Le protocole se déroule ainsi :

Alice		Bob
choisit **a** aléatoirement		choisit **b** aléatoirement
Calcul A = g^a mod p		**Calcul B = g^b mod p**
	→ **A** →	
	← **B** ←	
Calcul K = A^b mod p		**Calcul K' = B^a mod p**
K =	**g^(a*b) mod p**	**= K'**

Une version plus graphique de ce protocole est proposée en figure 3.

À la fin du protocole, les deux parties se sont mises d'accord sur une clef commune K = g^(ab). L'attaquant a seulement eu accès à A=g^a, et B=g^b, et peut certes calculer certaines combinaisons, telles que A * B = g^(a+b). Par contre, pour retrouver g^(ab) la seule méthode connue est de retrouver a à partir de A = g^a (ou symétriquement

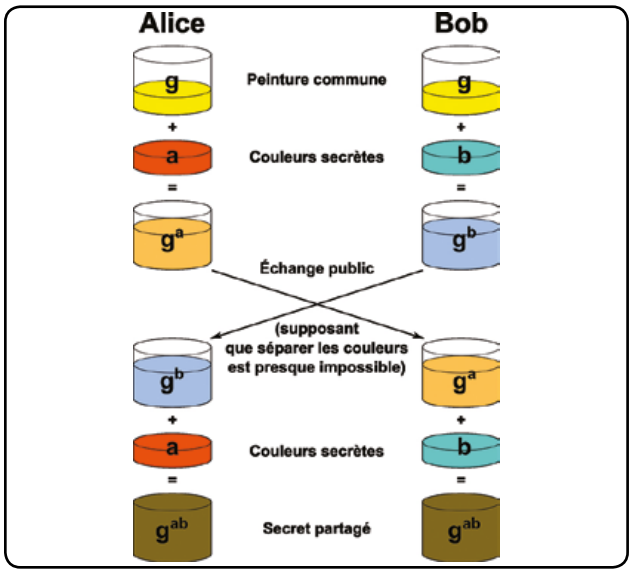


Fig. 3 : Le protocole Diffie-Hellman expliqué avec des mélanges de peinture. La métaphore veut qu'il soit facile de mélanger les peintures (ainsi on peut calculer A=g^a connaissant g et a), mais difficile de les séparer (retrouver a connaissant g et A). Image dérivée de Wikimedia, non éligible au copyright.

b à partir de B = g^b) c'est-à-dire résoudre le problème du logarithme discret ; les seuls algorithmes connus pour résoudre ce problème sont (presque) exponentiels.

Ce protocole est toujours d'actualité par exemple comme sous-protocole de SSL, mais au lieu d'utiliser l'exponentiation modulaire, on utilise aujourd'hui les courbes elliptiques (ECDHE) : les calculs restent similaires, mais la signification du symbole ^ est changée. Malheureusement, c'est un sujet difficile à aborder sans un lourd bagage mathématique.

2.2. Fonction à trappe, Chiffrement et Signature

En 1977, Ronald Rivest, Adi Shamir et Leonard Adleman proposent le premier algorithme de chiffrement à clef publique RSA. L'échange de clefs vu précédemment ne permet à deux parties que de s'accorder de façon confidentielle sur une clef commune aléatoire, qui servira de clef privée à la suite des échanges ; le chiffrement à clef publique RSA permet directement la transmission d'un message confidentiel, et ne nécessite pas d'interaction après la publication de la clef publique. Plus précisément, Alice souhaitant pouvoir recevoir des messages confidentiels, choisit deux grands nombres premiers `p` et `q`, qui constituent sa clef privée. De ces nombres elle déduit la clef publique `N = pq`, en calculant un simple produit. Ainsi retrouver la clef privée à partir de la clef publique est exactement le problème de la factorisation, pour lequel aucun algorithme efficace n'est connu, et ce malgré des décennies de recherche ; il est raisonnable de croire qu'aucun algorithme efficace n'existe !

Cet entier `N` sert de paramètre à une fonction dite *à trappe* : `f_N(x) = x^e mod N`, pour une valeur de `e` partagée par tous, disons `e=17`. Ainsi, connaissant `N`, il est possible à tous de calculer `y = f_N(x)` pour n'importe quelle valeur de `x`, mais inverser la fonction, c'est-à-dire retrouver `x` à partir `y = f_N(x)` ne semble être possible (au sens calculable efficacement) qu'en connaissance de la trappe : la factorisation de `N` en nombres premiers `p` et `q`. En effet, pour un entier `p`, il est possible (grâce au « petit Théorème de Fermat ») d'inverser la fonction `f_p(x) = x^e mod p` en calculant `d` l'inverse modulaire de `e`, c'est-à-dire un entier vérifiant `e*d = 1 mod (p-1)`, car on a alors `x^(e*d) = x mod p`. Ainsi, en inversant séparément `f_p` et `f_q`, on peut inverser complètement `f_N` en utilisant le « Théorème des restes Chinois ».

On peut aussi utiliser ces fonctions à trappe pour faire de la signature électronique, par exemple pour émettre des certificats X.509 dans le protocole TLS. On utilise la trappe pour trouver une signature `s` telle que `f_N(s) = m`, où `m` est le message.

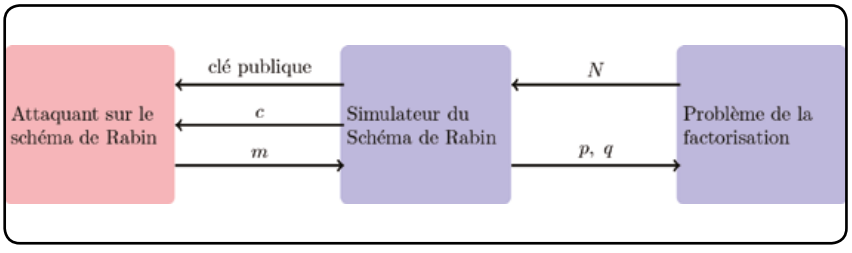


Fig. 4 : Le simulateur interagit avec l'attaquant, en prétendant être le vrai schéma de Rabin (mais en fait ne connaissant pas la clé privée). Il envoie donc une clé publique et un challenge c, un message chiffré (dont il ne connaît en fait pas le contenu). Le simulateur se sert de la réponse m de l'attaquant, pour résoudre le problème de la factorisation : retrouver p et q.

N'importe qui connaissant juste `N` mais pas la trappe peut vérifier que `F_N(s) = m`.

2.3. Les preuves de sécurité

Il reste cependant une question essentielle : la difficulté de factoriser `N` suffit-elle à garantir la confidentialité de ce schéma de chiffrement ? Ou, au contraire, existerait-il une façon de casser RSA sans pour autant factoriser `N` ? Cette question reste un problème ouvert : aucune méthode autre que la factorisation n'est connue à ce jour pour s'attaquer à ce chiffrement, mais les seules preuves établissant formellement un lien entre la confidentialité du schéma RSA et la difficulté de la factorisation ont été établies dans des modèles de calcul restreints.

Heureusement, deux ans après la publication de RSA, Michael O. Rabin propose un autre cryptosystème, cette fois-ci vraiment basé sur la factorisation au sens où il prouve mathématiquement que s'il existe un algorithme efficace compromettant la confidentialité de ce nouveau schéma, alors il existe un algorithme efficace de factorisation **[7]**. De telles preuves mathématiques sont appelées réductions, ou plus simplement preuves par l'absurde : on suppose l'existence d'un attaquant efficace, et l'on construit un algorithme (appelé un simulateur) qui fait appel à cet attaquant comme une sous-routine, par exemple ici pour résoudre le problème de la factorisation.

En d'autres termes, Rabin construit une fonction à trappe `F_N` (différente de celle de RSA), telle qu'il existe un algorithme (la réduction) qui ayant accès à une méthode pour inverser `F_N` (l'attaquant) permet de factoriser `N`. Et comme factoriser `N` semble être quasiment impossible, inverser `F_N` doit aussi être tout aussi impossible.

Une façon d'interpréter une preuve de sécurité est de considérer que l'algorithme de réduction « hack » l'attaquant :

l'attaquant ne fait que déchiffrer des messages, pourtant la réduction s'en sert pour un autre usage, résoudre des problèmes mathématiques supposés difficiles. Pour les lecteurs familiers de la notion de problème NP-complet, les méthodes de preuve de NP-complétude **[8]** sont tout à fait similaires.

3 Les nouvelles primitives

3.1. Le chiffrement à gestion de droit d'accès

Bien que ces outils cryptographiques suffisent à répondre aux problèmes principaux de sécurité sur un réseau non sécurisé, certains scénarios requièrent de nouvelles fonctionnalités. Une restriction est que le chiffrement ou la signature seule n'intègrent pas de façon native de structures hiérarchiques permettant une gestion fine des droits d'accès; si l'on veut chiffrer un message pour plusieurs personnes, il faut le chiffrer séparément pour chacune d'entre elles. Dans bien des cas, il serait beaucoup plus efficace de chiffrer le message une seule fois, en l'associant à une règle qui définit une règle d'accès. Avec un tel outil, on pourrait imaginer garantir les droits et les interdictions en lecture sur un système de fichiers, non pas par les verrous imposés par l'OS (parfois contournables), mais par une telle primitive cryptographique. Certes il est déjà possible de chiffrer ses fichiers, voire l'intégralité de son répertoire `/home/~`, mais cela n'offre pas la même finesse que la gestion des droits à la UNIX (`root > user > group > all`).

Le premier pas dans cette direction a lieu en 2001, lorsque Dan Boneh et Matthew K. Franklin proposèrent un schéma de chiffrement basé sur l'identité **[9]**. Leur construction s'appuie sur les courbes elliptiques et le couplage de Weil. Pour ces travaux, Boneh, Franklin et Joux ont reçu en 2013 le Prix Gödel.

Cette première version n'offre pas encore de structure fine d'accès, mais simplifie les échanges entre les autorités de certification et les parties par rapport à l'infrastructure à clefs publiques. Le point essentiel pour la suite est que les clefs ne sont plus choisies complètement indépendamment les unes des autres, mais toutes générées par une autorité unique, ce qui permet d'intégrer des structures de gestion de droits au sein même des clefs. La première évolution sera le chiffrement basé sur l'identité hiérarchique, permettant de chiffrer un message de telle façon qu'il soit déchiffrable par une entité précise, mais aussi par tous ses supérieurs hiérarchiques. Ont suivi de nombreuses variations offrant de nouvelles règles d'accès de plus en plus fines : grâce au couplage de Weil on peut exprimer essentiellement des règles d'accès sous forme de regexp (expression régulière). Très récemment, sont apparues de nouvelles solutions basées sur un autre objet mathématique, le réseau euclidien, et permettent cette fois-ci d'exprimer n'importe quelle règle qui serait calculable par un programme quelconque.

3.2. Le partage de clef et les multi-autorités

Le partage de clefs est une idée simple qui s'avère très utile dans de nombreux scenarii. Imaginons que l'on souhaite pouvoir déclassifier des documents (par exemple militaires) si deux autorités (disons judiciaires et législatives) s'accordent sur la déclassification de ces documents. Si la clef de chiffrement de ces documents est `k`, alors on peut partager cette clef, en choisissant un masque aléatoire `a`, et les deux parts du partage sont alors `k1 = k XOR a`, et `k2 = a`. Si les deux autorités s'accordent, elles peuvent retrouver `k = k1 XOR k2`. Par contre, en ne connaissant que `k1` (ou `k2`), on n'apprend aucune information sur la clef `k` elle-même, car le masque `a` est aléatoire. Pour une application plus courante, cela permet de pouvoir retrouver sa clef en cas de perte sans pour autant donner tout le pouvoir à une seule personne.

Il est possible de rendre le partage de clefs plus subtil en utilisant un peu d'arithmétique. Précisément, on peut partager la clef en `N` parts, et faire en sorte que n'importe quel sous-ensemble d'au moins `M` parts permette de reconstituer la clef partagée. On peut ainsi garantir que la déclassification soit possible si et seulement si une majorité des partis vote en faveur de la déclassification. La technique utilise l'interpolation de Lagrange pour les polynômes **[10]**.

L'idée du partage d'autorité peut s'appliquer aussi à la gestion fine de droits d'accès. En effet, l'un des problèmes des schémas présentés dans le précédent paragraphe est l'existence d'une autorité centrale capable de tout déchiffrer, et qui décide unilatéralement des droits de chaque utilisateur. Cela peut correspondre à une certaine hiérarchie interne en entreprise ou en agence gouvernementale ; mais une telle autorité centrale à l'échelle globale d'internet serait tout à fait inacceptable autant d'un point de vue de la sécurité (la fuite de la clef maîtresse compromet toutes les communications) que d'un point de vue politique. Par contre, si cette autorité était partagée entre différentes entités basées dans divers pays avec des antagonismes prononcés, cela pourrait être acceptable.

3.3. Le Chiffrement Homomorphe

Une autre direction de recherche ouvrant de nouvelles applications est le chiffrement dit homomorphe, c'est-à-dire un chiffrement qui préserve certaines structures arithmétiques entre les clairs et les chiffrés. L'exemple le plus simple est en fait le chiffrement RSA vu plus haut : le produit de deux chiffrés est un chiffré valide du produit des deux messages clairs. Dans beaucoup de contextes,

notamment en présence d'attaques actives, cette propriété est considérée comme une faiblesse, car la préservation de structure peut permettre à un attaquant d'extraire de l'information ; des contre-mesures brisant ces structures sont ajoutées.

Cependant, cette structure peut aussi être un atout considérable pour de nouveaux scénarios cryptographiques, car elle autorise un tiers à effectuer des opérations sur des données chiffrées sans qu'il soit pour autant capable de les déchiffrer. Un exemple d'application est le vote électronique **[11]** : chaque vote est chiffré puis publié, on additionne le contenu de tous les bulletins sans les déchiffrer ; calcul qui peut être publiquement vérifié pour éviter certaines fraudes. C'est seulement ce résultat final qui sera déchiffré, révélant uniquement le résultat de l'élection, mais pas les votes individuels. Et pour limiter les risques d'utilisation frauduleuse de la clef de déchiffrement, on peut inclure du partage de clef.

Outre RSA qui autorise la multiplication, d'autres schémas, comme celui de Pascal Paillier **[12]**, permettent l'addition. Il faudra attendre les travaux de Craig Gentry en 2009 **[13]** pour voir apparaître un schéma crédible de chiffrement qui autorise à la fois l'addition et la multiplication ; et le fait de pouvoir conjointement utiliser ces deux opérations autorise en fait d'effectuer n'importe quelle opération sans déchiffrer les données ! Cela rend par exemple possible pour deux personnes d'optimiser un partage de ressources, sans révéler à l'autre ses propres préférences ; ou encore – paradoxe ultime! – d'effectuer une recherche sur internet, sans que le moteur de recherche n'apprenne le contenu de notre requête.

Ces nouveaux cryptosystèmes sont basés sur un outil mathématique récemment introduit en cryptographie : le réseau Euclidien. Un prochain article sera consacré à ces nouvelles techniques. Ces derniers résultats furent d'abord très théoriques, dans la mesure où les calculs et les tailles des messages chiffrés semblaient astronomiques ; mais les améliorations successives pourraient permettre à une telle technologie d'être appliquée dans un futur proche.

Conclusion

Cet état des lieux révèle une situation paradoxale : la théorie semble offrir un arsenal de solutions cryptographiques à des scénarios d'utilisation assez complexes, mais en pratique nos données sont extrêmement mal protégées sur Internet. Le fait est que les implémentations de la cryptographie sont essentiellement motivées par les applications commerciales, et de ce fait disponibles en interne pour les entreprises, et mises à la disposition des utilisateurs uniquement dans le cadre très restreint du protocole HTTPS.

Quid de Tor et de GPG ? Il existe en effet des solutions libres permettant de garantir son anonymat et la confidentialité de ses communications, mais elles restent difficiles d'accès aux utilisateurs qui ne seraient pas des power-users. Pour paraphraser Dan J. Bernstein **[14]**, il est temps de développer un logiciel de chiffrement suffisamment simple pour qu'un journaliste soit capable de s'en servir. Mais le problème n'est pas que l'interface logicielle : il y a un énorme effort d'éducation et de sensibilisation à faire et mon avis est qu'il est nécessaire de comprendre les notions de base de la cryptographie pour en faire bonne utilisation. C'est l'objectif du jeu vidéo Cryptris **[15]** par exemple : démystifier les principes de la cryptographie asymétrique.

Un autre challenge est celui d'imposer la cryptographie là où elle serait bénéfique à l'utilisateur, mais pas forcément aux acteurs du net. Les nouvelles techniques comme le chiffrement homomorphe pourraient permettre de garantir à la fois l'utilisabilité des données mises dans le cloud tout en préservant leur confidentialité, mais on ne peut pas vraiment compter sur des acteurs dont l'intérêt est de connaître ces données. Rappelez-vous « si c'est gratuit, c'est que vous – et vos données – êtes le produit ».

Ainsi, un véritable challenge à long terme est de réussir à faire collaborer le monde de la recherche en cryptologie directement avec les communautés open sources. Contrairement à d'autres communautés scientifiques, l'IACR a réussi à imposer l'open-access **[16]** aux éditeurs, et l'on voit apparaître des prototypes de cryptosystèmes innovant en open source. Mais les chercheurs ne sont pas armés pour la mise en pratique et le déploiement : il y a un effort de transfert à faire. Il faudrait certainement imaginer des structures plus rigides qu'à l'habitude pour des projets open source afin d'éviter des mésaventures comme celle de Debian/openSSH **[17]**. Néanmoins, une telle collaboration semble être la seule bonne voie pour développer des solutions pour l'intimité des utilisateurs de la toile.

Un problème majeur serait le financement de telles collaborations. Il n'est pas clair que le crowdfunding soit adapté en termes d'échelle et de durabilité. La recherche en cryptologie, et plus généralement en informatique ne souffre pas trop des difficultés de financement des autres sciences **[18]**. Le financement par projet proposé par les agences nationales et internationales (ANR et ERC) favorise de facto ce domaine (et plus généralement l'informatique). En effet, ces agences apprécient particulièrement les projets incluant une branche transfert, ce qui est bien plus direct en informatique que dans d'autres sciences. Cela pousse les laboratoires à collaborer avec des entreprises et la recherche publique est ainsi détournée vers les intérêts privés, transformée en brevets. Dans le cas particulier de la cryptographie, on favorise ainsi les technologies qui protègent les entreprises (chiffrement en interne, DRM ...),

mais on ne développe rien pour protéger les individus. Dans le cas plus général, on prétend favoriser l'économie nationale ; on oublie peut-être qu'une technologie libre et gratuite peut contribuer grandement au bien commun – ce qui est rappelons-le le but premier de la recherche publique, telle qu'elle à été définie par la loi Française **[19]** – y compris à l'économie **[20]** et aux finances publiques **[21]**.

Tout en arguant pour un changement politique d'envergure, serait-il possible de composer avec le système actuel pour laisser de nouveau aux chercheurs la possibilité de s'attaquer à des problèmes d'intérêt public ? Pourrait-on imaginer que des structures à but non lucratif montent de tels projets types ANR en partenariat avec les laboratoires ? L'idée qu'en République **[22]** l'impôt finance la recherche, mais aussi son transfert et son déploiement dans des produits gratuits et open source semble après tout assez défendable. ■

Références

[1] Affaire Lavabit (2013) : https://www.eff.org/deeplinks/2013/08/lavabit-encrypted-email-service-shuts-down-cant-say-why

[2] La NSA et le DES : https://en.wikipedia.org/wiki/Data_Encryption_Standard#NSA.27s_involvement_in_the_design. La page Wikipedia (EN) offre de nombreuses références, notamment American Cryptology during the cold war (p. 232), recueil de documents déclassifiés. Disponible sur le site de la NSA : https://www.nsa.gov/public_info/_files/cryptologic_histories/cold_war_iii.pdf.

[3] Things the NSA doesn't want you to know and why you should know about it : https://www.nsa-observer.net/

[4] Dual EC DRBG. https://projectbullrun.org/dual-ec/index.html

[5] IACR Statement On Mass Surveillance, Copenhagen resolution, mai 2014 : http://www.iacr.org/misc/statement-May2014.html

[6] Munroe Randall, xkcd 177, Alice and Bob, 2006 : http://www.explainxkcd.com/wiki/index.php/177:_Alice_and_Bob

[7] Rabin Michael, « Digitalized Signatures and Public-Key Functions as Intractable as Factorization », 1979 : http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-212.pdf

[8] La NP-complétude : https://fr.wikipedia.org/wiki/Probl%C3%A8me_NP-complet

[9] Boneh Dan et Franklin Matthew, « Identity Based Encryption From the Weil Pairing », 2001 : https://eprint.iacr.org/2001/090

[10] Partage de clef de Shamir : https:/fr.wikipedia.org/wiki/Partage_de_cl%C3%A9_secr%C3%A8te_de_Shamir. Une implémentation open source est disponible sur http://point-at-infinity.org/ssss/.

[11] Projet open source de vote électronique cryptographique : https://vote.heliosvoting.org/

[12] Pailler Pascal, « Public-Key Cryptosystems Based on Composite Degree Residuosity Classes », 1999 : http://www.lamsade.dauphine.fr/~litwin/cours98/Doc-cours-clouds/Pai99pai.pdf

[13] Gentry Craig, « A fully homomorphic encryption scheme », Thèse de Doctorat, 2009 : https://crypto.stanford.edu/craig/

[14] Bernstein Dan J. , Lange Tanja, et Heninger Nadia, « The year in Crypto », at Chaos Communication Congress, 2013 : https://www.youtube.com/watch?v=G-TM9ubxKIg

[15] Cryptris : un jeu vidéo sur la cryptographie. Présentation vidéo : http://vimeo.com/105507991. Jeu : http://inriamecsci.github.io/cryptris/

[16] Rauzy Pablo, « Le libre accès à la recherche : une introduction », 2014 : http://pablo.rauzy.name/openaccess/introduction.html

[17] Schneier Bruce , « Random Number Bug in Debian Linux », 2008. https://www.schneier.com/blog/archives/2008/05/random_number_b.html

[18] Mouvement Science en Marche, 2014 : http://sciencesenmarche.org/fr/

[19] Points b) et c) de l'article 112-1 Code de la recherche : http://www.legifrance.gouv.fr/affichCode.do?cidTexte=LEGITEXT000006071190.

[20] Groupe de travail Européen, « The impact of Free/Libre/Open Source Software on innovation and competitiveness of the European Union », 2006/2007 : http://www.flossimpact.eu/

[21] Le libre et l'administration publique : http://www.adullact.org/, http://www.journal-officiel.gouv.fr/mimo/

[22] République, du latin Res Publica : Le bien commun.

