−Table des matières

Slony, réplication des données par trigger

Cet article, écrit par Guillaume Lelarge, a été publié dans le hors-série 44 du magazine GNU/Linux Magazine France, hors-série dédié à PostgreSQL. Il est disponible maintenant sous licence Creative Commons.

Il existe en gros trois façons de répliquer une base de données : répliquer les changements des fichiers (c'est ce que propose le Log Shipping, ainsi que DRBD), répliquer les instructions (c'est ce que propose pgpool) et répliquer les changements de données. Ce dernier cas est proposé par trois outils, dont Slony.

Un peu de théorie

Slony est un outil de réplication asynchrone pour un maître/plusieurs esclaves. Bien que ces derniers ne sont pas limités, il est généralement prudent de ne pas dépasser 12 nœuds.

Pour répliquer les données, il suffit que le moteur soit capable d'appeler une fonction personnalisée quand une donnée est insérée, modifiée ou supprimée. Or, cette capacité, tout moteur sérieux de bases de données l'a : cela s'appelle un trigger. PostgreSQL en dispose depuis bien longtemps, le système est rodé. Cependant, le trigger ne pourra pas envoyer les données sur un autre serveur. Il pourra par contre enregistrer les modifications dans une table qui sera surveillé par un démon qui lui enverra les données sur le serveur esclave. Il est à noter que les esclaves seront aussi munies de triggers qui empêcheront dans ce cas toute modification. En effet, les esclaves sont en lecture seule. Première limitation notable : PostgreSQL ne proposant pas de triggers sur les modifications de schéma, et n'acceptant pas l'ajout de triggers sur les catalogues systèmes, il n'est pas possible de répliquer les changements dans la structure de la base. Les opérations comme l'ajout d'une colonne, l'ajout ou la suppression d'une table sont toujours possibles, mais ne sont pas répliquées automatiquement.

Slony a tout un vocabulaire particulier que nous allons passer tout de suite en revue, histoire de discuter avec les bons termes. Un nœud est un serveur. Il peut être maître ou esclave d'un ou plusieurs sets. Un set est un ensemble cohérent de tables et de séquences, seuls objets que Slony sait répliquer. En effet, une fonction ou une vue ne se répliquent pas. Ils sont créés à un instant t et ne seront pas modifiées par la suite. S'ils le sont, c'est une modification de la structure de la base et ce n'est donc pas pris en compte automatiquement par Slony. Le cas des index est un peu particulier dans le sens où la modification des données des tables va modifier les données des index… mais c'est géré par PostgreSQL, Slony n'a rien à faire pour que cela fonctionne, que cela soit sur le maître ou sur les esclaves. Pour en revenir au set, il s'agit donc d'un ensemble cohérent de tables et de séquences, ayant un nœud maître et un ou plusieurs nœuds esclaves. Cette notion de set permet de définir un nœud A maître du set s1 et esclave du set s2 sur la même base de données. Autre terme important : l'abonnement. Il est généralement dit qu'un nœud s'abonne aux modifications des données réalisées pour un certain set. Cela se voit très nettement lors de la mise en place de Slony : nous créons le set sur un nœud (qui en sera donc le maître, au moins temporairement), puis nous abonnons un nœud à un set (ce nœud sera dans ce cas esclave).

Toutes les informations spécifiques à Slony sont stockées dans un schéma de la base de données à répliquer. Ce schéma contient des tables et des vues permettant de connaître la liste des objets répliqués, de savoir qui est le maître d'un set particulier, de connaître le lag de la réplication entre différents nœuds.

L'installation consiste donc simplement à ajouter quelques objets dans la base de données, dont des triggers, et à exécuter un démon d'envoi des données. C'est ce que nous allons mettre en pratique immédiatement.

Récapitulatif des prérequis

- Avoir une clé primaire sur chaque table à répliquer (cette clé est nécessaire pour pouvoir identifier chaque ligne de façon certaine);
- Ne pas utiliser l'ordre SQL TRUNCATE (instruction SQL permettant de vider très rapidement le contenu d'une table, beaucoup plus rapidement que ne le ferait un DELETE sur la table complète).
- Ne pas modifier fréquemment le schéma de la base de données répliquée.

Mise en pratique rapide

Considérons un serveur debian1 qui sera le maître et un serveur debian2 qui sera l'esclave. Nous allons créer une base sur debian1 avec un schéma assez simple composé de deux tables, chacune ayant une séquence.