MONITORING avec RIEMANN

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

Nous ne parlerons pas mathématiques, mais de l'outil de monitoring Riemann baptisé ainsi en hommage au mathématicien du même nom. Il s'agit d'un outil aux concepts différents de Sensu, que nous avons abordé dans un article précédent [1].

Riemann **[2]** est un logiciel qui traite les événements envoyés par les hôtes de votre réseau, orienté monitoring. Il pourra vous envoyer un message d'alerte en cas de charge, détecter le changement d'état d'un service ou encore transmettre des métriques vers un système de graphiques comme Cacti ou Graphite.

Contrairement aux systèmes de monitoring habituels qui « requêtent » les clients (ça consomme de la ressource et c'est long) à intervalles réguliers (intervalles de 1 à 5 minutes généralement, entre lesquels il vous faut patienter avant de savoir si tel ou tel service est revenu à la normale), Riemman est beaucoup plus réactif, spécialement conçu pour les infrastructures fortement réparties. D'une part, ce sont les clients qui contactent le serveur, d'autre part le processus serveur Riemann peut traiter un événement reçu en quelques millisecondes, à raison de plusieurs milliers d'événements par seconde. Une autre particularité, Riemann étant écrit en Clojure, il utilise également Clojure comme langage de configuration, ce qui permet d'écrire des règles très complexes de gestion des flux d'événements.

1 Installation

Riemann est disponible en version 0.2.6 à l'écriture de l'article, sous forme d'archive source, de paquet RPM et Deb. Nous utiliserons ce dernier type de paquet pour installer Riemann sur une Debian 7 Wheezy. Il n'y a d'ailleurs pas de section téléchargement sur le site officiel. Les différents paquets sont disponibles directement depuis la page d'accueil du site du projet.

Clojure, et donc le processus serveur Riemann, fonctionnent sur une JVM Java. On commence donc par installer le JRE sur notre Debian (Java 6 n'étant plus supporté, les versions 7 ou 8 sont à utiliser) :

```
$ sudo aptitude install -y openjdk-7-jre-headless
```

Puis on récupère et on installe l'archive Debian :

```
$ wget http://aphyr.com/riemann/riemann_0.2.6_all.deb -O /tmp/
riemann_0.2.6_all.deb
$ sudo dpkg -i /tmp/riemann_0.2.6_all.deb
```

Vous pouvez maintenant démarrer le service :

```
$ sudo service riemann start
```

Celui-ci écoute sur les ports **5555** en TCP et UDP, et **5556** en Websocket. En cas de souci, n'hésitez pas à jeter un coup d'œil dans le fichier journal **/var/log/riemann/riemann.log**.

2 Principes

Riemann reçoit des événements depuis des clients (votre application, Collectd ou Logstash). Un événement est une structure de type tableau associatif, où les clés sont des champs comme le nom de l'hôte émetteur, le service, l'état, etc. Le serveur va recevoir ces événements et pouvoir les manipuler (les filtrer, les modifier, les combiner, etc.) pour y réagir (envoi d'e-mails,

alimentation d'un serveur de graphiques, etc.) de différentes manières. Cette manipulation d'événements se fait en définissant des flux (streams) : voyez l'arrivée des événements sur le serveur comme un flux global que vous allez canaliser dans d'autres sous-flux de votre choix.

La configuration de votre serveur reviendra à écrire ces flux à l'aide de fonctions Clojure qui prennent des événements en paramètre. La configuration du serveur Riemann est écrite en Clojure, comme le code de l'application elle-même. Le but est de permettre aux utilisateurs de gérer leur configuration Riemann selon leurs désirs, leurs besoins, suivant la philosophie « Configuration as Code ». Là où du XML ou du YAML seraient limités, ici vous pourrez résoudre des cas complexes de supervision d'architecture grâce à la puissance de Clojure. Car, en tant que langage de programmation fonctionnelle, la force de Clojure est d'être orienté données, et donc parfait pour matérialiser ce travail de flux de données. Voyons un exemple de définition d'un flux :

```
(streams

  (where (service "load")

    (graphite {:host "graphite.acme.fr"})

    (where (> metric 1)
       (with :state "warning" reinject))

    (where (state "critical")
      (email "sysadmin@acme.fr"))
))
```

Pas besoin d'être expert Clojure pour comprendre que ce flux va canaliser les événements et, suivant des valeurs de métriques, ou d'état de ces événements, alimenter un serveur Graphite, modifier l'état ou encore envoyer une alerte. Vous trouverez dans la partie Howto de la documentation officielle une multitude d'exemples de flux **[3]**.

En pratique, étudions la configuration de base disponible à l'installation de Riemann, contenue dans le fichier **/etc/riemann/riemann.config** :

```
(logging/init {:file "/var/log/riemann/riemann.log"})

(let [host "127.0.0.1"]
  (tcp-server {:host host})
  (udp-server {:host host})
  (ws-server  {:host host}))

(periodically-expire 5)

(let [index (index)]
  (streams
    (default :ttl 60
      index

      (expired
        (fn [event] (info "expired" event))))))
```

On passe vite sur la première directive **logging/init** qui définit le fichier de log du serveur.

Le deuxième bloc crée une variable **host** avec pour valeur **127.0.0.1**, suivant la syntaxe Clojure **let [nom-variable valeur]**. La variable a une durée de vie dans toute la portée du bloc, afin d'être utilisée dans ce bloc pour démarrer des ports d'écoute TCP et UDP, ainsi qu'en Websocket.

La fonction **periodically-expire** supprime automatiquement de l'index de Riemann toutes les 5 minutes les événements dont la durée de vie (champs **ttl**) a expiré, puis les réinjecte dans Riemann avec l'état **expired**.

L'index de Riemann, Kézako ? Riemann n'a pas pour but de stocker des métriques, ni d'être un historique de vos données. Son intérêt est d'être un système transitoire, vivement réactif aux événements reçus, grâce aux flux. L'index vous permet de connaître l'état global courant de votre infrastructure, incluant l'événement le plus récent pour chaque paire hôte/service reçue. Cet index pourra ensuite être interrogé depuis d'autres clients, comme une interface Web, que nous verrons plus tard.

Revenons à notre fichier de configuration, et le dernier bloc qui est le plus intéressant, contenant la définition de flux définis par défaut. Ces flux sont inclus dans un bloc **let**, qui crée un index à l'aide de la fonction **(index)** dans une variable éponyme. Dans ce bloc dans lequel la variable **index** a effet, on commence par donner un **ttl** par défaut de 60 secondes aux événements qui seraient envoyés à Riemann sans ttl. Ensuite, on utilise la variable **index** pour rendre disponible l'index dans Riemann. Puis la fonction **expired** filtre les événements avec l'état **expired**, et leur applique une fonction (**fn**), qui prend l'événement courant en entrée (**[event]**), et écrit cet événement dans les logs avec la sévérité INFO, avec la fonction **info**.

On va écrire notre premier flux, qui va lui aussi travailler sur les logs. Ajoutez le flux suivant, à la fin de la configuration, hors de tout autre bloc de définition :

```
(streams
  #(info "événement reçu:" %))
```

Cette fonction de flux s'applique à tous les événements (elle n'est pas dans une fonction **expired**) : pour tous les messages reçus, ceux-ci seront loggés avec le message **événement reçu**. La deuxième ligne est un raccourci de la version plus longue **(fn [event] (info "événement reçu:" event))**.

Redémarrez le processus serveur, puis utilisez **tail** avec l'option **-F** pour scruter les logs en direct :

```
$ tail -F /var/log/riemann/riemann.log
```

```
INFO [2014-11-05 21:02:22,831] Thread-8 - riemann.config
- événement reçu: #riemann.codec.Event{:host vagrant,
:service riemann server ws 127.0.0.1:5556 in latency 0.99,
:state ok, :description nil, :metric nil, :tags nil, :time
707608871379/500, :ttl 20}
INFO [2014-11-05 21:02:22,832] Thread-8 - riemann.config
- événement reçu: #riemann.codec.Event{:host vagrant,
:service riemann server ws 127.0.0.1:5556 in latency 0.999,
:state ok, :description nil, :metric nil, :tags nil, :time
707608871379/500, :ttl 20}
```

Vous pouvez constater que des événements sont déjà reçus et loggés, avec le message **événement reçu**, par notre flux, car par défaut le serveur s'envoie à lui-même des messages sur le statut de son service.

Beaucoup de bibliothèques clientes existent pour communiquer avec Riemann et vont vous permettre de l'intégrer facilement dans votre infrastructure de monitoring **[4]**. Que ce soient des librairies clientes disponibles pour plusieurs langages, afin d'être utilisées directement depuis le code de votre application. Ou des extensions pour des systèmes dédiés de collecte de données comme Collectd. Nous allons rester simples, et utiliser un client Ruby dans la suite :

```
$ sudo aptitude install -y ruby1.9.3 build-essential rubygems
$ sudo gem install riemann-client
```

Client que l'on va utiliser depuis un shell ruby interactif :

```
$ irb -r riemann/client
irb(main):001:0>
```

On initialise un client (par défaut, sans plus de paramètre, le client utilisera un serveur à l'adresse **127.0.0.1**) :

```
irb(main):001:0> client = Riemann::Client.new
```

Notre client initialisé, on peut interroger l'index, en précisant un ou plusieurs motifs de recherche. Ici, on utilise **true**, ce qui signifie de récupérer tous les événements actuellement indexés :

```
irb(main):002:0> client['true']
=> [<Riemann::Event time: 1415217967, service: "riemann streams
latency 0.0", host: "vagrant", tags: ["riemann"], ttl: 20.0>,
<Riemann::Event time: 1415217967, state: "ok", service: "riemann
server ws 127.0.0.1:5556 out latency 0.99", host: "vagrant",
ttl: 20.0>, <Riemann::Event time: 1415217967, state: "ok",
service: "riemann server ws 127.0.0.1:5556 in latency 0.5",
host: "vagrant", ttl: 20.0>,...
```

Ou bien on peut forger un événement fictif, en injectant dans notre client un tableau associatif composé des différents champs qui composent un événement :

```
irb(main):003:0> client << {
irb(main):004:1*  service: "www1",
irb(main):005:1*  state:  "critical",
irb(main):006:1*  metric: 3.14
irb(main):007:1> }
```

Puis ensuite interroger à nouveau l'index, et par exemple ne récupérer que cet événement, suivant un motif différent :

```
irb(main):008:0>  client['state="critical"']
=> [<Riemann::Event time: 1415218018, state: "critical",
service: "www1", host: "vagrant", ttl: 60.0, metric_d: 3.14,
metric_f: 3.140000104904175>]
```

On voit que différents champs par défaut ont été rajoutés, comme le ttl. Une ligne équivalente devrait être apparue dans les logs, suite à notre flux écrit précédemment. Si vous passez le délai de 5 minutes de **periodically-expire** sans regénérer manuellement l'événement, l'événement précédent sera sorti de l'index, son état changé à **expired**, et vous devriez le voir dans le fichier de logs :

```
INFO [2014-11-05 21:08:02,351] Thread-8 - riemann.config -
expired {:ttl 60, :time 353804520569/250, :state expired,
:service www1, :host vagrant}
```

On va rajouter le flux suivant à notre configuration :

```
(streams
    (changed-state
       (rollup 5 60
         #(info "changed" %))))
```

Le flux **changed-state** va détecter les événements mis à jour dans l'index dont l'état (champs **state**) a changé depuis l'indexation précédente. Pour les événements concordants, on va écrire de tels événements dans les logs, avec un message **changed**. On n'exécute pas ce flux **info** directement, on le donne en paramètre d'un flux **rollup** : au lieu de signaler l'événement à chaque changement d'état (imaginons que le service change d'état plusieurs fois par seconde, on recevrait autant de notifications), **rollup** va servir de tampon suivant les deux nombres passés en paramètre. Le premier est le nombre maximum de fois que l'on désire appeler la fonction passée en troisième paramètre, durant l'intervalle de temps en secondes donné en deuxième paramètre.

Réinitialisez le client shell Ruby Riemann, et insérez plusieurs événements équivalents au précédent, mais en alternant plusieurs fois la valeur de **state** de **ok** à **critical** :

```
irb(main):052:0> client << { service: "www1",state:  "ok",  metric: 0}
irb(main):053:0> client << { service: "www1",state:  "critical",  metric: 1}
irb(main):054:0> client << { service: "www1",state:  "ok",  metric: 2}
```

```
irb(main):055:0> client << { service: "www1",state:  "critical",  metric: 3}
irb(main):056:0> client << { service: "www1",state:  "ok",  metric: 4}
irb(main):057:0> client << { service: "www1",state:  "critical",  metric: 5}
irb(main):058:0> client << { service: "www1",state:  "ok",  metric: 6}
irb(main):059:0> client << { service: "www1",state:  "critical",  metric: 7}
irb(main):060:0> client << { service: "www1",state:  "ok",  metric: 8}
irb(main):061:0> client << { service: "www1",state:  "critical",  metric: 9}
irb(main):062:0> client << { service: "www1",state:  "ok",  metric: 10}
irb(main):063:0> client << { service: "www1",state:  "critical",  metric: 11}
```

Puis on consulte les logs :

```
INFO [2014-11-05 21:26:29,032] pool-1-thread-37 - riemann.config - changed [#riemann.
codec.Event{:host vagrant, :service www1, :state ok, :description nil, :metric 0, :tags
nil, :time 1415219189, :ttl nil}]
INFO [2014-11-05 21:26:33,933] pool-1-thread-38 - riemann.config - changed [#riemann.
codec.Event{:host vagrant, :service www1, :state critical, :description nil, :metric 1,
:tags nil, :time 1415219193, :ttl nil}]
INFO [2014-11-05 21:26:38,035] pool-1-thread-39 - riemann.config - changed [#riemann.
codec.Event{:host vagrant, :service www1, :state ok, :description nil, :metric 2, :tags
nil, :time 1415219198, :ttl nil}]
INFO [2014-11-05 21:26:39,671] pool-1-thread-40 - riemann.config - changed [#riemann.
codec.Event{:host vagrant, :service www1, :state critical, :description nil, :metric 3,
:tags nil, :time 1415219199, :ttl nil}]
INFO [2014-11-05 21:26:41,300] pool-1-thread-41 - riemann.config - changed [#riemann.
codec.Event{:host vagrant, :service www1, :state ok, :description nil, :metric 4, :tags
nil, :time 1415219201, :ttl nil}]
INFO [2014-11-05 21:27:05,241] riemann task 3 - riemann.config - changed [#riemann.
codec.Event{:host vagrant, :service www1, :state critical, :description nil, :metric
5, :tags nil, :time 1415219202, :ttl nil} #riemann.codec.Event{:host vagrant, :service
www1, :state ok, :description nil, :metric 6, :tags nil, :time 1415219205, :ttl nil}
#riemann.codec.Event{:host vagrant, :service www1, :state critical, :description nil,
:metric 7, :tags nil, :time 1415219207, :ttl nil} #riemann.codec.Event{:host vagrant,
:service www1, :state ok, :description nil, :metric 8, :tags nil, :time 1415219209,
:ttl nil} #riemann.codec.Event{:host vagrant, :service www1, :state critical,
:description nil, :metric 9, :tags nil, :time 1415219211, :ttl nil} #riemann.codec.
Event{:host vagrant, :service www1, :state ok, :description nil, :metric 10, :tags nil,
:time 1415219214, :ttl nil} #riemann.codec.Event{:host vagrant, :service www1, :state
critical, :description nil, :metric 11, :tags nil, :time 1415219216, :ttl nil}]
```

On constate bien que dans l'intervalle défini de 60 secondes, seulement 5 événements similaires sont loggés individuellement, puis au bout de 60 secondes le tampon de tous les événements similaires sont condensés sur une ligne.

Ici nous avons simplement écrit dans les logs, mais vous pourriez tout autant envoyer des e-mails, des sms, ou générer une alerte vers un système de monitoring.

Interroger l'index depuis un client comme la gemme utilisée c'est rapide et pratique, surtout pour le débogage durant l'écriture de vos flux ou le développement de vos propres outils. Riemann dispose néanmoins d'une interface Web pour une utilisation plus visuelle et réactive.

3 Interface Web

L'interface web de Riemann est une gemme Ruby, basée sur le framework Sinatra. Elle se connecte au port Websocket **5556** du serveur Riemann et elle nécessite Ruby 1.9 au minimum. Comme nous avons installé précédemment l'environnement Ruby, on peut installer directement la gemme :

```
$ sudo gem install riemann-dash
```

Par défaut, cette application écoute sur l'interface locale. On peut néanmoins préciser un fichier de configuration optionnel, dans lequel paramétrer, entre autres, l'interface d'écoute. On va stocker cette configuration dans un dossier dédié :

```
$ sudo mkdir /etc/riemann-dash
$ sudo chown riemann /etc/riemann-dash
```

Créez donc le fichier **/etc/riemann-dash/config.rb** de configuration minimale suivant pour écouter sur toutes les interfaces réseaux :

```
set :bind, '0.0.0.0'
config[:ws_config] = '/etc/riemann-dash/workspace.json'
```

On a également indiqué un fichier **/etc/riemann-dash/workspace.json**, qui sert de configuration du tableau de bord affiché par l'interface Web. C'est un fichier de description au format **json**, que vous pouvez créer et éditer vous même. Pour faire simple, nous allons nous contenter de profiter du fait que l'on peut paramétrer la disposition du tableau de bord et sauvegarder le résultat directement en ligne depuis l'interface, sans avoir à connaître la syntaxe de description, et avoir à éditer manuellement ce fichier.

Nous allons également créer un script de démarrage pour lancer l'interface comme un service, car la gemme **riemann-dash** ne fournit pas de tel script. On va tout simplement se servir du script du serveur Riemann comme base :

```
$ sudo cp -a  /etc/init.d/riemann /etc/init.d/riemann-dash
```

Éditez ce fichier **/etc/init.d/riemann-dash** comme suit pour le personnaliser et l'adapter à ce nouveau service :

```
...
# Provides:          riemann-dash
...
# Short-Description: Riemann Dashboard
# Description:       The Riemann monitoring dashboard.
...
DESC="Riemann Dashboard"
NAME=riemann-dash
DAEMON=/usr/local/bin/riemann-dash
...
RIEMANN_CONFIG=${RIEMANN_CONFIG:-${RIEMANN_PATH_CONF}/config.rb}
...
```

Vous pouvez ensuite démarrer l'interface en tant que service :

```
$ sudo service riemann-dash start
```

Fig. 1 : Le tableau de bord par défaut de l'interface Web.

Ouvrez votre navigateur à l'adresse de la machine sur laquelle vous avez installé l'interface, port **4567**, pour valider qu'elle est bien disponible (voir figure 1).

Par défaut, le tableau de bord ne comporte qu'un onglet ***Riemann*** (symbolisé par le rectangle grisé en haut à gauche), divisé en deux panneaux horizontaux. En haut à droite, une zone préremplie avec **127.0.0.1:5556** indique à quelle adresse atteindre le service Websocket mis à disposition par le serveur. Cette adresse est relative au navigateur exécutant l'interface Web, pas au processus **riemann-dash** exécutant cette interface : ici, le navigateur s'exécute sur la même machine que le serveur Riemann, donc tout est bon.

Par défaut, les panneaux affichés ne sont pas très utiles : celui du dessus est de type **Title** pour afficher un titre (sic!), celui du dessous de type **Help** pour afficher l'aide (re-sic!). Comme indiqué dans cette aide, on peut modifier les type et contenu d'un panneau. Pour cela on commence par sélectionner un panneau en cliquant dessus en maintenant la touche **<Ctrl>** enfoncée, puis en appuyant sur **<e>**. Comme documenté dans la prise en main rapide sur le site officiel **[5]**, on va passer ce panneau de type **Title** en type **Grid**, qui permet d'afficher des événements

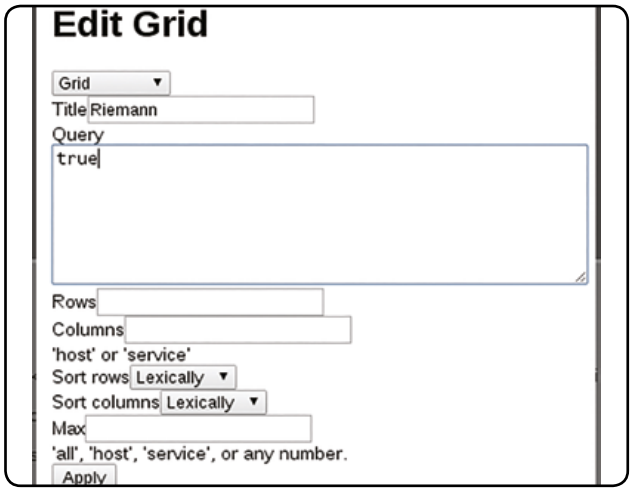


Fig. 2 : Modification d'un panneau.

à partir d'une requête sur l'index. Comme requête, mettez tout simplement **true** pour récupérer tous les événements (voir figure2). Cliquez sur **Apply**, le panneau créé affichera une ligne pour chaque machine, contenant tous les services pour lesquels un état d'événement est indexé (ici **vagrant**, pour le nom de la machine sur lequel est installé Riemann, pour les métriques envoyées par le serveur Riemann à lui-même)(voir figure 3). Appuyez sur **<s>** pour sauvegarder la configuration du tableau de bord, qui sera écrite dans le fichier **/etc/riemann-dash/workspace.json**, défini précédemment dans le fichier de configuration **/etc/riemann-dash/config.rb.**

Rien de fantastique comme tableau de bord pour le moment : outre le fait que l'interface ne soit pas très sexy (mais très rapide pour afficher les métriques), nous n'avons pas beaucoup de données à afficher. Le projet Riemann offre un client facile à installer, et prêt à l'emploi pour envoyer des métriques de base d'un hôte (utilisation cpu, mémoire, réseau) au serveur. Il suffit d'installer la gemme **riemann-tools** :

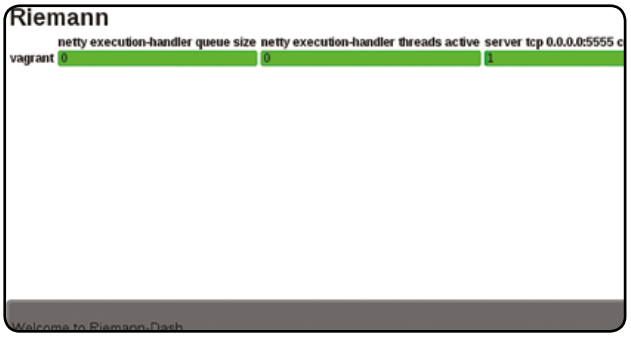


Fig. 3 : Panneau affichant pour chaque machine tous les services pour lesquels un état d'événement est indexé.

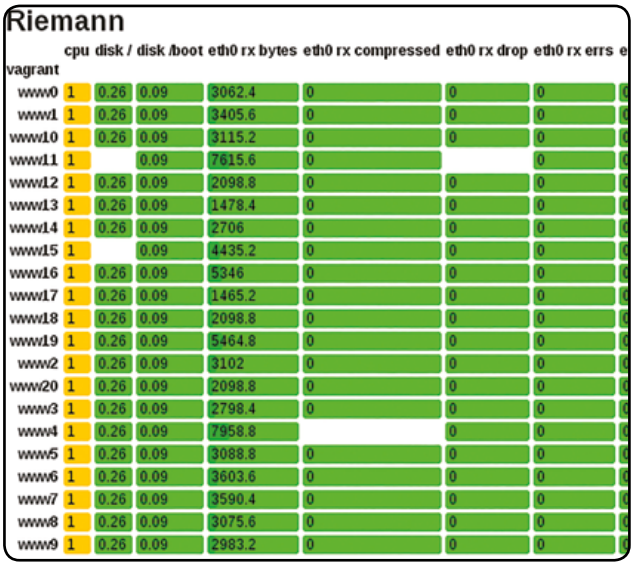


Fig. 4 : Affichage des informations en temps réel sur l'interface Web

```
$ sudo gem install riemann-tools
```

Cette gemme contient plusieurs petits outils pour envoyer des événements à un serveur Riemman pour des services comme Nginx, Haproxy ou Memcached. Ceux qui nous intéressent ici sont **riemann-health** et **riemann-net** :

```
$ riemann-health  &
$ riemann-net  &
```

Note

On a démarré ces services en tant que tâches de fond, vous pouvez tout aussi bien créer un fichier **init.d** pour gérer ces deux processus sous forme de services, comme on l'a fait avec **riemann-dash** pour l'interface Web.

Si vous jetez un œil à votre tableau de bord, la ligne des métriques qui est affichée pour le serveur devrait être complétée par des métriques sur le cpu, le débit, etc. grâce à ces outils.

Les deux outils précédents peuvent accepter plusieurs options, comme des seuils à la manière d'un check Nagios. On peut également forger le nom de l'hôte émetteur. Combinons ces différentes options dans une boucle shell, pour simuler plusieurs hôtes clients communiquant avec notre serveur :

```
$ for i in {0..20}
> do
> riemann-net --event-host www${i} &
> riemann-health --event-host www${i} --load-warning 1 --cpu-critical 80 &
> done
```

Cette boucle va simuler une vingtaine d'hôtes clients nommés de **www1** à **www20**, qui seront affichés dans le tableau de bord. Ces processus, plus le processus serveur, plus l'interface Web tournent tous sur une même machine virtuelle de test, et l'interface Web affiche toutes ces informations en temps réel, de manière très fluide et réactive (voir figure 4).

Imaginez plus d'hôtes clients et une meilleure organisation de l'espace de travail à l'aide d'onglets, de panneaux horizontaux et verticaux, vous obtenez un tableau de bord temps réel optimal de votre infrastructure.

Conclusion

Les choix techniques de Riemann peuvent freiner son adoption : il est fortement lié à Clojure, il n'est pas un outil de monitoring prêt à l'emploi « out-of-the-box », et il n'intègre pas de base la compatibilité avec les checks Nagios existants comme le fait Sensu. Mais ces inconvénients sont en fait ses qualités, car il est extrêmement configurable et flexible, pensé pour les systèmes distribués, et met l'accent sur la rapidité. Si vous êtes à la recherche d'un middleware facilement intégrable dans votre infrastructure pour faire du monitoring, il mérite largement votre intérêt. ■

Références

[1] Benedetti B., « Graphing, logging et monitoring 2.0 : vos alertes avec Sensu », GNU/Linux Magazine n°170, avril 2014, p.66 à 79.

[2] Le site officiel de Reimann : http://riemann.io

[3] Cas pratiques d'utilisation : http://riemann.io/howto.html

[4] Outils clients : http://riemann.io/clients.html

[5] Tutoriel de prise en main rapide de Riemann et son Dashboard : http://riemann.io/quickstart.html

PAVE : DÉPLOIEMENT À DISTANCE DE SERVEURS HÉTÉROGÈNES

*par **Julien Nycz** [Sysadmin, dba, devops and security engineer. Utilisateur GNU/Linux depuis 1999.]*

La gestion de parcs de machines hétérogènes est devenue monnaie courante de nos jours avec l'émergence des clouds et des différents services du type Amazon EC3, ce qui rend l'administration système et les phases de déploiement de plateformes de plus en plus complexes.

1 Déploiement de plateformes : quelle stratégie adopter ?!

Pour répondre à la problématique de déploiement de plateformes hétérogènes, le marché des outils de type « administration à distance » est déjà bien fourni. On peut citer, parmi les plus utilisés, Puppet, Chef, Ansible, CFEngine, Salt, Fabric, …

Tous ces outils sont très complets, couvrent un large spectre de fonctionnalités… mais sont assez complexes à prendre en main, comportent souvent de multiples dépendances ou possèdent un coût initial d'installation important. De plus, ils sont souvent très orientés vers les besoins entreprises.

Parmi les dépendances, on peut trouver des fichiers XML, des templates Jinja, du Ruby, ou encore des démons (qui ajoutent une couche supplémentaire de communication et participent à un gâchis de ressources sur les serveurs cibles). Ces dépendances sont souvent le point bloquant, notamment dans des contextes de sécurité renforcés ou quand elles n'existent pas sur les serveurs cibles (portage inexistant, problème de version cible…).

2 Constat : des ressources communes déjà disponibles sur nos serveurs

Dans ma société, nous possédons des centaines de parcs machines totalement hétérogènes à administrer.

En fonction du cycle de vie des projets, des contextes de sécurité ou des besoins clients, on se retrouve souvent à devoir intervenir sur des parcs totalement différents pour effectuer une même opération (modifier des fichiers, créer des arborescences, exécuter des commandes…).

Malgré une telle disparité, on peut néanmoins retrouver de façon systématique des ressources disponibles tels que des interpréteurs (Perl, Python, Bash), des outils communs tels que les Unix *tools* (sed, awk, etc.), ou encore pour Windows : le powershell.

On peut aussi identifier des protocoles de transport déjà disponibles : SSH pour Linux ou *Windows Remote Management* (WinRM) pour Windows (que bien sûr nous n'aborderons pas dans cet article).

3 Less is more !!!

Comment faire vite et bien, avec peu et de façon efficiente ?

Partons d'un outil comme Fabric **[1]** déjà présenté il y a quelques temps dans notre magazine préféré **[2]**. Fabric a pour grand intérêt de n'utiliser aucune dépendance sur les serveurs distants et de passer par le protocole SSH pour joindre nos serveurs Linux distants. Il n'est pas typé gestion de configuration, ce qui est un bon point pour répondre à notre problématique de déploiement. Ajoutons-y une couche d'abstraction en Python et une interface utilisateur à base d'un fichier de configuration de type YAML et nous obtenons ainsi un outil bien pratique appelé **Pave** qui répond à notre besoin !

4 Going to « Pave » my servers

Pour cela, il nous faut :

- Un serveur central sur lequel l'interpréteur Python est installé,
- Un accès SSH entre notre serveur central et nos serveurs à construire (par clé ou par mot de passe).

1, 2, 3… Allons construire nos serveurs !

4.1 Installation

Pave peut être installé via le gestionnaire de paquetages **pip** de Python :

```
$ sudo pip install pave
```

4.2 Configuration

Pour utiliser **pave**, il faut configurer le fichier **pave.yml** qui contient les informations sur les opérations à effectuer sur nos serveurs distants.

Pour cela, nous allons créer un squelette de ce fichier effectuant un simple « Hello Pave World » :

```
$ pave -S
  ✓ INFO    main: created "pave.yml".
```

Et le « pavefile » au format YAML (le fichier **pave.yml**) :

```
# starter pavefile

vars:
    user: ubuntu

main:
    user: %user
    sudo: True
    targets:
        - 192.168.2.1    # a list/string of hosts/groups

tasks:
    - packages:
        if-platform: Debian
        install: sysvbanner

    - packages:
        if-platform: Redhat
        install: banner

    - banner "Hello Pave World!"
```

Après avoir remplacé les informations de connexion concernant l'utilisateur distant (**%user**) et l'adresse IP (dans la section **targets**), nous pouvons interagir avec notre serveur.

4.3 Une première pierre à l'édifice

Notre premier pavefile va pour l'instant installer en fonction de l'OS le paquetage contenant le binaire **banner** et l'appeler pour effectuer notre « Hello world » :

```
$ pave
  ✓ INFO    main: pave version 0.68, Python: 2.7.5, fabric: 1.8.1
                pavefile: ./pave.yml, logs: /home/user/.cache/pave/logs
  ★ NOTE    main: paving user@192.168.2.1 with 3 tasks…
  ✓ INFO    transport: Connected (version 2.0, client OpenSSH_6.0p1)
  ✓ INFO    transport: Authentication (publickey) successful!
  ✓ INFO    main: found a(n) Debian system, version 7.1.
  ✓ INFO    packages: inventory: collecting installed pkg list.
  » SKIP     packages: update: occurred recently.
  » SKIP     packages: upgrade: occurred recently.
  » SKIP     packages: install: NADA—inventory complete.
  » SKIP     packages: test(s) returned a False.
  ✓ INFO    run: banner "Hello Pave World!"…
  ★ NOTE    main: results {u'remote_server': (0, 0)}
```

L'ensemble des traces liées à chaque opération se retrouve dans un fichier de logs par serveur sous **$HOME/.cache/pave/logs**.

L'ensemble des options du client **pave** est disponible dans la documentation officielle **[3]**.

5 Documentation du « pavefile »

L'ensemble des tâches et des options configurables du fichier pavefile sont disponibles depuis la documentation en ligne **[4]**. On peut notamment y trouver différentes fonctionnalités disponibles, telles que l'installation de paquetages, la configuration de fichiers à distance, le déploiement d'applications Django,la gestion des utilisateurs et des groupes, le déploiement et la manipulation de bases de données postgreSQL, l'exécution de commandes, la copie de fichiers, la gestion des services, la manipulation de gestionnaires de sources (SVN, hg, Git), ou encore la création de modules Pave.

6 Fonctionnalités avancées

Parmi les options intéressantes, on peut relever le lien avec un **fabfile.py** issu de Fabric.

Bien que l'ensemble de l'outil repose sur l'API de Fabric, il est aussi possible de connecter Pave avec un **fabfile.py** externe et ainsi d'utiliser des tâches personnalisées :

```
main:
    env:
        fabfile: ../fabric/fabfile.py  # optional => default ./fabfile.py

fab-tasks:
    - task1
    - task2: arg1
- task2: arg2,arg1=val1
```

Conclusion

Derrière son petit air de trousse à outils, Pave se montre très puissant et montre qu'avec très peu de configuration ou de dépendances, on peut déployer rapidement et facilement un ensemble de serveurs faisant partie de parcs totalement hétérogènes. ■

Références

[1] Site officiel de Fabric : http://docs.fabfile.org

[2] HEITOR E., « Administration distante d'un parc dynamique et hétérogène avec FABRIC », GNU/Linux Magazine n°135, février 2011.

[3] Documentation officielle du client Pave : https://pave.readthedocs.org/en/latest/cmdline.html

[4] Options du fichier « pavefile » : https://pave.readthedocs.org/en/latest/pavefile.html

L'ENVIRONNEMENT POSIX DU MINI SERVEUR EMBARQUÉ EN C

par ***Yann Guidon*** *[Électronique, musique et informatique en folie^Wliberté]*

Après les explications données précédemment sur le fond et la forme de HTTaP [1][2], nous pouvons commencer à coder notre petit serveur embarqué. Nous nous concentrons sur les fonctions de bas niveau essentielles au support du protocole HTTP/1.1. En effet, nous devons d'abord régler de nombreux détails en langage C, comme la configuration et les droits d'accès, en utilisant des techniques de codage communes aux autres types de serveurs TCP/IP.

HTTaP est un protocole construit au-dessus de HTTP/1.1, dont il hérite certaines caractéristiques, mais qui ne supporte que celles qui sont strictement nécessaires. Notre serveur est constitué d'un seul ensemble de fonctions en C et il doit fonctionner dans deux modes (« *pollé* » et *bloquant*) et dans deux environnements : comme un programme autonome, ou bien embarqué dans un autre logiciel qui peut être dans un autre langage (nous utilisons le VHDL). Les fonctions spécifiques à l'application sont alors prises en charge par un *wrapper*, du code qui interface l'application avec le serveur HTTaP.

Nous nous appuyons sur le code développé pour le serveur HTTP/0.9 qui contrôle une LED, branchée sur le port GPIO d'un Raspberry Pi **[3]**. La version de départ est téléchargeable sur le dépôt GitHub du magazine et nous allons :

- configurer le serveur (avec les variables d'environnement),
- modifier les droits d'accès POSIX,
- permettre de changer à la volée le mode de fonctionnement du serveur (appels bloquants ou non bloquants).

Comme pour les articles précédents, nous allons ajouter ou modifier progressivement un aspect précis du code, ce qui facilite les vérifications à chaque étape. L'ordre des modifications est important, car chacune permet à la suivante de fonctionner avec un minimum d'effort.

1 Configuration

L'utilisateur doit être en mesure de changer les valeurs par défaut de plusieurs paramètres du serveur. Un programme normal lit habituellement ces paramètres sur sa ligne de commandes, un programme plus lourd comme Apache *parsera* des fichiers de configuration. La première méthode est impossible à réaliser avec GHDL, l'autre est beaucoup trop lourde. Heureusement, une troisième approche a été déjà présentée : utiliser les variables d'environnement **[4]**.

Le serveur configure déjà un paramètre par ce moyen :

```
char *env_port;
[...]
case ETAT0_initialisation :
  /* gestion du numéro du port */
  env_port = getenv("GHDL_TCPPORT");
  if ((env_port == NULL) || (env_port[0]==0))
    env_port=PORT_NUMBER;
  printf("Port: %s, ",env_port);
```

On peut noter que je teste les variables d'environnement de deux manières. La valeur de retour normale est **`NULL`** lorsque **`getenv()`** ne trouve pas la variable, mais celle-ci peut être vide. En effet, il existe des gens qui ne connaissent pas la commande **`unset`** et qui croient effacer une variable en y affectant une chaîne vide (*ça sent le vécu...*).

Donc je teste aussi la longueur, mais pas besoin de **strlen()**. Si le premier caractère de la chaîne est à zéro (ce qui correspond au zéro terminal d'une chaîne en C), alors on considère que la variable est absente. Pour alléger le reste du code source, j'ai ajouté ce test dans une petite fonction qui enrobe **getenv()** :

```
void *my_getenv(char *s) {
  s=getenv(s);
  if (s && s[0])
    return s;
  return NULL;
}
```

Nous pouvons maintenant initialiser les autres paramètres. On obtient ce code modifié et allongé :

```
#define KEEPALIVE   15 /* en secondes */
[...]
int keepalive;
char *chemin_statique, *page_racine;
[...]
  /* gestion du numéro du port */
  env_port = my_getenv("GHDL_TCPPORT");
  if (!env_port)
    env_port=PORT_NUMBER;

  /* Durée de persistance de la connexion */
  keepalive = KEEPALIVE;
  b = my_getenv("GHDL_KEEPALIVE");
  if (b) {
    keepalive = atol(b);
    if ((keepalive < 3) || (keepalive > 200))
      erreur("GHDL_KEEPALIVE doit être entre 3 et 200");
  }
  printf("Port: %s, Keepalive: %ds\n",
      env_port, keepalive);

  /* Chemin vers le répertoire des fichiers statiques */
  chemin_statique = my_getenv("GHDL_STATICPATH");
  if (!chemin_statique)
    chemin_statique = "."; // Le répertoire courant

  /* Nom du fichier pour l'URI racine
     (habituellement index.html) */
  page_racine = my_getenv("GHDL_ROOTPAGE");
  if (!page_racine)
    page_racine = "index.html";
  printf("Chemin: %s  Page racine : %s\n",
      chemin_statique, page_racine);
```

Ainsi, on peut modifier ces paramètres avec les commandes suivantes :

```
$ export GHDL_TCPPORT=61234
$ export GHDL_KEEPALIVE=123
$ export GHDL_STATICPATH=/data
$ export GHDL_ROOTPAGE=index.txt
$ ./serv
Port: 61234, Keepalive: 123s
Chemin: /data  Page racine : index.txt
IPv4 found, IPv6 found, v4 over v6 => close(v4)
...
```

Et si vous tenez absolument à maintenir un fichier de configuration, vous pouvez garder tous ces paramètres dans un fichier : il suffit simplement d'écrire un script shell !

Le code source de cette version est disponible sous le nom **serv_7_config.c**.

2 Abandon des privilèges

Pour réduire l'impact d'éventuelles failles dans le serveur, ou simplement éviter des erreurs de manipulation, une méthode courante est de changer l'utilisateur du programme afin qu'il ne dispose que d'un minimum de privilèges. Le changement d'utilisateur permet en particulier de n'accéder qu'à certains fichiers qui auront été marqués au préalable avec la commande **chown**. Cela simplifie notre code de serveur, qui a seulement besoin de vérifier le propriétaire des fichiers à renvoyer au client.

En pratique, c'est un peu plus compliqué que cela. Les fonctions que nous allons utiliser ont chacune besoin de certains paramètres et d'un environnement particulier pour s'exécuter correctement. L'ordre des appels est important, par exemple **setuid()** ne peut s'exécuter qu'après l'appel à **setgid()**, qui lui-même ne peut s'effectuer qu'après le changement de répertoire...

Le code semble un peu confus, car la séquence logique a l'air de partir dans tous les sens. Pour y voir plus clair, la séquence est divisée en trois parties : lecture de la configuration, mise en place des répertoires et enfin changement de l'utilisateur.

D'abord, le nouveau nom et le nouveau groupe sont transmis en passant par les variables d'environnement, qui sont lues juste après le code du chapitre précédent :

```
#include <pwd.h>  // permet de lire /etc/passwd
#include <grp.h>  // permet de lire /etc/group
[...]
uid_t prog_uid;
[...]
```

```
char *env_user, *env_group;
struct passwd *pswd=NULL;
struct group *grp=NULL;
[...]
/* détecte l'utilisateur et le groupe */
env_user = my_getenv("GHDL_USER");
if (env_user) {
  errno = 0;
  pswd = getpwnam(env_user);
  if (!pswd)
    erreur("$GHDL_USER invalide ");

  printf("User: %s", env_user);
  prog_uid = pswd->pw_uid;

  env_group = my_getenv("GHDL_GROUP");
  if (env_group) {
    errno = 0;
    grp = getgrnam(env_group);
    if (!grp)
      erreur("$GHDL_GROUP invalide ");
    printf(", Group: %s", env_group);
  }
}
else
  prog_uid = getuid();
printf("\n");
```

À la fin, on s'assure que le numéro de l'utilisateur est bien connu, car nous le comparerons ensuite avec le propriétaire de chaque fichier accédé par le serveur.

Par défaut, l'utilisateur courant peut travailler normalement, mais le programme prend une autre identité si on lui en indique une. Normalement, on peut faire appel à l'utilisateur **nobody** et au groupe **nobody**, mais vous pouvez gérer les droits plus finement en créant d'autres utilisateurs et groupes. Mais attention : pour que **setuid()** fonctionne, le programme doit être lancé par l'utilisateur **root** ou être marqué avec l'attribut **Set-UID**.

Une fois que les paramètres sont connus, il est temps de vérifier l'accessibilité du répertoire de travail et s'y déplacer. On s'assure aussi que le répertoire appartient bien à l'utilisateur, ce qui évite des erreurs 403 plus tard.

```
#include <sys/stat.h>
[...]
struct stat stat_buf;
[...]
/* Vérifie le chemin pour accéder aux fichiers statiques */
if (stat(chemin_statique, &stat_buf)
    || !S_ISDIR(stat_buf.st_mode))
  erreur("$GHDL_STATICPATH : Chemin invalide");
if (stat_buf.st_uid != prog_uid)
  err("L'utilisateur n'est pas propriétaire de $GHDL_
STATICPATH\n");
if (chdir(chemin_statique))
  erreur("Echec de déplacement vers $GHDL_STATICPATH");
if (chroot(chemin_statique))
  perror("Warning: Echec de chroot() ");
/* Vérifie le fichier racine */
if (stat(page_racine, &stat_buf)
    || !S_ISREG(stat_buf.st_mode))
  erreur("$GHDL_ROOTPAGE : Fichier invalide");
if (stat_buf.st_uid != prog_uid)
  err("L'utilisateur n'est pas propriétaire de $GHDL_ROOTPAGE\n");
```

Grâce à **chdir()**, il ne sera pas nécessaire de concaténer l'URI au chemin statique pour trouver les fichiers. Mais le plus intéressant est l'usage de **chroot()**, car en changeant le répertoire racine, on économise du code qui vérifie si l'URI remonte dans le système de fichiers. Cela réduit encore le risque qu'une adresse mal formée accède à un fichier sensible.

Par contre, seul l'utilisateur **root** peut effectuer cette opération. Si un autre compte l'exécute, un avertissement est affiché et le programme continuera quand même. L'accès aux fichiers sensibles est réduit, car nous allons vérifier le propriétaire des fichiers servit, mais cela ne garantit pas une sécurité totale.

Aussi, **chroot()** empêche la lecture de tous les fichiers qui ne sont pas dans le sous-répertoire courant, donc **/etc/passwd** est inaccessible et **getpwnam()** échoue si on ne respecte pas l'ordre des appels.