Créons donc cette petite base:

```
debian1:~# su – postgres
postgres@debian1:~$ createdb base1
postgres@debian1:~$ psql base1
Bienvenue dans psql 8.3.7, l'interface interactive de PostgreSQL.

Saisissez:
   \copyright pour les termes de distribution
   \h pour l'aide-mémoire des commandes SQL
   \? pour l'aide-mémoire des commandes psql
   \g ou point-virgule en fin d'instruction pour exécuter la requête
```

```
   \q pour quitter

base1=# CREATE TABLE t1 (id serial PRIMARY KEY, texte text);
NOTICE: CREATE TABLE / PRIMARY KEY créera un index implicite « t1_pkey » pour la table « t1 »
CREATE TABLE
base1=# CREATE TABLE t2 (id serial PRIMARY KEY, texte text, autre boolean);
NOTICE: CREATE TABLE / PRIMARY KEY créera un index implicite « t2_pkey » pour la table « t2 »
CREATE TABLE
base1=# INSERT INTO t1 (texte) SELECT 'Ligne '||i::integer FROM generate_series(1, 100000) AS i;
INSERT 0 100000
base1=# INSERT INTO t2 (texte, autre)
base1-# SELECT 'LIGNE '||i::integer, true FROM generate_series(1, 10000) AS i;
INSERT 0 10000
base1=# INSERT INTO t2 (texte, autre)
base1-# SELECT 'LIGNE '||(i+10000)::integer, false FROM generate_series(1, 10000) AS i;
INSERT 0 10000
```

Voilà. Nous avons donc notre base à répliquer avec Slony.

Commençons par installer les paquets. Il existe deux paquets pour Slony, dont un dépendant de la version de PostgreSQL.

```
postgres@debian1:~$ exit
logout
debian1:~# aptitude install slony1-bin postgresql-8.3-slony1
[… messages d'installation …]
```

Les mêmes paquets sont à installer sur le serveur debian2. Comme vous le voyez, nous avons installé les paquets pour PostgreSQL 8.3. En effet, il n'y a pas encore de paquet Debian pour la version 8.4 au moment de l'écriture de cet article.

Nous allons ensuite créer un utilisateur PostgreSQL slony. Les connexions effectuées par les démons et les outils Slony passeront par cet utilisateur. C'est un moyen simple et efficace pour déterminer qui fait quoi sur les bases.

```
debian1:~# su – postgres
postgres@debian1:~$ createuser -sP base1
Saisissez le mot de passe pour le nouveau rôle :
Saisissez-le à nouveau :
```

L'option -s indique que l'utilisateur sera en fait un superutilisateur. L'option -P demande l'ajout immédiat d'un mot de passe.

Le même utilisateur doit être créé sur debian2. Le mot de passe peut ne pas être le même.

Ceci fait, nous devons nous assurer que l'utilisateur slony puisse se connecter sur la base base1 sur debian1 et sur debian2. En fait, par facilité, nous allons nous assurer que tous les utilisateurs PostgreSQL puissent se connecter à toutes les bases de données de ces deux serveurs. Pour cela, nous allons modifier le fichier de configuration des connexions. Ce fichier a pour nom pg_hba.conf et se trouve dans le répertoire /etc/postgresql/8.3/main sous Debian. Nous allons dans notre cas ajouter les trois lignes suivantes :

```
# Connexions Slony
host    all         all         192.168.0.7/32        md5
host    all         all         192.168.0.10/32       md5
```

Attention Si vous avez déjà modifié ce fichier, il est fort possible qu'ajouter les lignes à la fin du fichier ne donne pas le résultat escompté. En effet, PostgreSQL identifie la méthode d'authentification pour un client grâce à quatre données: sa méthode de connexion (socket Unix ou TCP/IP), la base, le nom de l'utilisateur et l'adresse IP du client (dans le cas d'une connexion TCP/IP). Ces quatres données peuvent correspondre à plusieurs lignes du fichier pg_hba.conf. Dans ce cas, PostgreSQL sélectionne la première ligne par ordre d'apparition dans le fichier correspondant à ces quatres donnes. Donc, si vous l'avez déjà modifié, il est préférable d'ouvrir le fichier avec un éditeur de texte et de le modifier manuellement.

Donc ainsi nous autorisons la connexion des utilisateurs postgres et slony (seuls utilisateurs actuellement créés). Seul l'utilisateur slony dispose ici d'un mot de passe. Or la connexion se fait par mot de passe, chiffré en md5 dans la communication entre le serveur et le client. Nous allons donc ajouter un mot de passe à l'utilisateur PostgreSQL

```
postgres:
postgres@debian1:~$ psql -q postgres
postgres=# ALTER USER postgres PASSWORD 'postgres';
```

L'option -q permet, entre autres, d'éviter le message de bienvenue. J'ai mis ici comme mot de passe postgres car ça simplifie les tests. Un mot de passe en production doit suivre des règles beaucoup plus draconiennes pour s'assurer de la sécurité des données.