Au troisième temps de la valse, nous pouvons enfin changer l'identité du programme.

```
/* change le groupe PUIS l'utilisateur (après avoir chrooté) */
if (pswd){  // GHDL_USER existe
  if (grp) {
    if (setgid(grp->gr_gid))
      erreur ("Echec de changement de groupe ");
  }
  if (setuid(pswd->pw_uid))
    erreur ("Echec de changement d'utilisateur ");
}
```

Maintenant, il y a un peu moins de trous dans la passoire et le programme fonctionne quand même en tant qu'utilisateur normal.

```
$ export GHDL_TCPPORT=61234
$ export GHDL_KEEPALIVE=123
$ unset GHDL_STATICPATH
$ unset GHDL_ROOTPAGE
$ touch index.html
$ ./serv
Port: 61234, Keepalive: 123s
Chemin: .  Page racine : index.html
```

```
Warning: Echec de chroot() : Operation not permitted
IPv4 found, IPv6 found, v4 over v6 => close(v4)
...
```

Mais en cas d'utilisation publique, assurez-vous d'exécuter le serveur à partir du compte **root**, ou bien faites compiler le programme par **root** et activez le bit **SETUID** (voir le script de **passport** **[5]**) :

```
$ su
Mot de passe :
# cd /tmp/
# mkdir srv
# touch srv/index.html
# chgrp -R nobody srv
# chown -R nobody srv
# export GHDL_USER=nobody
# export GHDL_GROUP=nobody
# unset GHDL_ROOTPAGE
# export GHDL_STATICPATH=/tmp/srv
# unset GHDL_TCPPORT
# unset GHDL_KEEPALIVE
# /articles/serv/src/serv
Port: 60000, Keepalive: 15s
Chemin: /tmp/srv  Page racine : index.html
User: nobody, Group: nobody
IPv4 found, IPv6 found, v4 over v6 => close(v4)
...
```

Le code source de cette version est disponible sous le nom **serv_8_change-user.c**.

3 Basculer entre le mode bloquant et le mode « pollé »

L'idée de réaliser un serveur non bloquant est très bonne mais avec le recul, j'ai trouvé que ce n'était pas encore la panacée. Cela a commencé quand je me suis rendu compte que je devais réaliser deux versions du serveur, afin que le code source puisse être utilisé dans plus de projets :

- La première version du serveur est classique : tant qu'il ne se produit rien, il ne fait rien. Il est dit bloquant, car son thread est mis en berne en absence d'activité sur le réseau. Cela réduit l'activité du microprocesseur au minimum, ou l'accapare lorsque les requêtes sont nombreuses. C'est le serveur typique, tel qu'il est traditionnellement réalisé, même notre cas est souvent différent.
- Pour la deuxième version, le serveur est intégré dans un autre programme qui doit aussi fonctionner (une simulation GHDL, au hasard), donc il est hors de question de bloquer arbitrairement tout le programme lorsqu'on attend des données venant du réseau. La partie serveur doit être réveillée très souvent (par exemple 4 à 10 fois par seconde pour maintenir une bonne interactivité) par le programme principal (qui peut avoir beaucoup de choses à faire). Puisqu'il faut interroger périodiquement le serveur, on dit que ce dernier est « pollé » (selon l'anglicisme consacré qui signifie à peu près « scruté »).

Mais il apparaît que cela ne suffit pas et on aimerait passer d'un mode à l'autre à volonté. Lors de l'attente d'une requête du client, lorsque la simulation est en pause, il est dommage que le simulateur continue d'interroger la socket dix fois par seconde, il faudrait plutôt passer en mode bloquant. En fait, il serait naturel d'utiliser le mode bloquant lorsque le simulateur est en pause ou en attente, et de revenir au mode pollé durant les simulations.

L'article sur la couche TCP/IP abordait déjà ce sujet et a montré comment transformer un code bloquant en code pollé au moyen d'une machine à états, réalisée avec un gros *switch-case* **[6]**. Les aspects techniques ne sont pas très compliqués et il n'y a que deux endroits à modifier.

- Juste après l'ouverture de la socket, on rend la fonction **accept()** non bloquante en appelant **fcntl()** ainsi :

```
/* Débloque la socket */
flags = fcntl(sock_serveur, F_GETFL);
fcntl(sock_serveur, F_SETFL, flags | O_NONBLOCK);
```

- Pour la lecture de la requête, **recv()** devient bloquant si on utilise l'option **MSG_DONTWAIT** :

```
case ETAT2_attente_donnee_entrante :
  /* attend quelque chose */
  if (recv(sock_client, buffer, TAILLE_TAMPON, MSG_DONTWAIT) <
0) {
    if ((errno == EWOULDBLOCK) || (errno == EAGAIN))
      return;
    else
      erreur("Echec de réception du message du client");
  }
```

Un dernier détail est que si le serveur est entièrement en mode bloquant, la machine à états doit pouvoir continuer de tourner, donc la fonction **serveur()** est appelée dans une boucle infinie, comme pour la version pollée, mais sans temporisation :

```
int main(int argc, char *argv[]) {
  while (1) {
    serveur();
  }
}
```

À partir de là, on voit facilement les modifications à effectuer. Pour commencer, le plus simple est de remplacer la constante **MSG_DONTWAIT** par une variable, qui prend soit cette valeur constante, soit zéro. En prime, cette variable indique que le mode actuel est bloquant lorsqu'elle est à zéro.

```
int mode_polled=MSG_DONTWAIT;
[...]
case ETAT2_attente_donnee_entrante :
  /* attend quelque chose */
  if (recv(sock_client, buffer, TAILLE_TAMPON, mode_polled) < 0)
{
[...]
```

Pour changer cette variable à partir du code VHDL, créons maintenant deux fonctions (vues comme des procédures par GHDL) qui vont mettre les bonnes valeurs :

```
void serv_polled() {
  mode_polled=0;
}

void serv_bloquant() {
  mode_polled=MSG_DONTWAIT;
}
```

La suite est à peine plus compliquée : on déplace le code qui change l'option de socket vers ces petites fonctions. La variable temporaire **flags** est aussi renommée et déplacée, et le tour est joué !

```
int fcntl_flags;

void serveur_polled() {
  mode_polled=MSG_DONTWAIT;
  fcntl(sock_serveur, F_SETFL, fcntl_flags | O_NONBLOCK);
}

void serveur_bloquant() {
  mode_polled=0;
  fcntl(sock_serveur, F_SETFL, fcntl_flags);
}
```

```
[...]

/* Débloque la socket */
fcntl_flags = fcntl(sock_serveur, F_GETFL);
serveur_polled();
```

Les fonctions **serveur_bloquant()** et **serveur_polled()** peuvent être appelées en VHDL au travers de l'interface VHPI et ainsi, le wrapper VHDL peut contrôler lui-même le mode de fonctionnement optimal du serveur.

Pour vérifier que le nouveau système fonctionne correctement, voici une idée un peu saugrenue, mais simple : appeler **serv_bloquant()** et **serv_polled()** dans le serveur Web lui-même. Le serveur peut déjà contrôler une LED, il suffit de remplacer l'accès aux broches par les deux fonctions. Mais d'abord, pour témoigner de l'activité du programme appelant, on va restaurer le code qui fait tourner une barre dans le terminal.

```
int main(int argc, char *argv[]) {
  const char spinner[4]="-\\|/";
  int phase=0;
  while (1) {
    usleep(100*1000);   /* attente de 100ms soit 10Hz */
    printf("%c%c", spinner[(phase++)&3],0xd);
    fflush(NULL);
    serveur();
  }
}
```

L'appel aux nouvelles fonctions remplace l'allumage et l'extinction de la LED à l'endroit où on décode l'URL :

```
/* analyse de la requête */
if ((recv_len >= 7) && (strncmp("GET /1 ", buffer, 7) == 0)) {
  printf("Allumage de la LED\n");
  b=LED_On;
  serveur_bloquant();
}
else {
  if ((recv_len >= 7) && (strncmp("GET /0 ", buffer, 7) == 0)) {
    printf("Extinction de la LED\n");
    b=LED_Off;
    serveur_polled();
  }
```

Si tout va bien, le terminal devrait arrêter de s'agiter lorsque vous accédez à `http://127.0.0.1:60000/1` et recommencer lorsque vous consultez `http://127.0.0.1:60000/0`.

Le code source de cette version est disponible sous le nom `serv_9_blocant.c`.

Pour finir (pour l'instant)

Le petit serveur commence lentement à se doter de fonctions dignes des grands, mais il est loin d'être fonctionnel. Les prochains articles s'occuperont d'interrompre les connexions inactives, de mettre en place les réponses et codes d'erreur, d'analyser la commande de la requête et les paramètres de l'entête, de trouver et lire un fichier, donc de supporter HTTP/1.1. On pourra alors (re)coder le loopback de fichiers **[7]**, ainsi que le système (spécifique à GHDL) pour configurer la liste des signaux et de leurs propriétés.

À bientôt ! ■

Références

[1] Guidon Y., « Pourquoi utiliser HTTP pour interfacer des circuits numériques ? », GNU/Linux Magazine n°173, juillet 2014, p. 38.

[2] Guidon Y., « HTTaP : Un protocole de contrôle basé sur HTTP », GNU/Linux Magazine n°173, juillet 2014, p. 44.

[3] Mise en place du serveur HTTP primitif : Guidon Y., « Comment contrôler les GPIO du Raspberry Pi par HTTP en C », Open Silicium n°6, mars 2013/

[4] Guidon Y., « GHDL et les chaînes de caractères : application à getenv() », GNU/Linux Magazine n°131, octobre 2010. Code source : http://ygdes.com/GHDL/ghdl_env/

[5] Guidon Y., « Interfaçage de GHDL avec le port parallèle sous Linux », GNU/Linux Magazine n°133, décembre 2010, p.74. Code source : http://ygdes.com/GHDL/io_port/

[6] Création d'un serveur TCP/IP minimal : Guidon Y., « Un mini serveur HTTP pour dialoguer avec des applications interactives, partie 1 : les sockets réseau », GNU/Linux Magazine n°141, septembre 2011, p. 44.

[7] Guidon Y., « Accéder aux fichiers en JavaScript (ou le Cross-Site Scripting utile) », GNU/Linux Magazine n°105, mai 2008.

BRAINFUCK, UNE MACHINE DE TURING

par **Nicolas Patrois** *[Professeur de mathématiques dans l'enseignement secondaire]*

Vous avez aimé les machines de Mealy [1] et de Turing [2] ? Vous voulez passer à la pratique ? Vous aimerez le langage Brainfuck.

Comme son nom l'indique, ce langage fait des nœuds au cerveau, mais il oblige à un minimum d'organisation si on veut écrire un programme même tout simple. Pour quoi faire ? Vous pouvez répéter la question ? Bon, d'accord : pour que Bolosse 31 puisse parler avec Toto, son petit robot iDiot qui ne comprend que ce langage. Pas de chance, Toto ne comprend pas le Python ni même le C, mais vous pouvez aider Bolosse 31 et simuler son comportement histoire de ne pas l'envoyer dans les limbes d'une boucle infinie.

1 Présentation du langage

Brainfuck (ou BF) est un langage de programmation minimaliste qui contient exactement huit instructions. Nous allons utiliser un dialecte qui en comprend quatre complémentaires, deux pour manipuler les nombres entiers et deux pour déboguer. BF est un langage Turing-complet, ce qui veut dire que théoriquement, vous pouvez coder ce que vous voulez avec. En pratique, c'est une tout autre affaire.

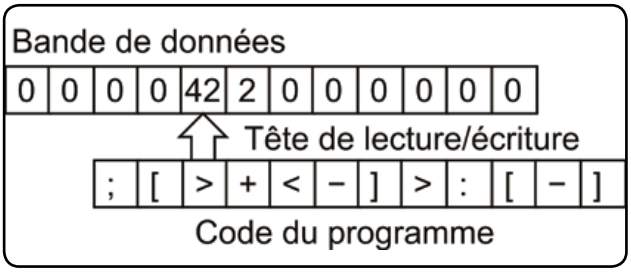


Fig. 1 : Représentation de BF. Toto est la flèche, la figure ne présente pas l'entrée et la sortie standards.

1.1. Le cadre

Brainfuck est une variété de machine de Turing simplifiée constituée :

- d'un ruban de cellules fini, mais non nécessairement borné (des deux côtés dans cet article) sur lequel se déplace une tête de lecture/écriture. Les cellules contiennent des nombres entiers, ici relatifs et sans limites a priori ;
- d'un programme, le code BF exécuté pas à pas ;
- d'une entrée et d'une sortie standards.

Certains dialectes de BF comportent en plus une pile (et deux instructions pour empiler et dépiler) ou quelques autres raffinements, parfois des interdictions qui lui enlèvent son caractère Turing-complet. Le langage n'est pas standardisé même s'il existe un paquet pour Debian (`hsbrainfuck`) ou un autre pour insérer du code BF dans Perl (`libacme-brainfck-perl`).

1.2. Les instructions

Les huit premières instructions sont celles de BF « standard », les deux suivantes viennent du dialecte BF++ [3] et les deux dernières de la littérature. Oui, Toto ne sait faire que ça.

Pour les exemples, on se basera sur le schéma de la figure 1 :

- `,` (virgule) : Insère le code ASCII de l'entrée standard dans la cellule courante.

- **.** (point) : L'inverse du précédent, affiche donc « * » (42 en ASCII correspond au caractère *).
- **+** : Ajoute 1 à la cellule courante (qui deviendra 43). Toto dépose un petit clou sur la cellule.
- **–** : L'inverse (41, donc).
- **>** : Déplace le pointeur d'une cellule vers la droite, donc vers la cellule qui contient 2. Toto avance d'une case.
- **<** : L'inverse, vers le 0 immédiatement à gauche.
- **[** : Si la cellule courante est nulle, va juste après le **]** correspondant, sinon passe à l'instruction suivante. Toto déroule son rouleau d'instructions.
- **]** : Retourne au **[** correspondant. Les boucles sont construites avec ces deux seules instructions.
- **;** (point virgule) : Insère la valeur du nombre (entre deux espaces) de l'entrée standard dans la cellule courante en écrasant la valeur présente.
- **:** (deux points) : L'inverse, affiche « 42 ». Toto répond à LA question. Formidable !
- **!** (point d'exclamation) : Affiche l'état de la bande et la position du pointeur par des crochets, utile pour déboguer.
- **#** : Commentaire, ce qui suit n'est pas interprété jusqu'à la fin de la ligne.

Les deux commandes **;** et **:** permettent de coder plus rapidement des programmes d'arithmétique élémentaire et de s'initier aux subtilités du langage sans devoir comprendre comment utiliser la table ASCII pour traduire une entrée standard numérique en nombres. Nous verrons comment, finalement, nous en passer.

Notes

Si vous vous êtes intéressé aussi aux machines de Turing, vous avez dû remarquer qu'il est facile de transformer un code en BF en table des transitions alors que l'inverse est loin d'être évident. On peut traduire les trois états d'une machine de Turing (blanc, 0 et 1) en BF en 0, 1 et 2 ou −1, 0 et 1.

2 L'interpréteur en Python

Le code est un énorme *switch*, plus quelques traitements préliminaires qui vérifient l'emboîtement des crochets ou suppriment les commentaires éventuels.

Le programme en Python utilisera en entrée deux fichiers : le premier sera le code en BF et le suivant, les données éventuelles de l'entrée standard.

2.1. Les vérifications préliminaires

On commence par vérifier si un code en BF est bien envoyé en entrée :

```
001: #!/usr/bin/python3
002: # -*- coding: utf-8 -*-

004: from sys import exit,argv

006: try:
007:   codebf=argv[1]
008: except IndexError:
009:   print("Usage: brainfuck.py code.bf [donnees.txt]")
010:   exit(0)
```

On continue par une petite analyse syntaxique du code : les crochets doivent être correctement emboîtés et on teste la présence de commandes nécessitant un fichier de données ligne 26. On supprime aussi tous les caractères inutiles, que ce soit ce qui est écrit après un **#** ligne 23 ou les caractères qui ne sont pas des instructions. Les instructions sont stockées dans **code**.

```
012: code=""
013: entree=False
014: commandes={",",".","+","-",">","<","[","]",";",":","#","!"}
015: sep={"\n","\t"," "}
016: nbo=0

018: with open(codebf,"r") as bf:
019:   for l in bf:
020:     for c in l.strip():
021:       if c not in commandes:
022:         continue
023:       if c=="#":
024:         break
025:       code=code+c
026:       if c in {",",";"}:
027:         entree=True
028:       if c=="[":
029:         nbo+=1
030:       elif c=="]":
031:         nbo-=1
032:         if nbo<0:
033:           print("Crochets incorrects.")
034:           exit(0)

036: if nbo!=0:
037:   print("Crochets incorrects.")
038:   exit(0)
```

S'il y a plus de crochets fermants qu'ouvrants en cours de route ligne 32 ou l'inverse à la fin du traitement ligne 36, on arrête tout.

Si le **code** nécessite des entrées, on les stocke dans **entrees** s'il y en a, sinon on s'arrête :

```
040: if entree:
041:   try:
042:     entree=argv[2]
043:   except IndexError:
044:     print("Votre code nécessite un fichier de données.")
045:     exit(0)
046:   with open(entree,"r") as en:
047:     entrees=[]
048:     for l in en:
049:       entrees=entrees+list(l.strip())
```

2.2. Toto fait ce qu'on lui demande

Le **code** est maintenant nettoyé et prêt à être exécuté par Toto, il reste donc à écrire le *switch* géant.

dp est la position de Toto sur la **memoire**. Si Toto cherche à dépasser les bornes du ruban à droite ligne 63 ou à gauche ligne 68, on ajoute un 0 où il faut : on ne bloque pas l'exécution du code, on crée *ad hoc* la **memoire** nécessaire.

p est la position de l'instruction courante sur son rouleau. Noter que l'instruction virgule ligne 81 ne prend qu'un caractère de l'entrée alors que le point virgule ligne 87 vide les séparateurs éventuels puis construit le nombre jusqu'au séparateur suivant avec l'algorithme de Horner qu'on reverra plus loin.

J'ai ajouté le compteur **c** pour ne pas planter Toto dans une boucle infinie.

```
051: p=0
052: dp=0
053: c=0
054: memoire=[0]

056: while p<len(code):
057:   c+=1
058:   if c>=10**6:
059:     print("Limite max. Votre code semble tourner en rond.")
060:     break
061:   elif code[p]==">":
062:     dp+=1
063:     if dp==len(memoire):
064:       memoire.append(0)
065:     p+=1
066:   elif code[p]=="<":
067:     dp-=1
068:     if dp==-1:
069:       memoire=[0]+memoire
070:       dp=0
071:     p+=1
072:   elif code[p]=="+":
073:     memoire[dp]+=1
074:     p+=1
075:   elif code[p]=="-":
076:     memoire[dp]-=1
077:     p+=1
078:   elif code[p]==".":
079:     print(chr(memoire[dp]),end="")
080:     p+=1
081:   elif code[p]==",":
082:     memoire[dp]=ord(entrees.pop(0))
083:     p+=1
084:   elif code[p]==":":
085:     print(memoire[dp],end=" ")
086:     p+=1
087:   elif code[p]==";":
088:     while entrees[0] in sep:
089:       entrees.pop(0)
090:     while entrees and entrees[0] not in sep:
091:       try:
092:         memoire[dp]*=10
093:         memoire[dp]+=int(entrees.pop(0))
094:       except ValueError:
095:         print("Entrée non numérique.")
096:         exit(0)
097:     p+=1
098:   elif code[p]=="[":
099:     if memoire[dp]!=0:
100:       p+=1
101:       continue
102:     nbo=1
103:     while nbo!=0:
104:       p+=1
105:       if code[p]=="[":
106:         nbo+=1
107:       if code[p]=="]":
108:         nbo-=1
109:     p+=1
110:   elif code[p]=="]":
111:     if memoire[dp]==0:
112:       p+=1
113:       continue
114:     nbf=1
115:     while nbf!=0:
116:       p-=1
117:       if code[p]=="]":
118:         nbf+=1
119:       if code[p]=="[":
120:         nbf-=1
121:   elif code[p]=="!":
122:     for i in range(len(memoire)):
123:       if i==dp:
124:         print("["+str(memoire[i])+"]",end=" ")
125:       else:
126:         print(memoire[i],end=" ")
127:     print()
128:     p+=1

130: print()
```

Le code en Python est du type âne qui trotte, mais même Toto ne le comprend pas. Essayons néanmoins de communiquer avec lui.

3 Quelques exemples de codes en BF

L'interpréteur est en Python, on utilise donc son arithmétique, ce qui veut dire que les entiers sont relatifs et sans limites *a priori*. Attention, BF n'a aucun moyen de savoir si un nombre sur le ruban est positif ou négatif, c'est à vous de prévenir Toto.

Les codes en BF seront expliqués par des commentaires au bout de chaque ligne.

3.1. Arithmétique élémentaire

```
+++:
```

Ce code écrira le nombre 3 en sortie.

```
++++>++++++[<+>-]<:
```

Ce code effectue 4+6=10. En détail :

- dans la première cellule, on part de 0 et on effectue **++++** donc on stocke la valeur 4 ;
- **>** : on se déplace d'une cellule vers la droite ;
- **++++++** : on stocke le nombre 6. On a donc la cellule précédente qui contient 4 et la suivante (cellule courante) qui contient 6 ;
- **[** : début de boucle qui s'arrête si la cellule est nulle, ce qui n'est pas le cas pour l'instant. La boucle effectue :
  - **<** : retour à la case précédente et ajout de 1 (**+**). Donc les cellules valent 5 et 6 ;
  - **>** : passage à la cellule suivante et on soustrait 1 (**-**). Les cellules valent 5 et 5 ;
  - **]** : retour au début de la boucle.
- **<** : retour sur la première cellule qui contient le résultat ;
- **:** : affiche le contenu de la cellule.

On vide la cellule de droite petit à petit, regardez la sortie de ce code, c'est le même avec un point d'exclamation qui affiche l'état du ruban et la position de Toto :

```
++++>++++++[!<+>-]<:
```

```
> ./brainfuck.py code.bf entree.txt
4 [6]
5 [5]
6 [4]
7 [3]
8 [2]
9 [1]
10
```

Et voici comment multiplier 6 par 7 :

```
++++++[>+++++++<-]>:
```

C'est presque le même code, mais on ajoute 7 à la deuxième cellule chaque fois qu'on décrémente la première. Avec le point d'exclamation pour afficher l'état de la bande, on a :

```
++++++[!>+++++++<-]>:
```

```
./brainfuck.py code.bf
[6]
[5] 7
[4] 14
[3] 21
[2] 28
[1] 35
42
```

Toto donne LA réponse, sans tricher et en moins d'un million d'années.

Mais comment multiplier deux nombres qui viennent de l'entrée standard ? Commençons par les additionner :

Si l'entrée contient deux nombres, ce code les ajoute et affiche leur somme.

```
;>;[<+>-]<:
```

C'est le même code que pour ajouter sauf qu'on ajoute deux nombres de l'entrée standard et non deux nombres codés en dur. Pas de chance, le code précédent de la multiplication ne fonctionnera pas à cause du multiplicande dans la boucle.

Pour multiplier, on a besoin de planifier l'occupation du ruban pour Toto, c'est malheureux, mais il ne le fera pas pour nous ni même pour son maître, Bolosse 31. Voici :

Premier facteur	Stockage du deuxième	Copie temporaire du deuxième	Produit

Nous allons utiliser la technique de la duplication :

```
++++[>+>+<<-]!
```

Ce code déplace deux fois la première cellule dans les deux suivantes :

```
> ./brainfuck.py code.bf
[0] 4 4
```

```
;>;<          # Stockage des deux facteurs
[
  >[>+>+<<-]  # Duplication de la cellule 2 vers les n°3 et 4
  >[<+>-]     # Déplacement de la sauvegarde qui est temporaire
vers la cellule 2
  <<-         # On décrémente la cellule 1
]             # Et on recommence tant qu'elle n'est pas nulle
>[-]          # On vide la cellule 2
>>:           # On affiche le produit
```

Il faut veiller à anticiper la position de Toto sur le ruban pour qu'il arrive où il faut avant le crochet fermant. En gros, entre une paire de crochets, sauf exception, on veillera à avoir autant de > que de < et surtout, Toto doit arriver à la cellule qu'on teste avant le crochet fermant. Il faut absolument contrôler à tout moment la position de Toto, sinon il va tourner en rond ou partir dans le décor. Usez et abusez de la commande !

Regardez ce que fait ce code :

```
+[>+]
```

Toto avance, puis dépose un petit clou sur la cellule courante. Elle n'est donc pas nulle, il recommence et ne s'arrête jamais. Vous avez intérêt à prévoir un ruban jusqu'à la Galaxie d'Andromède voire jusqu'à Zotra plus une réserve de clous qui entre dans le TARDIS.

La division est plus complexe, vous commencez à deviner pourquoi. On le verra plus loin.

3.2. Bonjour tout le monde

Comment faire parler Toto, par exemple lui faire dire « Bonjour ! » ? Il suffit de lire une table ASCII et de construire les nombres qui vont bien. Cela dit, sans l'astuce précédente pour multiplier, c'est vite infernal.

Les codes ASCII sont, respectivement 66, 111,110,106,110,117, 114, 32 et 33. Nous allons donc construire 32=4×8, 64=2×32 et 96=3×32, puis décliner le reste en fonction de ces trois-là.

```
++++[>++++++++<-]>      # Construit 32
[>+>++>+++<<<-]         # Construit 32, 64 et 96
>>++.--                 # Affiche " B " et rétablit 64
>+++++++++++++++.-.     # Va à 96 et affiche " on "
----.+++++.++++++.      # Affiche " jou "
---.-----------------   # Affiche " r " et rétablit 96
<<.+.-                  # Va à 32, affiche l'espace et le point
d'exclamation, rétablit 32
```

C'est plus simple avec une ou deux astuces, non ?

3.3. Les carrés

On peut utiliser l'idée du code erroné de la fin de la section 3.1. pour faire recopier le même nombre autant de fois que l'entrée standard le dit. Observez ce code sachant que l'entrée est 7 :

```
;[[>]+[<]>-!]>!
```

```
> ./brainfuck.py code.bf entree.txt
0 [6] 1
0 [5] 1 1
0 [4] 1 1 1
0 [3] 1 1 1 1
0 [2] 1 1 1 1 1
0 [1] 1 1 1 1 1 1
0 [0] 1 1 1 1 1 1 1
0 0 [1] 1 1 1 1 1 1
```

La cellule n°2 est en fait la cellule qui contient initialement 7. **[>]** oblige Toto à aller à la première cellule nulle tout à droite. La suite lui ajoute 1, retourne juste une cellule à gauche de là où il y a le 7, revient sur le 7 et décrémente. On recommence jusqu'à ce que la cellule 2 soit nulle.

Avec ceci et l'identité remarquable $(n+1)^2=n^2+(2n+1)$, on peut construire la suite des carrés par récurrence.

```
;[[>]+[<]>-]>:  # Construction du tableau de jeu, placement des 1
>[++>]<[<]>>    # Suite de la préparation, on ajoute 2 à tous les
1 sauf le premier
[
  <[>+<-]       # On ajoute le carré à gauche au nombre suivant
  >>[++>]       # On ajoute 2 à tous les nombres à sa droite.
  <[<]>:>       # On se place sur le carré à afficher, on
l'affiche
]               # Et on rembobine
```

Voici ce que ça donne si on remplace les deux points par un point d'exclamation :

```
> ./brainfuck.py code.bf entree.txt
0 0 [1] 1 1 1 1 1
0 0 0 [4] 5 5 5 5 0
0 0 0 0 [9] 7 7 7 0
0 0 0 0 0 [16] 9 9 0
0 0 0 0 0 0 [25] 11 0
0 0 0 0 0 0 0 [36] 13 0
0 0 0 0 0 0 0 0 [49] 0
```

On pourrait sûrement factoriser le code (on a deux fois **>[++>]<[<]>**), mais le code fonctionne et Bolosse 31 est déjà bien content de s'être fait comprendre par Toto.

3.4. Factorielle

La factorielle de 7 est 7!=1×2×3×4×5×6×7=5040.

Reprenons l'astuce précédente pour écrire la suite décroissante des entiers de 5 à 1 (si l'entrée vaut 5) :

```
;[>[+>]+[<]>-!]>!
```

```
> ./brainfuck.py code.bf entree.txt
0 [4] 1
0 [3] 2 1
0 [2] 3 2 1
0 [1] 4 3 2 1
0 [0] 5 4 3 2 1
0 0 [5] 4 3 2 1
```

Nous allons utiliser deux cellules à droite pour le stockage temporaire, comme dans la multiplication.

```
;[>[+>]+[<]>-]  # L'initialisation
>[>]<:<        # Placement à la fin et affichage de 1
[              # La boucle principale qui va construire le
produit par la droite
  [            # La boucle qui multiplie les deux nombres à
droite
    >[>+>+<<-]
    >[<+>-]
    <<-
  ]
  >>!
  [-<<<+>>>]   # On déplace la factorielle intermédiaire à
droite du facteur suivant à multiplier
  <<[-]<<      # On supprime un résidu du produit sinon ça
pourrit le produit suivant
]
```

Regardez comme Toto fait de belles choses :

```
> ./brainfuck.py code.bf entree.txt
0 0 7 6 5 4 3 2 [1] 0
0 0 7 6 5 4 3 0 1 0 [2]
0 0 7 6 5 4 0 2 0 [6] 0
0 0 7 6 5 0 6 0 [24] 0 0
0 0 7 6 0 24 0 [120] 0 0 0
0 0 7 0 120 0 [720] 0 0 0 0
0 0 0 720 0 [5040] 0 0 0 0 0
```

Étonnant, non ? Remplacez chaque **!** par un **:** et vous obtenez la suite des factorielles. Remarquez le résidu des deux cellules à gauche de Toto qu'on supprime avec **[-]** à l'avant-dernière ligne.

Notez qu'avec des additions au lieu de multiplications, on peut calculer les nombres triangulaires.

3.5. La suite de Fibonacci

Je rappelle qu'une suite de Fibonacci est une suite linéaire récurrente d'ordre 2, c'est-à-dire que si les deux premiers termes sont u_0=1 et u_1=3, u_2=u_0+u_1=4 et u_3=u_1+u_2=3+4=7. De manière plus générale, si n est un nombre entier positif, u_{n+2}=u_{n+1}+u_n ; on a besoin des deux premiers termes u0 et u1 de la suite pour calculer les autres de proche en proche, autrement dit, par récurrence.

L'entrée contient le rang du dernier terme à afficher, le premier terme u0 et le deuxième u1 de la suite. Voici le tableau du ruban :

n	Tampon	Tampon	u_n	u_{n+1}

Et le code :

```
;->;>>>;<<<<   # Dans l'ordre, n-1 (sinon on affiche un terme de
trop),
               # u0 puis u1 deux cellules plus loin
[
  >>>>
  [<+<+>>-]<<< # On duplique un+1 entre sa position et un
  [>>>+<<<-]>  # On déplace un
  [>>+<<-]>>!  # On calcule un+2=un+1+un
  <[<<+>>-]    # On déplace un+1 en vue de l'étape suivante
  <<<-         # On décrémente n
]
```

Voici :

```
> cat entree.txt
7 1 3
> ./brainfuck.py fibonacci.bf entree.txt
6 0 0 3 [4]
5 0 0 4 [7]
4 0 0 7 [11]
3 0 0 11 [18]
2 0 0 18 [29]
1 0 0 29 [47]
```

Comme d'habitude, remplacez le point d'exclamation par les deux points pour afficher les termes de la suite.

La longueur de ce code n'est pas optimisée, on peut la ramener à 31 instructions.

3.6. La division

Commençons par la division euclidienne par 3, qui affiche le quotient entier et le reste. Voici la disposition des variables dans la **memoire** :

Dividende	Accumulateur borné par le diviseur	Copie de cet accumulateur	A-t-on soustrait le diviseur (booléen) ?	Quotient

La cellule appelée accumulateur est décrémentée de 1 à chaque fois que le dividende l'est. Dès que l'accumulateur atteint 0, le quotient augmente de 1 et l'accumulateur revient à 3.

```
;>              # Entrée du dividende
++<             # C'est le diviseur-1
[               # Tant que le dividende n'est pas nul
  !>>>+<<       # Le booléen vaut 1 (enlever le point
d'exclamation)
  [>+>[-]<<-]>  # Si l'accumulateur ne vaut pas 0, le booléen
est annulé
                # et l'accumulateur est déplacé à droite
  [<+>-]>       # On déplace la copie de l'accumulateur s'il le
faut
  [<<+++>>>+<-] # Si l'accumulateur est nul, on incrémente le
quotient
                # et on rétablit l'accumulateur à 3
  <<-           # On décrémente l'accumulateur
  <-            # Et le dividende
]!
>>>>:<<<<       # On affiche le quotient
++>             # Le reste vaut diviseur-1-accumulateur=2-
accumulateur dans notre cas
[<->-]<:        # On affiche le reste
```

Voici ce que cela donne :

```
> ./brainfuck.py div.bg entree.txt
[7] 2
[6] 1 0 0
[5] 0 0 0
[4] 2 0 0 1
[3] 1 0 0 1
[2] 0 0 0 1
[1] 2 0 0 2
[0] 1 0 0 2
2 1
```

Le quotient de la division euclidienne de 7 par 3 est 2 et le reste vaut 1 car 7=2×3+1.

Oui, mais si on veut que le diviseur vienne aussi de l'**entree** ? Bolosse 31 sait comment faire : il lui faut une copie de sauvegarde du diviseur (par exemple une cellule à gauche du dividende) qu'il dupliquera à chaque fois qu'apparaissent dans le code des **+++** ou des **++** (auquel cas il soustraira 1 immédiatement). Il n'oubliera pas de rétablir la copie de sauvegarde à sa place, car la duplication la vide. Je vous laisse faire.

3.7. Pour aller plus loin

Autrement dit, comment revenir au BF pur, sans : ni ; ?

Voici, en pseudo-code pythonique, l'algorithme qui remplace les deux points, il s'agit de l'algorithme de Horner pour l'évaluation des polynômes :

```
Tant que le caractère entré n'est pas l'espace :
  Multiplier par 10 la cellule courante
  Lui ajouter le code ASCII du caractère entré-48
```

Le code ASCII des dix chiffres est 48+chiffre.

Voyons ce que ça donne en BF. Il sera déjà bon de sauvegarder quelque part les valeurs de 32 et de 16, pour ne pas saturer le code d'un nombre illisible de **+** successifs. Voici la disposition des cellules :

32	16	Code ACSII du caractère courant	Tampon à sauvegarde	Nombre voulu

Voici le code :

```
++++[>++++<-]       # Calcul de 16
>[<<++>+>-]         # Calcul de 32 et déplacement de 16 (pas besoin de 48)
,                   # On fait entrer le premier caractère
<<[>>>+>+<<<<-]     # On duplique 32
>>>>[<<<<+>>>>-]    # On déplace une des copies de sauvegarde
```

```
<[-<->]>            # On soustrait 32 au caractère pour lancer la boucle
[                   # On peut commencer la boucle
  <[->->+<<]        # On soustrait 16 sur l'ASCII en le déplaçant sur la sauvegarde
  >>[<<+>>-]        # On rétablit 16
  >[<++++++++++>-]  # On multiplie par 10
  <[>+<-]           # On replace le résultat à sa place
  <[>>+<<-]!        # On lui ajoute le chiffre en cours
  ,                 # On insère l'ASCII du chiffre suivant
  <<[>>->+<<<-]     # On soustrait 32 à l'ASCII en le déplaçant sur la sauvegarde
  >>>[<<<+>>>-]<    # On rétablit 32
]>>!                # Et on recommence
```

Voici le résultat :

```
> cat entree.txt
4057 51
> ./brainfuck.py dp.bf entree.txt
32 16 [0] 0 4
32 16 [0] 0 40
32 16 [0] 0 405
32 16 [0] 0 4057
32 16 0 0 [4057]
```

Et pour l'affichage avec le point, c'est l'algorithme réciproque, on a besoin de la division euclidienne par 10.

Voici l'algorithme en pseudo-code :

```
Tant que le nombre à afficher est non nul:
  On stocke le 48+reste de la division euclidienne par 10
  On recommence avec le quotient
On affiche les chiffres à partir du dernier calculé
```

Il peut se terminer par l'affichage d'un espace.

Il s'agit à peu de choses près de l'algorithme de division, mais emboîté dans la boucle de stockage. Vous savez aussi comment stocker un nombre à la fin d'une file de nombres non nuls (ils le seront, car on a ajouté 48). Faites-le vous-même !

Conclusion

Voilà, Bolosse 31 a quelques techniques pour rendre Toto utile. Il aura du mal à lui faire résoudre un sudoku, mais il pourra tout de même lui faire faire quelques tâches élémentaires.

Si vous voulez écrire des programmes en BF plus élaborés, vous pouvez écrire votre propre langage **[4]** constitué de macros ajoutées au BF. Vous accélérerez l'écriture du code en utilisant par exemple :

- **DUP(a,b,c)** qui duplique la cellule **a** sur les cellules **b** et **c**,
- **MUL2(a,b)=DUP(a,b,b)**,
- **PLUS(a,b)** qui déplace la cellule **a** sur la cellule **b** (si **b=0**), sinon qui ajoute **a** sur **b**,
- **MUL(a,b,c,d)** qui multiplie les cellules **a** et **b**, stocke le résultat en **c** et utilise comme tampon la cellule **d**.

Et ainsi de suite. Le code ainsi créé sera probablement moins joli qu'un autre écrit à la main, préparé aux petits oignons. Attention, les lettres dans les macros désignent des coordonnées relatives à la position de Toto au début de la macro.

Voici quelques algorithmes en BF que j'ai vu passer sans les avoir codés moi-même (je ne suis pas fou à ce point) : la factorisation en produit de facteurs premiers d'un entier, l'algorithme de Syracuse, des quines (code qui écrit son propre code sur la sortie standard), rot13 et même un décodeur de DVDCSS et interpréteur de BF en BF. À quand Nethack ?

Et pour finir, voici un cadeau de saison sur Ideone **[5]**. ■

Références

[1] Prcovic N., « Les machines de Turing », GNU/Linux Magazine n° 174, septembre 2014, p. 24 à 31.

[2] Raynaud M., « Apprenez à programmer avec une machine de Turing ! », Bulletin vert de l'APMEP n°510, septembre-octobre 2014, p. 471 à 484. Voir aussi le site http://www.machinedeturing.org/ où l'auteur présente sa machine physique fonctionnelle.

[3] Page sur BF++ de Code Abbey : http://www.codeabbey.com/index/wiki/brainfuck.

[4] Colombo T., « Créez votre langage de programmation », GNU/Linux Magazine n° 175, octobre 2014, p. 24 à 36.

[5] Programme en Brainfuck : http://ideone.com/nLwtdt. Ne le lancez pas depuis Ideone, mais dans une console. La version commentée est ici : http://ideone.com/L5lJjP.

DÉVELOPPER POUR ANDROID EN GO ?

par ***Tristan Colombo***

Depuis 7 ans, Google développe le langage Go. Devenu libre en 2009, ce langage est utilisé en interne pour de nombreux projets... or Android est bien un projet de chez Google, donc pourquoi ne pas développer d'applications pour Android en Go ?

Nous vous avons déjà parlé du langage Go en décembre 2012 **[1]**, pour la sortie de la première version stable (datant de mars 2012). Déjà à l'époque je m'étais questionné sur la stratégie de Google : pourquoi dépenser du temps, de l'énergie (et de l'argent) pour développer un énième langage destiné à des développements internes ? Il faut toujours chercher à améliorer, optimiser les choses, mais de là à créer un nouveau langage... J'ai longtemps pensé que le but non avoué de cette conception était de posséder un langage natif de développement spécifique aux plateformes Android. En effet, Microsoft possède le C# pour développer des applications Windows Phone et Apple détient l'Objective-C pour les iPhones. Quid de Google ? À une période où il était très nettement en retard par rapport à ses concurrents (plutôt son concurrent en l'occurrence, Microsoft étant encore plus à la traîne), Google a fait le pari d'un langage connu d'une grande part des développeurs, le Java, pour accélérer le développement d'applications. La migration vers Go était peut-être un objectif, mais le piège s'est sans doute refermé sur Google : les développeurs accepteraient-ils maintenant cette migration ? Bien sûr le support de Java ne serait pas abandonné, il suffirait d'encourager l'utilisation du Go auprès des développeurs, car n'oublions pas que Java n'est pas franchement un modèle de rapidité alors qu'avec le Go on a une compilation efficace, une exécution tout aussi efficace et un développement simple. De plus, il est possible d'utiliser du code Go à l'intérieur d'un code Java.

La question de la migration est peut-être encore d'actualité... toujours est-il que j'ai voulu en vérifier la faisabilité : peut-on développer et exécuter sur une plateforme Android une application codée en Go ? Je vous propose donc d'installer le compilateur Go et d'essayer de compiler et d'exécuter un code Go sur Android.

1 Installation du langage Go

Il existe deux compilateurs Go : le compilateur standard nommé **gc** et le compilateur GNU **gccgo** qui est une surcouche au compilateur **gcc**. Il est possible d'installer ces compilateurs depuis les dépôts ou depuis les sources. Comme l'objectif de cet article n'est pas de détailler les possibilités du Go, je ne décrirai qu'une seule méthode avec l'installation de **gc** depuis les sources, ce qui permettra de travailler avec la version la plus récente.

Tout d'abord, certains outils sont indispensables pour la compilation des sources. Normalement, ils doivent déjà être installés sur votre système, mais au cas où, installons-les. La méthode sera donnée pour des distributions basées sur Debian :

```
# aptitude install gcc libc6-dev
```

Ensuite, pour récupérer les sources nous aurons besoin de Mercurial :

```
# aptitude install Mercurial
```

Enfin, comme Go utilise des variables d'environnement qui permettent notamment de savoir dans quel répertoire se trouvent les sources du langage, nous allons commencer par les définir de manière à se servir de cette information lors de la récupération des sources. Les variables d'environnement utilisées seront les suivantes :

- **GOROOT** : répertoire contenant les sources du langage ;
- **GOOS** : nom du système d'exploitation cible ;
- **GOARCH** : architecture du système cible ;
- **GOBIN** : répertoire contenant les outils (fichiers exécutables) du projet Go ou écrits en Go ;
- **GOPATH** : répertoire contenant des codes sources en Go et leurs binaires associés.

La définition de ces variables se fait dans le fichier de configuration de votre shell. Pour le shell bash, ce sera **~/.bashrc** :

```
# Variable d'environnement pour Go
export GOROOT=/opt/go
export GOOS=linux
export GOARCH=amd64
export GOPATH=~/Mes_tests_en_Go
export PATH=$PATH:$GOPATH/bin:$GOROOT/bin
```

Ici nous supposons (pour le moment) que nos programmes seront compilés pour une utilisation sur linux avec un processeur 64 bits, que les sources du langage seront stockées dans **/opt/go**, et que nos fichiers de test en Go seront dans **~/Mes_tests_en_Go**. N'oubliez surtout pas d'activer les modifications du fichier **.bashrc** dans le terminal courant avec :

```
$ source ~/.bashrc
```

Nous pouvons maintenant récupérer le code source :

```
# hg clone -u release https://code.google.com/p/go $GOROOT
```

Pour lancer la compilation, il faut se déplacer dans le répertoire **$GOROOT/src** et exécuter **all.bash** :

```
# cd $GOROOT/src
# ./all.bash
```

Après quelques minutes vous obtiendrez l'affichage du message :

```
ALL TESTS PASSED

---
Installed Go for linux/amd64 in /opt/go
Installed commands in /opt/go/bin
*** You need to add /opt/go/bin to your PATH.
```

Comme nous avons déjà ajouté **$GOBIN** à notre variable **$PATH**, ne tenez pas compte de la remarque de la dernière ligne.

Si vous souhaitez que tous les utilisateurs aient accès au répertoire **/opt/go** et puissent utiliser le compilateur :

```
# chmod -R ugo+rx $GOROOT
```

Vérifiez que tout fonctionne correctement en affichant la version de go installée :

```
$ go version
go version go1.3.3 linux/amd64
```

Tout marche bien, nous pouvons maintenant écrire un petit programme.

2 Un code de test

Écrivons un petit code qui affichera un message à l'écran. Placez-vous dans le répertoire **$GOPATH** et créez un fichier **display.go** :

```
01: package main
02:
03: import "fmt"
04:
05: func main() {
06:     fmt.Println("GNU/Linux Magazine")
07:     fmt.Println("Du Go dans Android ?")
08: }
```

Faites bien attention à la place de vos accolades : seule la notation Kernighan et Ritchie est autorisée. Par exemple, si vous placez l'accolade de la fin de la ligne 5 sur une nouvelle ligne, vous obtiendrez le message :

```
# command-line-arguments
./display.go:6: syntax error: unexpected semicolon or newline
before {
./display.go:8: non-declaration statement outside function body
./display.go:9: syntax error: unexpected }
```

Oui, mais pour obtenir un message, encore faut-il pouvoir compiler et exécuter le programme... Pour cela vous pourrez lancer la commande suivante :

```
$ go run display.go
```

Cette commande est utile pour les tests, mais vous ne conservez pas la version compilée du programme. Pour cela, il faut utiliser la commande :

```
$ go build display.go
```

Un nouveau fichier exécutable **display** apparaîtra alors dans le répertoire courant.