Ensuite, toujours dans le but de se connecter d'un serveur à un autre, nous devons modifier, dans le fichier /etc/postgresql/8.3/main/postgresql.conf, le paramètre listen_addresses. Ce dernier vaut par défaut 'localhost', ce qui fait qu'il n'écoute que sur la boucle locale. Nous allons le mettre à '*' pour qu'il écoute sur chaque interface réseau. Ceci fait, il faut redémarrer PostgreSQL:

```
debian1:~# /etc/init.d/postgresql-8.3 restart
Restarting PostgreSQL 8.3 database server: main
```

À partir de debian1, testons la connexion vers debian1:

```
postgres@debian1:~$ psql -h debian1 -U slony -l
Mot de passe pour l'utilisateur slony : [... saisie du mot de passe ...]
       Liste des bases de données
    Nom      | Propriétaire | Encodage
-------------+--------------+-----------
 base1       | postgres     | UTF8
 postgres    | postgres     | UTF8
 template0   | postgres     | UTF8
 template1   | postgres     | UTF8
```

L'option -h indique le nom de l'hôte où se connecter, l'option -U précise le nom d'utilisateur PostgreQL à prendre en compte pour la connexion. L'option -l demande le renvoi de la liste des bases. C'est un moyen simple, non interactif pour savoir si nous avons réussi à nous connecter. Ce qui est bien le cas.

Testons la connexion vers debian2:

```
postgres@debian1:~$ psql -h debian2 -U slony -l
Mot de passe pour l'utilisateur slony : [... saisie du mot de passe ...]
       Liste des bases de données
    Nom      | Propriétaire | Encodage
-------------+--------------+-----------
 postgres    | postgres     | UTF8
 template0   | postgres     | UTF8
```

```
 template1   | postgres    | UTF8
```

Cela fonctionne. Il faut évidemment configurer debian2 de la même façon et s'assurer que la connexion fonctionne bien.

Nous allons créer un fichier .pgpass dans le répertoire personnel de l'utilisateur Unix postgres pour ne plus avoir à saisir de mot de passe.

```
postgres@debian1:~$ cat <<_EOF_ >> ~/.pgpass
> 192.168.0.7:5432:*:slony:slony
> 192.168.0.10:5432:*:slony:slony
> _EOF_
```

Pour que les outils puissent utiliser ce fichier, nous devons le rendre lisible et modifiable que par son utilisateur:

```
postgres@debian1:~$ chmod 600 ~/.pgpass
```

Maintenant, nous pouvons préparer la base du serveur esclave. Nous allons simplement recréer le schéma de la base maître, mais nous n'y placerons pas les données.

```
postgres@debian1:~$ createdb -h debian2 base1
Mot de passe :
postgres@debian1:~$ pg_dump -h debian1 -s base1 | psql -h debian2
Mot de passe :
Mot de passe :
SET
SET
SET
SET
SET
SET
SET
SET
CREATE TABLE
ALTER TABLE
CREATE SEQUENCE
ALTER TABLE
ALTER SEQUENCE
CREATE TABLE
ALTER TABLE
CREATE SEQUENCE
ALTER TABLE
ALTER SEQUENCE
ALTER TABLE
ALTER TABLE
ALTER TABLE
REVOKE
REVOKE
GRANT
GRANT
```

L'option -s de l'outil pg_dump permet de ne récupérer que les requêtes SQL de création du schéma. Nous avons bien un mot de passe à saisir car nous nous connectons en tant qu'utilisateur postgres. Ce dernier n'est pas indiqué dans le fichier .pgpass, donc la demande de mot de passe se fait. La première et la deuxième demandes concernent l'utilisateur postgres sur debian1, alors que la troisième demande concerne l'utilisateur postgres sur debian2. J'ai donné à la base esclave le même nom, mais j'aurais très bien pu en mettre un autre.

Nous devons aussi activer l'utilisation du langage PL/pgsql dans la base base1 de debian1 et de debian2:

```
postgres@debian1:~$ createlang -h debian1 plpgsql base1
Mot de passe :
postgres@debian1:~$ createlang -h debian2 plpgsql base1
Mot de passe :
```

Ce langage est nécessaire pour que Slony puisse installer les procédures stockées qu'il utilisera.