Enfin, bien que cela sorte du cadre de cet article, si vous souhaitez installer le binaire de l'un de vos programmes dans **$GOBIN** pour qu'il puisse être exécuté par les autres utilisateurs, ce sera :

```
$ go install display.go
```

Notre code fonctionne sous Linux... c'est bien ! Mais il va maintenant falloir le faire tourner sous Android...

3 Exécution sous Android

Pour que notre code puisse être exécuté sous Android, il va falloir modifier la plateforme cible dans les variables d'environnement. C'est là que l'aspect de cross-compilation de Go va nous rendre de grands services. Android est un système Linux sur architecture ARM, il faudra alors modifier trois variables d'environnement :

```
$ export GOOS=linux
$ export GOARCH=arm
$ export GOARM=7
```

Nous n'avions pas vu la dernière variable qui permet de fixer la version d'architecture ARM. Sur un système ARM, cette variable est allouée automatiquement si l'on compile sur l'architecture cible et sinon elle est fixée à 6. Trois valeurs sont possibles :

- **GOARM=5** : pas de co-processeur VFP ;
- **GOARM=6** : architecture VFPv1 ;
- **GOARM=7** : architecture VFPv3.

Pour les smartphones actuels, il y a de fortes chances pour que **7** soit la bonne valeur (sinon vous aurez toujours le temps de changer la valeur et de recompiler).

Il ne reste plus qu'à compiler notre code pour Android :

```
$ go build display.go
go build runtime: linux/arm must be bootstrapped using make.bash
```

Raté ! Nous avons compilé le langage seulement pour une cible linux/amd64. Pour chaque nouvelle plateforme cible, il faudra effectuer une compilation qui permettra de le prendre en compte :

```
# cd $GOROOT/src
# GOOS=linux GOARCH=arm GOARM=7 ./make.bash --no-clean
```

Maintenant, nous pouvons compiler notre code pour Android sans aucune erreur. Si vous voulez vous éviter les exports de variables d'environnement si vous enchaînez les compilations sur différentes plateformes, vous pouvez utiliser la même écriture que pour la compilation de Go :

```
$  GOOS=linux GOARCH=arm GOARM=7 go build display.go
```

Bien sûr, si tout a fonctionné correctement, vous ne pourrez pas utiliser le fichier exécutable sur votre ordinateur :

```
$ ./display
fatal error: rt_sigaction failure

runtime stack:
runtime.throw(0x116ce5)
        /opt/go/src/pkg/runtime/panic.c:520 +0x5c
...

goroutine 16 [runnable]:
runtime.main()
        /opt/go/src/pkg/runtime/proc.c:207
runtime.goexit()
        /opt/go/src/pkg/runtime/proc.c:1445
```

Donc nous sommes pour l'instant potentiellement en possession d'un binaire que l'on pourrait lancer sous Android... Eh bien plaçons le code dans un smartphone et exécutons-le ! Pour réaliser ces opérations nous allons utiliser un shell **adb**. Pour commencer, il faut placer le fichier exécutable sur le smartphone (dans un répertoire qui ne risque rien) :

```
$ adb push display /data/local/tmp/display
```

N'oubliez pas de confirmer le transfert si vous ne voulez pas obtenir le message suivant :

```
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
error: device unauthorized. Please check the confirmation dialog
on your device.
```

Si tout se déroule correctement vous obtiendrez un message similaire au message suivant :

```
4615 KB/s (1478912 bytes in 0.312s)
```

On se connecte ensuite en shell adb sur le smartphone :

```
$ adb shell
```

Il reste à se déplacer dans le répertoire **/data/local/tmp**, à donner les droits d'exécution sur notre programme et à l'exécuter :

```
shell@GT-N7000:/ $ cd /data/local/tmp
shell@GT-N7000:/data/local/tmp $ chmod 755 display
shell@GT-N7000:/data/local/tmp $ ./display
GNU/Linux Magazine
Du Go dans Android ?
```

Le message qui s'affiche à l'écran est le résultat de l'exécution du code sur le smartphone. Donc oui, on peut coder en Go pour Android... maintenant de là à pouvoir développer une application, il reste encore du travail !

À quoi peut donc servir cette technique ? Comme je l'ai dit précédemment, à effectuer des traitements lourds et « alléger » le code Java. Pour cela, il faut placer l'exécutable dans le répertoire **libs/armeabi** de votre projet Android en le renommant en **lib<nom>.so**. Dans le cas de notre programme **display**, ce serait donc **libs/armeabi/libdisplay.so**. Avec ce mécanisme la librairie Go sera ajoutée à votre apk et vous pourrez l'utiliser dans le code de votre projet Android avec, par exemple :

```
Process disp = Runtime.getRuntime().exec(getApplicationInfo().
nativeLibraryDir() + "/libdisplay.so") ;
```

Conclusion

Dans cet article, nous avons pu tester la possibilité d'exécuter du code Go sur une plateforme Android et même de l'intégrer à un projet Android en Java. Il s'agissait bien sûr uniquement d'un test, mais suivant les développements il peut être intéressant de garder en mémoire que cette option existe et permet d'accélérer certains traitements. ■

Référence

[1] Armand J.-M. et Colombo T., « Apprenez à programmer en Go ! », GNU/Linux Magazine HS n°63, novembre/décembre 2012.

LAISSEZ SOUFFLER UNE PETITE BRISE DANS L'UNIVERS DE PHP AVEC ZEPHIR

par ***Stéphane Mourey*** *[Taohacker]*

On ne cesse de le répéter, de le constater, de s'en plaindre : PHP est lourd, lent, à se demander si l'éléphant choisi pour être son logo ne l'a pas été pour cela. Diverses initiatives ont été prises pour y remédier, voici l'une des plus prometteuses : Zephir.

Zephir est un véritable langage de programmation, mais dédié à une seule tâche : la réalisation d'extensions à PHP. Son intérêt par rapport à un développement en pur PHP est que l'extension, d'abord traduite en C, est ensuite compilée une bonne fois pour toutes en langage machine. Ce choix garantit une vitesse d'exécution optimale à comparer avec celle de la compilation JIT (*Just In Time*), la mise en cache de byte-code ou la simple interprétation. S'il est connu depuis longtemps qu'un algorithme développé en C sera toujours plus rapide que s'il est développé en PHP, le développeur PHP lui, ne se sent pas toujours à l'aise avec le C. Par ailleurs, si PHP a pour une bonne part évolué en intégrant des bibliothèques C existantes, il est connu que de réaliser une extension C à PHP n'est pas une chose aisée.

Le but de Zephir est dès lors de proposer un langage qui paraisse familier aux développeurs PHP, tout en leur offrant les performances du C. Zephir n'est fait que pour cela et ne pourra faire que cela. Les développements réalisés avec lui ne produiront pas autre chose que des extensions à PHP, à charger comme les autres via le `php.ini` (solution à préférer) ou la fonction `dl`, mettant à disposition du programmeur PHP de nouvelles classes. L'extension, une fois chargée par le `php.ini`, restera en mémoire, ce qui évitera d'avoir à charger de multiples fichiers, comme c'est le cas aujourd'hui avec beaucoup de frameworks PHP. Pour les adeptes de Zephir, la méthodologie à privilégier est alors de développer toutes les classes nécessaires à un projet en Zephir, puis de lier tout cela à l'aide de PHP classique, qu'on tentera alors de réduire au minimum.

Outre la syntaxe, qui se veut assez proche de celle de PHP, Zephir permet au développeur PHP d'utiliser les fonctions standard de PHP et des autres extensions chargées, ce qui ouvre immédiatement l'accès à des bibliothèques larges et déjà familières.

Il y a tout de même quelques inconvénients à utiliser Zephir. Tout d'abord, le temps de développement sera un peu plus long qu'avec du pur PHP :

- la compilation est incontournable avant de pouvoir tester le code,
- les messages envoyés par PHP lorsque des erreurs se produisent, situant toujours l'erreur à la ligne 0 ne facilitent pas vraiment le débogage,
- et les erreurs levées lors de la compilation demandent aussi un certain art de l'interprétation.

Mais n'oublions pas que ce projet est encore très jeune et que tout cela devrait s'améliorer lorsqu'il gagnera en maturité. Toutefois, la mise en production, elle, sera plus compliquée que pour un simple script PHP : il faudra copier la nouvelle bibliothèque et redémarrer le serveur Web, ce qui demande souvent les droits d'administrateur sur la machine.

Pour la petite histoire, sachez que Zephir a été développé par l'équipe qui a réalisé Phalcon, le framework pour PHP écrit en C, et que Phalcon 2, lui, a été entièrement réécrit en Zephir.

1 Installation

Zephir n'étant pas encore inclus dans nos dépôts préférés, l'installation par le code source est une étape incontournable.

1.1. Prérequis

Mais avant de se lancer, il convient de vérifier les prérequis, à savoir :

- gcc 4 ou supérieur ;
- re2c 0.13 ou supérieur ;
- gnu make 3.81 ou supérieur ;
- autoconf 2.31 ou supérieur ;
- automake 1.14 ou supérieur ;
- libpcre3 ;
- les outils et les en-têtes de développement de PHP.

Sous Debian et dérivés, avec la commande suivante, les dépendances requises devraient être installées :

```
$ sudo apt-get install git gcc make re2c php5 php5-json php5-dev
libpcre3-dev
```

Vous devez également vous assurer que vous avez une version récente de Zephir (qui, tenez-vous bien, est écrit en PHP !) et PHP doit être accessible depuis la ligne de commandes. Assurez-vous en avec la commande suivante :

```
$ php -v
PHP 5.5.7 (cli) (built: Dec 14 2013 00:44:43)
Copyright (c) 1997-2013 The PHP Group
Zend Engine v2.5.0, Copyright (c) 1998-2013 Zend Technologies
```

Enfin, pour vous assurer que vous avez les bibliothèques de développement de PHP, utilisez la commande :

```
$ phpize -v
Configuring for:
PHP Api Version:         20121113
Zend Module Api No:      20121212
Zend Extension Api No:   220121212
```

Si tout est bon jusqu'ici, vous pouvez passer à l'étape suivante.

1.2. Dépendance supplémentaire : json-c

Pour cette dépendance, il vous faudra l'installer à partir des sources depuis GitHub. Normalement, l'enchaînement classique de commandes suivant devrait vous permettre d'y parvenir sans problème :

```
$ git clone https://github.com/json-c/json-c.git
$ cd json-c
$ sh autogen.sh
$ ./configure
$ make && sudo make install
```

1.3. Enfin Zephir...

Là aussi, il faut copier les sources depuis GitHub et procéder à l'installation.

```
$ git clone https://github.com/phalcon/zephir
$ cd zephir
$ ./install -c
```

Enfin, testez votre installation à l'aide de la commande ci-dessous, et vous devriez obtenir votre numéro de version de Zephir :

```
$ zephir version
0.5.5
```

Vous n'avez plus qu'à vous lancer dans la création de votre première extension !

2 Hello world !

Comme j'aime les classiques, je ne vous proposerai rien d'original pour commencer, nous allons faire un classique « Hello world ». Mais si ce que nous faisons n'est pas original, cela nous permettra tout de même de découvrir les bases de Zephir.

2.1. Création du squelette

Zephir ne vous abandonne pas perdu dans la nature : une aide vous est proposée pour mettre en place les premiers éléments attendus pour la réalisation de votre extension. Ainsi, une fois que vous vous êtes placé dans votre répertoire de travail, vous lancez la commande :

```
$ zephir init hello
```

Voilà, vous avez créé le squelette de votre extension **hello**. Un nouveau dossier **hello** est apparu, contenant un fichier **config.json** et deux dossiers **ext** et **hello**. Le fichier **config.json** contient des directives de configuration : a priori, vous ne devriez pas en avoir besoin, sauf à indiquer un auteur

et une description. **ext** est le dossier de travail utilisé par Zephir pour réaliser la traduction en C puis la compilation de votre extension. Là non plus, vous ne devriez pas avoir besoin de regarder, sauf à vouloir déboguer Zephir... Enfin **hello** est le dossier dans lequel vous allez écrire votre code Zephir proprement dit. Et devinez quoi ? Il ne contient encore rien.

2.2. Création de la première classe

Par principe, Zephir n'accepte pas la création de code simplement procédural : il est impératif de passer par l'écriture d'une classe, et Zephir ne vous permettra pas de sortir de là : tout votre code devra être objet, et placé dans un espace de noms ayant pour nom celui de votre extension, mais avec une majuscule initiale.

Le code Zephir est contenu dans des fichiers d'extension **.zep**. Écrivons donc notre première classe **World**, à placer dans le fichier **world.zep** :

```
namespace Hello;

class World
{
  public function say()
  {
    echo "Hello world!\n" ;
  }
}
```

Ensuite, vous pouvez déjà lancer votre première compilation avec la commande ci-dessous. Attention, cette commande est à lancer dans le dossier racine de votre extension (pour nous **hello**), mais pas dans celui où vous avez écrit votre fichier **zep** (pour nous **hello/hello**) :

```
$ zephir build
Preparing for PHP compilation...
Preparing configuration file...
Compiling...
Installing...
[sudo] password for username:
Extension installed!
Add extension=hello.so to your php.ini
Don't forget to restart your web server
```

Il vous faudra taper votre mot de passe root pour que l'installation puisse avoir lieu. Il faudra également que vous modifiiez votre **php.ini** pour que l'extension soit chargée au démarrage du serveur Web. Zephir vous l'indiquera à chaque installation, mais vous n'aurez naturellement à le faire que la première fois que vous compilez une nouvelle extension. Par contre, la dernière instruction est à suivre pour chaque compilation : l'extension étant chargée seulement au démarrage du serveur, les modifications ne seront prises en compte qu'au redémarrage. Le mieux est sans doute de tester votre code en ligne de commandes : chaque fois que **php -a** est exécuté, le fichier **php.ini** est réanalysé.

Si votre extension est bien installée, vous pouvez le vérifier ainsi :

```
$ php -m | grep hello
hello
```

php -m liste les extensions installées.

À partir de là, il y a une chance que votre code fonctionne depuis PHP :

```
$ php -a
Interactive mode enabled
php > $t = new Hello\World();
php > $t->say();
Hello world!
php >
```

Et voilà !

3 Tour du propriétaire

Bon, maintenant que Zephir est installé, il nous faut encore maîtriser la bête.

Si Zephir est conçu pour ne pas trop dérouter le programmeur PHP, il y a quand même un nombre significatif de différences.

3.1. Les variables

L'une des plus grandes forces de Zephir se situe dans sa gestion des variables, et c'est là-dessus que la réflexion du développeur devra s'approfondir le plus pour bien maîtriser Zephir.

3.1.1. Déclaration

La première chose à savoir est que toutes les variables doivent être déclarées avant de pouvoir être utilisées et cette déclaration doit indiquer leurs types. Voilà qui n'est pas dans l'esprit de PHP, mais la performance est à ce prix là. Qu'on se rassure toutefois, une instruction spéciale permet de déclarer une variable comme ayant un type dynamique, c'est-à-dire qu'il s'agira là d'une variable PHP classique, dont le type sera converti à la volée selon le contexte. Eh oui, tel est l'un des tours de force de Zephir : savoir marier les variables fortement typées comme en C et les variables faiblement typées comme en PHP. Toutefois, cela implique un certain nombre de contraintes différentes pour les unes et les autres dont il faut avoir conscience afin d'éviter certains désagréments. Mais commençons pas examiner la façon de les déclarer.

La déclaration commence par le nom du type de la variable, suivi du nom de la variable. Pour les types statiques, on retrouve les grands classiques (**boolean**, **string**, **integer**, **unsigned integer**, **float**...). Il y a également des types dynamiques, qui sont les mêmes que ceux proposés par PHP, puisqu'il s'agit ni plus ni moins que de variables PHP. Notons en passant que les types dynamiques et les types statiques ne se recouvrent pas exactement : il n'y a pas de types statiques **object** ou **array** par exemple. Pour déclarer une variable de type dynamique, on utilisera le mot-clef **var**.

Voici un exemple :

```
namespace Hello;
class World
{
  public function say()
  {
    string hello,toto; // déclaration des chaînes hello et toto
    int temperature;   // déclaration d'un entier signé
    uint counter;  // déclaration d'un entier non-signé
    var nimp;   // déclaration d'une variable dynamique
[…]
```

Toutes nos variables se trouvent dans une méthode et ne porteront pas au-delà, ce qui est rendu nécessaire par la contrainte que Zephir impose : tout notre code doit être placé dans une classe, les seules variables qui peuvent exister hors des méthodes sont les propriétés, que nous examinerons plus loin.

3.1.2. Assignation

Bon, ce n'est pas le tout de déclarer des variables, il faut encore pouvoir s'en servir et donc leur assigner des valeurs. Il est possible de le faire lors de la déclaration, ce qui permet toujours un petit gain de temps :

```
string hello = "Hello world !";
```

Un petit mot en passant à propos des guillemets : l'écriture d'une chaîne de caractères commence et termine toujours par des guillemets doubles **"** et, contrairement à PHP, Zephir ne supporte pas l'utilisation de guillemets simples **'** pour cet usage. Par ailleurs, aucune substitution de variables n'y sera effectuée.

Maintenant que nous lui avons donné une valeur, il nous reste encore à pouvoir modifier cette variable. L'utilisation du simple signe égal **=** ne suffit pas, contrairement à PHP. Il faut utiliser le mot-clef **let** :

```
let hello = "Hello GMLF !";
```

Cette syntaxe est plus verbeuse que ce à quoi les développeurs PHP ou C sont habitués, puisqu'elle évacue les opérateurs d'assignation (**++**, **--**, **+=**, **-=**, etc.), mais elle présente cet avantage de rendre les changements de valeurs extrêmement lisibles.

Une dernière chose à savoir, mais d'importance à l'usage : en lisant la documentation PHP, vous avez peut-être remarqué que les types des valeurs de retour des fonctions sont parfois indiqués en utilisant le faux type **mixed** pour signifier qu'une fonction peut retourner un type ou un autre. Si vous voulez affecter la valeur de retour de **array_search()** à une variable par exemple, comment savoir si vous devez utiliser un entier ou un booléen ? Zephir a fait pour cela un choix radical : seules les variables de type dynamique (**var**) peuvent se voir assigner une valeur par une fonction, quelle qu'elle soit.

3.3.2. Nom de variables dynamiques

Une des caractéristiques intéressantes de PHP est la possibilité d'utiliser des noms de variables dynamiques, c'est-à-dire d'utiliser le contenu d'une variable comme nom d'une autre :

```
<?php
$b = 'PHP permet ça !';
$a = 'b';
print $$a;
```

affichera

```
PHP permet ça !
```

Zephir ne permet pas un tel fonctionnement, les variables étant compilées en variables de bas niveau, mais il est tout de même possible d'arriver à des résultats similaires en utilisant une variable comme nom d'une autre variable, et même une variable comme nom d'une fonction :

```
namespace Hello;
class World
{
    public function createImg(filename, format)
    {
        string imgCreateFunc;
        switch (format)
        {
            case IMAGETYPE_JPEG:
                let imgCreateFunc = "imageCreateFromJpeg";
namespace Hello;
class World
{
    public function createImg(filename, format)
    {
        string imgCreateFunc;
        switch (format)
        {
            case IMAGETYPE_JPEG:
                let imgCreateFunc = "imageCreateFromJpeg";
                break;
```

```
            case IMAGETYPE_GIF:
                let imgCreateFunc = "imageCreateFromGif";
                break;
            case IMAGETYPE_PNG:
                let imgCreateFunc = "imageCreateFromPng";
                break;
        }
        return {imgCreateFunc}(filename);
    }
}
```

Dans cet exemple, le paramètre **format** est utilisé pour déterminer quelle fonction doit être utilisée pour créer une image au bon format. Le nom de cette fonction est affecté à une variable **imgCreateFunc**, ensuite utilisée directement pour appeler la fonction correspondante. La syntaxe à accolade permet en fait de contourner la compilation C pour préférer une interprétation par PHP (ne me demandez pas comment le compilateur Zephir fait pour partager son code de cette façon sans s'y perdre...).

Il faut tout de même bien comprendre que les pratiques de ce genre, si elles permettent de résoudre facilement certains problèmes qui sans cela peuvent se révéler épineux, se révèlent contre performantes en ce qui concerne Zephir. En effet, le code ne pourra être compilé en C de telle sorte à ce que l'appel de la fonction puisse se faire directement, mais celui-ci devra transiter par l'interpréteur PHP. De plus, Zephir se livre à certaines optimisations du code, et en particulier, il remplace les appels à des fonctions standards PHP par des appels à leurs équivalents sous-jacents en C, ce qui n'est pas pour rien dans les gains de performance offerts par Zephir : cela est bien entendu impossible si vous utilisez une variable comme nom de fonction.

3.2. Les méthodes

Comme vous l'avez sans doute remarqué dans les exemples de code précédents, l'écriture de méthodes en Zephir est assez similaire à ce que l'on fait en PHP. Il y a toutefois quelques améliorations intéressantes que je n'ai pas fait apparaître jusqu'ici.

3.2.1. Arguments constants

Il est possible d'indiquer dans la déclaration de la fonction que les arguments reçus ne seront pas modifiés par la fonction et ainsi traités par elle comme des constantes. Cela permet au compilateur d'optimiser son fonctionnement.

```
namespace Hello;
class World {
    public function sayHello2(const name) {
        echo "Hello " . name . " !\n";
    }
}
```

Nous voyons ici que le mot-clef **const** est utilisé avant le nom de l'argument pour indiquer que celui-ci ne sera pas modifié.

3.2.2. Arguments typés

Il est également possible d'indiquer un type attendu pour l'argument d'une manière similaire :

```
public function sayHello2(string name) {
```

Ce qui n'empêche pas notre argument d'être constant à l'intérieur de la fonction:

```
public function sayHello2(const string name) {
```

Oui, mais, vu que PHP est constant, comment va se comporter notre fonction si elle reçoit un argument qui n'est pas du bon type ? Avec cette syntaxe, l'argument reçu sera automatiquement converti au bon type à l'entrée de la fonction, à la manière habituelle de PHP. Mais il est possible de refuser un argument s'il n'est pas du bon type en utilisant une syntaxe légèrement différente :

```
public function sayHello2(!string name) {
```

Dans le cas où l'argument reçu ne serait pas une chaîne de caractères, une erreur se produirait.

3.2.3. Types de retour

Il est également possible d'indiquer le ou les types de la valeur de retour d'une méthode. Pour cela, il suffit de l'indiquer après la déclaration de la méthode suivie de l'opérateur **->** :

```
public function sayHello2(name)  -> string {
```

Toutefois, à l'heure où j'écris ces lignes, des discussions ont lieu pour permettre l'utilisation du double-point **:** à la place de **->**, pour se rapprocher ainsi de la syntaxe adoptée par PHP-NG et Hack, qui offrent la même fonctionnalité. Cela nous donnerait alors :

```
public function sayHello2(name)  : string {
```

Mais ce n'est pas encore fait, et aux dernières informations, il n'était pas question d'abandonner le support de la flèche.

3.3. Les propriétés

Les propriétés des objets Zephir ne diffèrent en rien de celles d'objets PHP. On peut regretter en particulier qu'elles ne bénéficient pas d'un typage fort, il s'agit de variables PHP. Dommage.

Toutefois, disons tout de même quelques mots sur certains raccourcis d'écriture qui permettent de créer rapidement « getters » et « setters » (Dieu que c'est laid en français !). Pour rappel, il s'agit de méthodes que certains développeurs objets doivent écrire très souvent s'ils les ont adoptées et qui permettent d'accéder en lecture ou en écriture à des propriétés d'un objet. Le plus souvent, les propriétés en question sont protégées, ce qui oblige alors à passer par ces méthodes lorsqu'on y accède depuis l'extérieur de l'objet. Cela permet ainsi de contrôler l'accès à ces propriétés, et d'en interdire une utilisation inadéquate. Toutefois, la plupart du temps, il s'agit simplement d'une possibilité qu'on se laisse pour l'avenir, ce qui fait que certains utilisent ces méthodes systématiquement, pour le cas où.

Voici un exemple en PHP :

```
class Hello {
    protected $text;

    public function getText() {
        return $this->text;
    }

    public function setText($text) {
        this->text = $text;
    }
}
```

Ici, nous avons une propriété protégée **`$text`** que l'on peut lire grâce à la méthode **`getText`** et affecter par la méthode **`setText`**. Ainsi on constate que l'habitude est de mettre la première lettre du nom de la propriété en majuscule puis d'y ajouter un préfixe correspondant à la méthode à implémenter (**`get`** ou **`set`**). Et, la plupart du temps, ces méthodes sont absolument identiques, seul le nom de la propriété étant à faire varier. On imagine que ces méthodes, si leur utilité peut se justifier, deviennent rapidement pénibles à écrire tant elles sont redondantes. Il existe heureusement des alternatives (en utilisant les méthodes magiques de PHP par exemple) et Zephir nous propose la sienne :

```
namespace Hello;
class World {
     protected text {
        get, set
    };
}
```

Juste pour que la syntaxe soit totalement éclaircie, donnons le même exemple, mais en attribuant une valeur initiale à la propriété lors de sa déclaration :

```
namespace Hello;
class World {
    protected text = "World" {
        get, set
    };
}
```

3.4. Cerise sur le gâteau

Pour le reste, Zephir est assez classique dans son fonctionnement, et comme pour tout langage, il faut prendre le temps de lire la documentation pour l'appréhender entièrement. Un point au chapitre des opérateurs m'a tout de même surpris et je ne peux m'empêcher de vous l'indiquer : il s'agit des indications de prédiction de branches. Par branches ici, il faut comprendre les deux membres d'un code conditionnel :

```
if (maCondition()) {
    // branche si vraie
} else {
    // branche si faux
}
```

Il arrive souvent d'écrire un test pour vérifier qu'une opération se déroule bien, ce qu'elle fait presque toujours, mais il faut bien éviter un plantage pour les quelques cas où cela ne se passera pas comme prévu :

```
if (presqueToujoursVraie()) {
    //branche si vraie
} else {
    //branche si faux
    throw new Exception("Planté !");
}
```

Le problème est que, de nos jours, nos processeurs sont tellement performants qu'ils essaient d'anticiper le travail qu'ils auront à faire quand ils connaîtront la valeur de la condition pour exécuter le code qui leur paraît le plus probable à l'avance (rien qu'à l'écrire comme ça, je doute de la santé mentale de celui qui a réussi à implémenter cette idée dans un circuit intégré...). Pour y parvenir, certains algorithmes statistiques de prédiction de branches sont mis en jeu, mais il arrive qu'ils se trompent, et dans ce cas, le processeur perd un peu de temps à défaire ce qu'il a fait...

Bref, si on pouvait indiquer au processeur qu'une branche a peu de chances de se réaliser, on pourrait lui faire gagner du temps.

Pour ma part, j'ai toujours eu tendance dans ces cas-là à ne conditionner que la branche improbable : je ne sais pas si le processeur va pour autant se priver de l'exécuter pour se faire le plaisir d'être en avance dans une réalité parallèle qui ne devrait pas se produire, mais cela rend le code moins complexe, plus lisible, et le code dont j'attends réellement l'exécution ne paraît pas, ni pour moi, ni pour mes collaborateurs, ni, j'espère, pour le processeur, conditionné :

```
if (!presqueToujoursVraie()) {
    //branche si échec
    throw new Exception("Planté !");
}
// branche normale
```

D'autres allègements de syntaxe sont encore possibles (suppression des accolades), ce qui rend le code plus lisible encore. Mais, revenons à nos moutons, nous ne sommes pas en train d'écrire un traité sur le beau style et voyons ce que Zephir a à nous proposer sur cette question des branches.

J'ai écrit plus haut qu'il serait bon de pouvoir indiquer au processeur qu'une condition a peu de chances de se réaliser. C'est exactement ce que Zephir permet de faire à l'aide du mot clef **unlikely** :

```
if unlikely presqueToujoursVraie()==FALSE {
    //branche si échec
    throw new Exception("Planté !");
} else {
    // branche normale
}
```

Quand on vous dit que Zephir a été écrit avec une réelle recherche de performances ! Notons en passant qu'il n'est pas nécessaire de placer l'expression à évaluer entre parenthèses lors d'un test, même si cela est possible.

Conclusion

Voilà, j'espère vous avoir donné envie de vous lancer à corps perdu dans la programmation d'extensions à PHP écrites en Zephir. Les esprits chagrins argueront que ce choix risque de réduire le nombre d'utilisateurs potentiels de leurs applications : les privilèges requis et les compétences techniques sont en effet un peu plus élevés que pour une habituelle application PHP.

Pour ceux-là, j'ai une surprise, car les développeurs de Zephir pensent à eux : Zephir pourra aussi compiler, traduire serait plus juste, son code en simple PHP. Cette fonctionnalité n'est pas encore implémentée, et ne le sera pas pour la version 1. La base actuelle d'utilisateurs n'en a pas besoin, et la communauté préfère se concentrer sur ce qui est aujourd'hui le cœur de Zephir avant d'ouvrir une nouvelle voie. Mais cette idée est tout de même considérée comme centrale, et elle sera à l'agenda pour la version suivante. Et on discute de choses encore plus folles sur l'espace GitHub dédié au projet (https://github.com/phalcon/zephir)... ■

NE MANQUEZ PAS LE TROISIÈME NUMÉRO !

n°3

L'ÉLECTRONIQUE PLUS QUE JAMAIS À LA PORTÉE DE TOUS !

DISPONIBLE CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :
www.ed-diamond.com

RÉALISEZ DES TESTS QUE TOUT LE MONDE PEUT LIRE (ET ÉCRIRE)...

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

... ou presque ! En méthode Agile, le BDD (Behaviour Driven Development), ou en français le développement piloté par le comportement, est une approche intéressante sur la manière de créer et de gérer les tests, notamment pour des personnes avec des profils non techniques. Je ne ferai pas ici l'apologie de la méthode agile ou des méthodes BDD, TDD ou autres acronymes. Je trouve en revanche que la manière de créer et surtout de gérer les tests avec cette méthode qui exploite le langage naturel mérite d'être connue et utilisée.

« *Behavior Driven Development* » (appelé communément BDD) **[1]** est une méthode Agile qui a vocation de faire collaborer les développeurs et les équipes de tests (pour faire court). Cette méthode utilise le langage naturel (français, anglais, ...) pour décrire le fonctionnement d'un système.

Dans cette méthode, la rédaction des tests fonctionnels se fait en langage Gherkin **[2]** avec deux objectifs : la documentation des tests et l'automatisation de ces tests. Gherkin est un langage orienté lignes. La plupart des lignes commencent par un mot-clé. Voici un exemple de fichier écrit en langage Guerkin :

```
Feature: Un texte court qui décrit la fonctionnalité
  D'autres informations additionnelles sur la fonctionnalité...
  Ces informations sont ici juste pour information et ne seront
pas traitées.

  Scenario: un cas d'utilisation d'une fonctionnalité
    Given un prérequis
      And un autre prérequis
    When une action est réalisée par l'utilisateur
      And une autre action
      And encore une autre action
    Then tel événement se passe
    And le système se met dans tel contexte

  Scenario: un autre cas d'utilisation
    ...
```

Ces lignes seront ensuite associées à du code qui pourra mettre en place les prérequis, lancer ou simuler les actions et vérifier les événements et l'état des sorties.

Pour écrire des tests, il faut donc deux types de personnes :

- les personnes (non techniques) qui rédigeront les scénarios en langage naturel. Elles s'appuieront quand c'est possible sur des phrases déjà existantes et si ce n'est pas possible, elles définiront de nouvelles phrases.
- les personnes qui vont écrire le code qui permet de gérer les différentes phrases.

Il existe de nombreux outils pour réaliser de tels tests.... Le choix de l'outil dépendra d'une part du langage et d'autre part de vos préférences. Pour ma part, j'ai choisi d'utiliser **Behave [3]** pour le langage Python.

1 Installation des outils

L'installation de **behave** se fait on ne peut plus simplement :

```
# pip install behave
```

2 Un premier test

Dans un dossier donné, créez l'arborescence suivante :

```
$ mkdir steps
```

Puis, créez un fichier de *feature* (une fonctionnalité donc). Nommons-le **essai.feature** :

```
Feature: Premier essai de behave

Scenario: Passer un simple test
    Given behave est installé
    When on implémente un test
    Then le test va être passé
```

Puis, créez un fichier contenant les *steps* (les étapes du test). Nommons-le **steps/steps.py** :

```
# -*- coding: utf-8 -*-
from behave import given, when, then

@given(u'behave est installé')
def step_impl(context):
    # c'est forcément le cas donc on n'a rien à faire ici
    pass

@when(u'on implémente un test')
def step_impl(context):
    # on fait une action et on vérifie qu'elle se passe bien
    assert 1+1==2

@then(u'le test va être passé')
def step_impl(context):
    # on vérifie que tout s'est bien passé
    assert context.failed is False
```

Pour finir, depuis le dossier, lancez **behave** :

```
$ behave
Feature: Premier essai de behave # essai.feature:1

  Scenario: Passer un simple test  # essai.feature:3
    Given behave est installé      # steps/steps.py:4 0.000s
    When on implémente un test     # steps/steps.py:9 0.000s
    Then le test va être passé     # steps/steps.py:14 0.000s

1 feature passed, 0 failed, 0 skipped
1 scenario passed, 0 failed, 0 skipped
3 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m0.000s
```

Tadaaa ! Vous avez écrit et passé votre premier test en langage naturel. Nous avons tout d'abord créé un fichier **feature**. Chaque fonctionnalité d'une application pourra avoir son propre fichier **ma_fonctionnalite_machin.feature**. Ce fichier contient la description en langage naturel du test de la fonctionnalité. Vous noterez ici les premiers mots de chaque ligne en anglais (**given**, **when**, **then**), qui sont des mots-clés et la suite de chaque ligne en français, qui correspond à la description du test.

Ce mélange de « franglais » est ici un peu curieux, mais comme le lectorat de cette revue est français, donnons la part belle à Molière ! Pour finir la parenthèse, sachez qu'il est possible d'utiliser des mots-clés dans la langue de votre choix, mais j'ai choisi de ne pas aborder cette possibilité afin de rester concentré sur la syntaxe classique qui est utilisée dans la majorité des documentations.

Choisissez bien la langue de vos tests !

Même si je suis un ardent défenseur de la langue de Molière et que je ne supporte pas l'usage abusif et inutile de l'anglais, il faut reconnaître que réaliser les descriptions de tests en anglais sur un projet libre (tout comme le code, la documentation et les commentaires) ne peut qu'aider votre projet à trouver de nouveaux contributeurs, surtout dans le milieu des tests qui est souvent vu comme la tâche ingrate.

Reprenons le fichier **feature** :

```
Feature: Premier essai de behave

Scenario: Passer un simple test
    Given behave est installé
    When on implémente un test
    Then le test va être passé
```

La première ligne sert à décrire la fonctionnalité testée, qu'on appelle aussi *la story* en concept BDD.

Ensuite vient un bloc **Scenario** (il peut y en avoir plusieurs) avec le nom du scénario de test. Dans ce bloc, on retrouve un contexte avec **Given**, suivi d'un événement avec **When** et pour finir, un état de sortie avec **Then**. Ce qui donnerait « en bon français » :

« En supposant que behave est installé, quand on implémente un test alors le test va être passé ».

Voyons maintenant le fichier contenant les steps :

```
# -*- coding: utf-8 -*-
from behave import given, when, then
```

La première ligne est liée à Python 2.7 et permet l'utilisation de caractères utf-8 (les accents entre autres donc) et est nécessaire pour écrire des tests en français. La seconde permet d'importer les directives de **behave** (qui seront utilisées en tant que décorateurs).

Le fichier contient ensuite une fonction par directive et phrase.

```
@given(u'behave est installé')
def step_impl(context):
    # c'est forcément le cas donc on n'a rien à faire ici
    pass

@when(u'on implémente un test')
def step_impl(context):
    # on fait une action et on vérifie qu'elle se passe bien
    assert 1+1==2

@then(u'le test va être passé')
def step_impl(context):
    # on vérifie que tout s'est bien passé
    assert context.failed is False
```

Nous voyons bien qu'il y a peu de différences entre l'enveloppe de chaque fonction. Seul le décorateur utilisé et le contenu de la fonction change.

Vous aurez remarqué que le lien entre le langage naturel et les fonctions est fait via l'appel aux décorateurs avec en paramètres les phrases en langage naturel.

Notez l'usage de **context** dans la dernière fonction, nous allons y revenir.

3 Les mots-clés

3.1. Given, when, then

Vous avez pu voir que l'on utilise des mots-clés : **given**, **when**, **then**. D'un point de vue technique, il n'y a aucune distinction entre ces différents mots-clés. Il serait donc possible de les utiliser n'importe comment. Dans la pratique, afin que les tests soient compréhensibles et logiques, il faut veiller à utiliser correctement ces mots-clés.

Given est utilisé pour mettre le système (ou l'environnement) dans un état connu : on va configurer l'application, se placer sur une certaine page web d'un site, se connecter, etc.

When décrit les actions réalisées par l'utilisateur ou une interaction extérieure. Ce sont les interactions qui vont causer un changement dans le système (ou l'environnement) : remplir les champs d'un formulaire, puis le valider, démarrage de l'application ou activation d'un mode, etc.

Then permet de décrire ce qu'on doit observer suite aux actions qui se sont déroulées. Par exemple, vérifier que la soumission d'un formulaire affiche une page donnée avec un message donné, que l'interaction d'un système externe va provoquer la génération d'un fichier, etc.

3.2. And, but

Il existe d'autres mots-clés qui permettent de compléter les scénarios : **and**, **but**.

Ces mots-clés permettent simplement de remplacer de multiples **given**, **when** et **then** par quelque chose de plus lisible. Pour illustrer, les 2 exemples qui suivent sont identiques :

```
Scenario: reconnaître une personne
  Given une fille s'appelle Alexandra
  Given elle est petite
  Given elle a deux enfants
   When je la regarde
   Then je vois une fille que je connais
   Then c'est une amie
   Then c'est une fille diabolique
```

```
Scenario: reconnaître une personne
  Given une fille s'appelle Alexandra
    And elle est petite
    And elle a deux enfants
   When je la regarde
   Then je vois une fille que je connais
    And c'est une amie
    But c'est une fille diabolique
```

4 Le contexte

La variable **context**, que vous avez déjà remarquée dans les steps, est une variable dans laquelle le développeur des tests et **behave** peuvent stocker des informations afin de les partager. Il s'agit d'une instance de **behave.runner.Context**. Cette variable peut être utilisée à 3 endroits différents :

- lorsque **behave** lance une nouvelle feature ou un nouveau scénario ;
- des valeurs peuvent être définies dans le fichier **environment.py** ;
- elle peut être utilisée pour partager des données entre les steps d'un scénario. Par exemple :

```
@given('je télécharge une page web')
def step_impl(context):
    ma_page = telecharge("http://127.0.0.1/une_page/")
    context.contenu_page = ma_page.contenu
    context.code_http_page = ma_page.code_http

@then('elle s'est bien téléchargée')
def step_impl(context):
    eq_(context.code_http_page, 200)
    eq_(context.contenu_page, "<html><head></head><body>contenu
de la page</body></html>")
```

Behave ajoute de lui-même ces données à **context** :

- **table** : contient les tables associées à une step (sujet non abordé dans cet article) ;
- **text** : contient le texte de plusieurs lignes associé à une étape (sujet non abordé dans cet article) ;
- **failed** : cette variable est définie à **True** quand une step échoue.

5 Utilisation des variables

Behave permet évidemment de « variabiliser » les tests. Voici un exemple simple. Tout d'abord, le fichier **feature** :

```
Feature: test des paramètres

Scenario: Addition
    Given x vaut 1 et y vaut 3
    When on additionne x et y
    Then on obtient 4

Scenario: Soustraction
    Given x vaut 5 et y vaut 2
    When on soustrait y à x
    Then on obtient 3
```

Ici, rien de changé par rapport à précédemment. On va juste souhaiter pouvoir « variabiliser » les chiffres, ce qui permettra de réaliser plusieurs scénarios d'additions si on le souhaite (1+2, 4+6, …).

Maintenant, le fichier contenant les steps :

```
# -*- coding: utf-8 -*-
from behave import given, when, then, register_type
import parse


# -- conversion de type pour Number
@parse.with_pattern(r"\d+")
def parse_number(text):
    return int(text)


# -- enregistrement du type Number
register_type(Number=parse_number)


# -- les étapes
@given(u'x vaut {x:Number} et y vaut {y:Number}')
def step_impl(context, x, y):
    context.x = x
    context.y = y


@when(u'on additionne x et y')
def step_impl(context):
    context.resultat = context.x + context.y


@when(u'on soustrait y à x')
def step_impl(context):
    context.resultat = context.x - context.y


@then(u'on obtient {resultat:Number}')
def step_impl(context, resultat):
    assert context.resultat == resultat
```

Voici le résultat obtenu :

```
$ behave
Feature: test des paramètres # essai.feature:1

  Scenario: Addition          # essai.feature:3
    Given x vaut 1 et y vaut 3 # steps/steps.py:15 0.000s
    When on additionne x et y  # steps/steps.py:20 0.000s
    Then on obtient 4          # steps/steps.py:28 0.000s

  Scenario: Soustraction      # essai.feature:8
    Given x vaut 5 et y vaut 2 # steps/steps.py:15 0.000s
    When on soustrait y à x    # steps/steps.py:24 0.000s
    Then on obtient 3          # steps/steps.py:28 0.000s

1 feature passed, 0 failed, 0 skipped
2 scenarios passed, 0 failed, 0 skipped
6 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m0.001s
```

Notez les imports supplémentaires (**register_type** et **parse**). Ensuite, nous avons 2 nouveautés :

- une fonction **parse_number**, qui via la fonction **with_pattern** de la librairie **parse** permet de convertir le paramètre en type de notre choix (ici en entier). Notez que si nous ne fournissons pas un paramètre qui corresponde au pattern du décorateur à la fonction **parse_number**, alors **behave** ne va pas réussir à faire correspondre la step erronée avec sa fonction. Exemple ici en additionnant « a » et « 3 » :

```
$ behave
Feature: test des paramètres # essai.feature:1

  Scenario: Addition          # essai.feature:3
    Given x vaut a et y vaut 3 # None
    When on additionne x et y  # None
    Then on obtient 4          # None

  Scenario: Soustraction      # essai.feature:8
    Given x vaut 5 et y vaut 2 # steps/steps.py:15 0.000s
    When on soustrait y à x    # steps/steps.py:24 0.000s
    Then on obtient 3          # steps/steps.py:28 0.000s

Failing scenarios:
  essai.feature:3  Addition

0 features passed, 1 failed, 0 skipped
1 scenario passed, 1 failed, 0 skipped
3 steps passed, 0 failed, 2 skipped, 1 undefined
Took 0m0.000s

You can implement step definitions for undefined steps with
these snippets:

@given(u'x vaut a et y vaut 3')
def step_impl(context):
    assert False
```

- l'appel à la fonction **register_type** de **behave**, qui permet ici d'associer un type de paramètre (**Number**) à une fonction de décodage du paramètre (**parse_number**).

Ce mécanisme peut être utilisé pour charger tout un ensemble de données depuis une base de données via un simple identifiant et prend ici tout son sens ! Par exemple, à partir d'un couple nom/prénom, on va pouvoir récupérer toutes les données d'un client.

Viennent ensuite les classiques fonctions des steps. La première prend 2 paramètres en compte :

```
@given(u'x vaut {x:Number} et y vaut {y:Number}')
```

Nous retrouvons ici le type défini précédemment. La syntaxe est **{nom_de_variable, nom_de_type_libre_au_choix_du_developpeur}**. Nous retrouvons les variables utilisées dans la déclaration de la fonction :

```
def step_impl(context, x, y):
```

Et ces variables sont ensuite stockées dans le contexte :

```
  context.x = x
  context.y = y
```

Les fonctions liées aux steps d'addition et de soustraction se contentent d'utiliser le contexte pour exploiter les variables définies et stocker le résultat également dans le contexte. La dernière fonction qui permet de vérifier le calcul vérifie que le résultat attendu est bien celui déjà calculé et est présent dans le contexte.

6 Le fichier environment.py

Vous pouvez créer un fichier d'environnement, nommé **environment.py** dans le dossier des features. Ce fichier pourra contenir des fonctions qui seront appelées avant ou après chaque scénario ou step :

- **before_step(context, step)**
- **after_step(context, step)**
- **before_scenario(context, scenario)**
- **after_scenario(context, scenario)**
- **before_feature(context, feature)**
- **after_feature(context, feature)**

- **before_tag(context, tag)**
- **after_tag(context, tag)**

Il existe deux autres fonctions qui sont appelées avant et après « tout » :

- **before_all(context)**
- **after_all(context)**

Toutes ces fonctions peuvent être utiles suivant vos usages : envoyer le résultat de chaque scénario à un service web, logger, etc. Les fonctions **before_tag** et **after_tag** notamment peuvent être très puissantes : mise en place d'un contexte lié à un tag par exemple.

Voici dans quel ordre elles sont appelées :

```
before_all
for feature in all_features:
    before_feature
    for scenario in feature.scenarios:
        before_scenario
        for step in scenario.steps:
            before_step
                step.run()
            after_step
        after_scenario
    after_feature
after_all
```

7 Tags, WIP

Il est possible de tagger certains scénarios :

```
Feature: Premier essai de behave

Scenario: Passer un test rapide
    Given on charge un petit volume de données
    When on lance le traitement
    Then le traitement est ok

@slow
Scenario: Passer un test lent
    Given on charge un gros volume de données
    When on lance le traitement
    Then le traitement est ok
```

Si on lance **behave** avec l'option **--tags=slow**, alors seul le scénario taggé **@slow** sera lancé. A contrario, avec l'option **--tags=-slow**, seul le scénario qui n'est pas taggé **@slow** sera lancé.

Une autre manière courante d'utiliser les tags est l'utilisation de **@wip** (*work in progress* : travail en cours). Lorsque vous créez de nouveaux tests de fonctionnalités, taggez-les **@wip**. Pour tester uniquement les nouveaux scénarios (et donc gagner du temps), il suffit de lancer **behave** avec l'option **--tags=wip**.

Pour utiliser les tags dans les fonctions définies dans **environment.py**, vous pouvez procéder ainsi :

```
def before_feature(context, feature):
    [...]
    if 'browser' in feature.tags:
        # faire des traitements dédiés au tag @browser
        [...]
    [...]
```

Conclusion

Cet article n'a fait qu'aborder les capacités de **behave** et il est possible d'aller plus loin :

- réaliser des tests qui prennent en entrée des données sous forme de table, afin de valider plusieurs jeux de données grâce à un seul scénario ;
- interfacer **behave** avec d'autres solutions de test, comme Selenium par exemple, afin de réaliser des tests d'interfaces web ;
- etc.

Je ne souhaitais pas vous faire découvrir toutes les fonctionnalités de **behave**, mais vous présenter et vous initier à cet outil et cette manière de procéder. Si vous voulez aller plus loin, votre navigateur et une console seront vos alliés ! ■

Références

[1] Définition du BDD : http://fr.wikipedia.org/wiki/Behavior_Driven_Development

[2] Langage Gherkin : https://github.com/cucumber/cucumber/wiki/Gherkin

[3] Philosophie de Behave : http://pythonhosted.org/behave/philosophy.html

GESTIONNAIRE DE CONSOLES POUR GRAPPE DE SERVEURS

par ***Lionel Tricon*** *[Linuxien prosélytique et Ingénieur Caméléon]*

L'ubiquité est la faculté divine d'être présent partout en même temps ou présenté différemment, la faculté d'être présent physiquement en plusieurs lieux à la fois. Nous allons vous proposer d'appliquer cette utopie au sein d'une grappe de plusieurs machines sous Linux sur le principe suivant : tapez une fois dans une console, appliquez N fois sur un mur d'images. Merci de nous accompagner quelques instants dans la matrice (en prenant au passage la pilule rouge).

Se retrouver dans la situation de devoir gérer un ensemble homogène de machines sous Linux (pour du calcul, lancer des batchs, ou autres) est devenu de plus en plus courant et banalisé. Cela peut aller de 2 ou 4 nœuds jusqu'à un nombre beaucoup plus important (32 nœuds pour rester raisonnable par exemple).

Dans l'optique où vous êtes confronté à cette situation, une problématique concrète de l'administrateur reste la gestion de tous ces nœuds. Si le système est distribué, et si l'on ne met pas en œuvre une solution pour centraliser les images, on est en effet confronté à devoir gérer des machines qui possèdent exactement (en général), la même image système. Ce qui est plus simple à la base entraîne d'autres complications.

Chaque nœud de votre cluster, car le terme usité est cluster (ou grappe), possède donc la même image système, ayant été installée en général au même moment. Gérer des clusters sur des parcs de systèmes hétérogènes est généralement peu conseillé, la raison principale étant la gestion des versions des paquets système. Un sacré casse-tête en perspective pour ceux qui se retrouveraient dans cette situation.

Mais ici vient alors la première préoccupation qui va se poser quand, disposant d'un parc de N machines installées, il faut gérer ces systèmes, installer ou mettre à jour des logiciels, faire face au jour le jour aux différentes tâches qui incombent aux exploitants système.

Passé l'usage abusif de scripts permettant de dupliquer certaines tâches administratives sur l'ensemble des nœuds, via ou non le protocole SSH (il est relativement aisé d'automatiser des tâches via le protocole `SSH` voire `SFTP`) ou le pseudo langage « `expect` », l'administrateur doit se rendre à l'évidence : quel bonheur ce serait de pouvoir taper des commandes Unix et de pouvoir, d'un simple coup d'œil, observer ce qui se passe sur CHACUN des nœuds de notre cluster (on tape 1 fois, c'est diffusé N fois). Car rien ne remplace, en effet, l'aptitude à l'ubiquité du regard humain. Semblable à la matrice, une commande tapée sur une machine renvoie généralement dans les grandes lignes exactement le même visuel sur une autre machine dont elle serait l'exacte réplique. Par le jeu de la répétition des patterns, ce serait en effet idéal : mélanger l'acuité humaine avec l'exactitude du langage machine sur un mur d'images.

Cette solution, nous allons la construire ensemble, en vous guidant sur les différents aspects fondamentaux pour écrire un programme permettant, à travers une console de saisie, de dispatcher les commandes tapées vers un ensemble de consoles, chacune connectée à un nœud différent. Et le must du must est que la connexion aux £différentes

machines serait elle aussi automatisée. Une solution élégante, rusée, pour disposer d'un mur d'images et flatter son ego d'administrateur système, de développeur aguerri ou tout simplement pour le curieux soucieux d'épater ses collègues et a fortiori de se dégager de certaines tâches fastidieuses qui nous font perdre un temps précieux nous éloignant bien trop souvent des réelles exigences de ce métier.

Il n'est jamais pertinent de perdre du temps, souhaitons que la lecture de cet article vous en fasse gagner un peu. Car comme dit le proverbe « Sauvez un arbre, bouffez un castor », il est indispensable, pour résoudre un problème, de savoir qu'il existe et de trouver le meilleur moyen de le contourner. Et pour en revenir à notre (faux) proverbe, de ne pas se tromper de solution.

1 La solution technique mise en place

Pour gérer la connexion avec tous les nœuds de notre cluster, nous avons besoin d'une console. Sans être rétrogrades, nous allons nous diriger vers le bon vieil **xterm** que je connais pour ma part depuis mes débuts sur Unix. Pourquoi ? Cette commande est disponible partout, est facile à appeler et à paramétrer ; enfin, elle est relativement légère au niveau mémoire et CPU. Nous allons nous en servir à la fois pour gérer notre console de saisie ainsi que les différents terminaux de connexion aux nœuds du cluster.

Notre système cible étant Linux, lui-même un pseudo dérivé d'Unix, nous allons utiliser la particularité des pseudo-devices, que l'on nomme aussi PTY (à prononcer, *pitiwouai*), présentes sous forme de fichiers, pour dialoguer avec nos terminaux. Il nous faut donc :

- Lancer une console de saisie ;
- Lancer les N terminaux pour dialoguer avec les nœuds du cluster ;
- Lire tous les caractères tapés dans notre console ;
- Diffuser chaque caractère saisi vers les terminaux.

Implicitement, dans ce modèle, le rendu de chaque terminal de connexion est automatique ce qui fait que, finalement, outre le besoin de disposer d'un espace d'affichage en rapport avec le nombre de terminaux que l'on souhaite afficher, les différentes briques logicielles utiles sont de facto relativement aisées à trouver.

Le premier problème à résoudre concerne la communication avec nos terminaux. Chaque terminal est connecté à un device appelé PTY ou encore pseudo-terminal, qui est, pour simplifier, un émulateur de console asservi à un processus de contrôle avec lequel nous allons pouvoir communiquer. Ce device assez antique, qui existe depuis la fin des années 1960, d'abord sur DEC puis sur les UNIX, émule une console. Une console n'a pas toujours été un objet logiciel sous Linux pour taper des lignes de commandes, il a tout d'abord existé sous forme de véritable équipement, généralement un écran, connecté aux vénérables ancêtres qu'étaient les systèmes d'exploitation d'alors. Les machines n'avaient pas d'autres moyens pour dialoguer avec leurs opérateurs et cela a fait les choux gras, notamment, de la société *Digital*, avant de disparaître bien plus tard, devant l'avancée inexorable de la technologie. Mais en ce qui nous concerne ici, ce device représente un moyen idéal pour envoyer des commandes à nos terminaux.

La première approche naïve consiste à écrire une ligne de commandes vers le terminal. Lançons donc un terminal **xterm** dans une console et récupérons son pseudo-terminal via la commande **tty** :

```
user@linux-u3ez:~> xterm &
[1] 5555
```

Dans la console **xterm** :

```
user@linux-u3ez:~> tty
/dev/pts/2
```

Essayons maintenant d'écrire à partir de la console de départ sur le device récupéré auparavant :

```
user@linux-u3ez:~> echo "ls -la" > /dev/
pts/2
```

Vous constaterez, avec désappointement, que certes la commande est parfaitement écrite dans la console fille, mais que rien ne se passe, ce qui est fort fâcheux. La console fille ne semble pas interpréter la commande.

En soi, rien de bien étonnant. Ce serait une sacrée faille de sécurité de pouvoir écrire, comme cela, dans les consoles où le super-administrateur **root** serait connecté par exemple. Ce qui va nous sauver est la page man de **TTY_IOCTL (4)** qui nous indique que via la commande **TIOCSTI**, il nous est possible d'injecter des caractères dans la file d'attente du processus de contrôle du terminal. La seule contrainte, qui est (ou pas) un problème, est que cet appel système doit être obligatoirement réalisé via un processus possédant des droits **root**.

La première solution est de lancer le processus sous **root** (un peu contraint comme solution), la seconde est de rajouter un attribut SUID à l'exécutable (plus élégant et discret) :

```
root$ chown root:root notre_programme
root$ chmod u+s notre_programme
```

Pour lire les caractères tapés dans la console maître, nous allons utiliser la primitive **select** sur **stdin** (standard input) qui possède le numéro 0 (**stdout** possède le 1 et **stderr** le 2). À chaque caractère lu, nous allons l'envoyer vers les consoles filles lancées précédemment.

2 Mise en pratique : Un peu de code

Nous avons donc besoin d'une fonction pour écrire un caractère vers une console via son descripteur de fichier :

```
bool ttych(int p_fd, char* p_command)
{
    return ioctl(p_fd, TIOCSTI, p_command) != -1;
}
```

D'une autre fonction pour savoir si un caractère est disponible sur le flux **stdin** :

```
int iskb(void)
{
    struct timeval tv = { 0L, 0L };
    fd_set fds;
    FD_ZERO(&fds);
    FD_SET(0, &fds);
    return select(1, &fds, NULL, NULL, &tv);
}
```

Et d'une autre fonction pour lire le caractère (on considère que l'on a testé la présence du caractère avant) :

```
int getch(void)
{
    unsigned char c;
    (void)read(0, &c, sizeof(c));
    return c;
}
```

Nous disposons ici de notre squelette pour lire des caractères tapés et les envoyer vers une console arbitraire. Que nous reste-t-il à réaliser pour finaliser notre application ? Trouver le moyen de lancer le processus **xterm** en question, voire de lancer plusieurs processus vu que la finalité est tout de même de gérer un mur de terminaux. Nous allons pour cela utiliser l'appel système **fork** pour lancer un processus fils. Le code n'est pas bien compliqué et se comprend de lui-même :

```
int run_xterm(std::string p_command)
{
    int pid = fork();
    if(pid < 0)
    {
        printf("Error");
        exit(1);
    }
    else if (pid == 0)
    {
        execl("/usr/bin/xterm", "/usr/bin/xterm", "-geometry",
"60x15", "-e", p_command.c_str(),  NULL);
    }
    else
    {
        std::string l_value = "";
        while(l_value == "")
        {
            l_value = bash_pid(pid);
            if (l_value != "")
            {
                g_list[l_value] = open(l_value.c_str(), O_RDWR);
                break;
            }
        }
    }
}
```

La subtilité est, ici, de lancer la commande **xterm** avec la commande **-e** qui nous permet de lui indiquer quelle est la commande à lancer en paramètre. Par défaut, un **xterm** lance un **shell** via le **Bash**. Dans notre cas, un shell peut certes être utile, mais tout sera lancé sur la machine père. Il est plus indiqué de lancer dans ce cas un **ssh** sur une machine différente dans chaque console. Plus prosaïquement, en donnant à notre programme la liste des commandes à lancer, cela va nous simplifier la vie.

Considérons alors que la commande, dans chaque console fille, soit : **ssh login@ip_address**.

Dans le code précédent, vous noterez sans doute la présence de variables, voire de fonctions inconnues. La variable **g_list** est une simple *map* de la STL pour faire la liaison entre le nom du device et son descripteur de fichier au moment ou nous allons l'ouvrir (histoire de ne pas le faire à chaque fois que nous souhaitons écrire un caractère) :

```
std::map<std::string, int> g_list;
```

Certes c'est une variable globale, mais la clarté de l'article l'impose. La fonction **bash_pid** à laquelle nous passons le **pid** du processus fils lancé via le **fork** précédent a pour fonction de nous retourner le nom littéral du device (**PTY**) auquel il est rattaché. Car si l'on y réfléchit bien, il est certes possible de lancer la commande **TTY**

dans chacun des terminaux, mais vu que l'on ne peut pas trivialement récupérer ses sorties (sans être infaisable), il n'existe pas de moyen simple et rapide de récupérer cette information essentielle.

Nous allons donc faire appel au répertoire virtuel **/proc** qui est notre liaison directe et intrusive avec le noyau. Sous ce répertoire, chaque répertoire identifié par un numéro est en fait le répertoire nous apportant des indications sur un processus précis. En prenant notre exemple précédent, si l'on regarde le contenu du fichier **stat** dans le processus qui nous concerne (pour rappel le **5555**), nous obtenons les indications suivantes :

```
user@linux-u3ez:~> cat /proc/5555/stat

5555 (xterm) S 4988 5555 4988 34817 5640 4202496 3456 178 0 0 2 3
0 0 20 0 1 0 896616 11980800 1424 4294967295 134512640 134954556
3218074352 3218073488 1073886244 0 0 2097153 86022 0 0 0 17 0
0 0 0 0 0 134958768 134977920 145219584 3218077970 3218077976
3218077976 3218083821 0
```

Ce qui nous intéresse n'est pas tant que toutes ces séries de nombres nous restent assez obscures, c'est que le numéro **4988** est précisément le numéro père de la console de saisie. Cela se confirme aisément via un **ps auxf** :

```
user@linux-u3ez:~> ps auxf
user    4984  0.9  3.2 199376 63168 ?       Sl   18:57   0:30 kdeinit4: konsole [kdeinit]
user    4988  0.0  0.1   6356  2956 pts/1   Ss   18:57   0:00  \_ /bin/bash
user    5555  0.0  0.2  11700  5696 pts/1   S    19:48   0:00      \_ xterm
user    5559  0.0  0.1   6072  2720 pts/2   Ss+  19:48   0:00      |   \_ bash
user    5650 75.0  0.0   5056  1212 pts/1   R+   19:51   0:00      \_ ps auxf
```

Comme je n'ai pas indiqué de commande, le shell Bash s'est tout naturellement lancé. Si je lance un nouvel **xterm** avec un **ssh**, force est de constater que **bash** a été remplacé par un processus **ssh** :

```
user@linux-u3ez:~> xterm -e "ssh user@localhost"&
user    4984  0.9  3.4 204704 66320 ?       Rl   18:57   0:33 kdeinit4: konsole [kdeinit]
user    4988  0.0  0.1   6356  2956 pts/1   Ss   18:57   0:00  \_ /bin/bash
user    5555  0.0  0.2  11700  5696 pts/1   S    19:48   0:00      \_ xterm
user    5559  0.0  0.1   6072  2720 pts/2   Ss+  19:48   0:00      |   \_ bash
user    5661  0.2  0.2  11700  5692 pts/1   S    19:53   0:00      \_ xterm -e ssh
user@localhost
user    5665  0.0  0.1   7272  2284 pts/3   Ss+  19:53   0:00      |   \_ ssh
user@localhost
user    5750  0.0  0.0   5056  1208 pts/1   R+   19:54   0:00      \_ ps auxf
```

Le principe va donc être de regarder dans la liste de tous les processus, ceux dont le père est précisément celui de notre **xterm**. Ensuite, rien de plus simple, le lien symbolique **/proc/5665/fd/0** va pointer exactement vers le device sur lequel écrire :

```
user@linux-u3ez:~> ls -la /proc/5665/fd/0
lrwx------ 1 user users 64  2 avril 19:53 /proc/5665/fd/0 -> /dev/pts/3
```

La boucle est bouclée. Par ce procédé, certes un peu rustique et systématique, à chaque **xterm** lancé nous pourrons récupérer son device, et par là même la porte d'entrée pour piloter et asservir votre terminal.

Une proposition de fonction pourrait être la suivante :

```
std::string bash_pid(int p_pid)
{
    struct dirent *lecture;
    char l_buffer[4096];
    DIR *rep;

    rep = opendir("/proc/" );
    while ((lecture = readdir(rep)))
    {
        std::string l_pid = lecture->d_name;
        bool has_only_digits = (l_pid.find_first_not_
of("0123456789") == std::string::npos);
        if (has_only_digits)
        {
            std::string l_file = "/proc/"+l_pid+"/stat";
            std::ostringstream l_string;
            l_string << " (ssh) S " << p_pid << " ";

            FILE* l_fichier = fopen(l_file.c_str(), "r");
            if (l_fichier)
            {
                while(!feof(l_fichier))
                {
                    (void)fgets(l_buffer, 4096, l_fichier);
                    if (std::string(l_buffer).find(l_string.
str()) != std::string::npos)
                    {
                        fclose(l_fichier);
                        closedir(rep);

                        int l_value;
                        sscanf(l_buffer, "%d ", &l_value);

                        l_string.str("");
                        l_string << "/proc/" << l_value << "/fd/0";
                        return  l_string.str();
                    }
                }
                fclose(l_fichier);
            }
        }
    }
    closedir(rep);

    return "";
}
```

Le reste est un *main* simplissime qui lance toute une série de terminaux et qui renvoie de façon systématique tous les caractères lus dans les terminaux de façon cyclique :

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<termios.h>
#include<sys/select.h>
#include<sys/ioctl.h>
#include<fcntl.h>
#include<errno.h>
#include<dirent.h>

#include<string>
#include<map>
#include<fstream>
#include<sstream>
#include<cctype>

[...]

int main(int argc, char *argv[])
{
    std::map<std::string, int>::iterator l_iter;

    for (int i=1; i<argc; i++)
    {
        run_xterm(argv[i]);
    }

    while(1)
{
         if (iskb())
         {
             char buffer[2];
             sprintf(buffer, "%c", getch());

             int l_count = 0;
             for (l_iter=g_list.begin(); l_iter!=g_list.end();
l_iter++)
            {
                 if (l_iter->second!=-1 && ttych(l_iter->second,
buffer) == false)
                 {
                     l_iter->second = -1;
                 }
                 else
                 {
                     l_count++;
                 }
             }
             if (l_count == 0) break;
         }
    }
}
```

On compile le tout et on appelle le programme avec en paramètre une liste de commandes de connexion **SSH** à exécuter (*user@machine*).

Notez que c'est le *Window Manager* qui va s'occuper de ranger vos fenêtres (voir figure 1) dont la taille est paramétrée dans le code précédent lors de l'appel au **xterm**.

N'oubliez pas enfin de lancer l'application sous **root** (ou via l'attribut SUID), car sinon, rien ne va fonctionner.

Conclusion

Nous avons vu dans cet article comment mettre en place une solution pratique et plutôt astucieuse pour créer un gestionnaire de consoles permettant d'adresser tous les nœuds d'une grappe de clusters. Toutes les briques techniques mises en œuvre sont disponibles en standard sur n'importe quelle distribution.

Le résultat est du plus bel effet. Avec un minimum d'effort et des connaissances somme toute relativement abordables, nous avons conçu un outil pratique et futé. Bien évidemment, tout le monde ne dispose pas de plusieurs machines identiques sur lesquelles s'entraîner, mais, si c'est le cas, évaluez ce type d'outils, vous verrez qu'il pourrait vous rendre bien service. Il est même possible de le tester sur une seule machine en simulant une auto-connexion de multiples xterm (limité toutefois en nombre sur une même machine). Absolument inutile et donc complètement indispensable. ■

Fig. 1 : Exemple d'une connexion multiple.

LE CLOUD GAULOIS, UNE RÉALITÉ ! VENEZ TESTER SA PUISSANCE

EXPRESS HOSTING

Cloud Public
Serveur Virtuel
Serveur Dédié
Nom de domaine
Hébergement Web

sales@ikoula.com
01 84 01 02 66
express.ikoula.com

ENTERPRISE SERVICES

Cloud Privé
Infogérance
PRA/PCA
Haute disponibilité
Datacenter

sales-ies@ikoula.com
01 78 76 35 58
ies.ikoula.com

EX10

Cloud Hybride
Exchange
Lync
Sharepoint
Plateforme Collaborative

sales@ex10.biz
01 84 01 02 53
www.ex10.biz

LINAGORA

POUR NOËL,

LINAGORA veut vous aider à réaliser un de vos voeux :

FAIRE PASSER VOTRE ENTREPRISE AUX LOGICIELS LIBRES

@linagora

*Toute commande passé entre 1 décembre 2014 et 5 janvier 2015 bénéficiera d'une réduction de -15%. Offre valable pour toute nouvelle commande de prestations faisant partie de la gamme de services de LINAGORA, y compris, la formation, le support et la maintenance de logiciels libres (OSSA) et services hébergés (LinHosting). Sont exclues de l'offre les appels d'offres publics et toutes commandes passées dans le cadre de l'exécution, de l'évolution ou du renouvellement de contrats et/ou prestations déjà existants.

www.linagora.com vente@linagora.com