Il nous faut ensuite configurer les nœuds, les sets, bref toute la configuration de Slony. Pour cette fois, nous allons tout faire manuellement. Donc nous allons écrire un script Slonik. Slonik est un interpréteur de commandes très particulières dont vous trouverez la référence sur la documentation de Slony. Il faut savoir qu'il y a toujours deux parties dans un script slonik: le préambule (sorte de partie déclarative du nom du cluster et des nœuds en présence), puis les instructions à exécuter. Voici le script que nous allons exécuter:

```
#--
# PREAMBULE
#--

# définit le nom du schéma que Slony utilise pour stocker ses données
cluster name = replication;

# spécification des DSN permettant d'accéder à chaque serveur
node 1 admin conninfo = 'dbname=base1 host=debian1 user=slony';
node 2 admin conninfo = 'dbname=base1 host=debian2 user=slony';

#--
# INSTRUCTIONS DE MISE EN PLACE DE LA REPLICATION
#--

# préparation du cluster
init cluster (id=1, comment = 'noeud 1, debian1');

# creation du set
create set (id=1, origin=1, comment='notre premier set');

# ajout des deux tables au set
set add table (set id=1, origin=1, id=1, fully qualified name = 'public.t1', comment='table t1');
set add table (set id=1, origin=1, id=2, fully qualified name = 'public.t2', comment='table t2');

# ajout des deux sequences au set
set add sequence (set id=1, origin=1, id=1, fully qualified name = 'public.t1_id_seq',
comment='séquence de la table t1');
set add sequence (set id=1, origin=1, id=2, fully qualified name = 'public.t2_id_seq',
comment='séquence de la table t2');
```

```
# création du deuxieme noeud
store node (id=2, comment = 'noeud 2, debian2', event node=1);

# enregistrement des chemins entre les deux noeuds
store path (server = 1, client = 2, conninfo='dbname=base1 host=debian1 user=slony');
store path (server = 2, client = 1, conninfo='dbname=base1 host=debian2 user=slony');
```

Le préambule définit donc le nom du cluster et le moyen pour se connecter sur les deux futurs nœuds. Ensuite, nous allons initialiser le cluster (instruction init cluster), ce qui aura trois effets visibles : la création du schéma _replication (d'après le nom du cluster) dans la base de données base1, l'ajout des tables et vues spécifiques de Slony dans ce schéma et la création du premier nœud. Ensuite, nous créons un set vide, nous lui ajoutons les deux tables et les deux séquences à répliquer pour notre base. À noter le paramètre origin indiquant le nœud maître pour le set. Nous créons le deuxième nœud (qui correspondra au serveur debian2) et nous enregistrons tous les chemins possibles pour aller d'un nœud à un autre.

Il faut donc stocker ce texte dans un fichier que nous allons fournir à l'interpréteur slonik:

```
postgres@debian1:~$ cat mep.slonik | slonik
```

Aucun message d'erreur, tout va bien. Attention, si vous avez des messages d'erreurs, la partie du script qui a pu s'exécuter avec succès a pu ajouter le schéma de réplication. Dans ce cas, une fois l'erreur corrigée dans le script slonik, vous devez supprimer le schéma avant de re-commencer la commande ci-dessus.

Avant de lancer les démons Slony, nous allons les configurer. Il nous suffit de copier le fichier exemple et de le personnaliser pour prendre en compte notre contexte:

```
debian1:~$ gunzip -c /usr/share/doc/slony1/slon.conf-sample.gz > /etc/slony1/slon.conf
debian1:~$ vi /etc/slony1/slon.conf
```

Deux paramètres sont à configurer. C'est évidemment un minimum et je vous conseille fortement de jeter un œil aux différents paramètres disponibles une fois qu'une première réplication fonctionne.

Les deux paramètres sont cluster_name, qui doit être décommenté et passé à 'replication', et conn_info, qui doit aussi être décommenté et passé à 'host=debian1 port=5432 dbname=base1 user=slony'.

Nous pouvons maintenant lancer le démon du serveur debian1 en tant qu'utilisateur postgres:

```
postgres@debian1:~$ slon -f /etc/slony1/slon.conf &>slon.log
```

L'option -f permet de préciser le fichier de configuration à utiliser. La suite de la commande redirige toutes les traces vers le fichier slon.log. Maintenant que ceci est fait, le démon doit être en cours d'exécution.

Il est nécessaire de faire la même configuration sur debian2 pour aussi exécuter le démon sur ce nœud. Par contre, le paramètre conn_info aura une configuration légèrement différente : 'host=debian2 port=5432 dbname=base1 user=slony'. Cela étant fait, il ne reste plus qu'à le lancer de la même façon que pour debian1.

Si vous regardez le fichier de traces sur l'un comme sur l'autre serveur, vous verrez qu'il est très verbeux. Vous verrez aussi de nombreuses opérations de synchronisation. Pour l'instant, elles sont sans intérêt. Nous avons déclaré un set, mais nous n'avons indiqué aucun esclave (le nœud du serveur qui sera esclave est déclaré, mais pas le fait que ce dernier est abonné aux modifications du set1). Profitons-en pour regarder le contenu de base1 sur le serveur debian2:

```
postgres@debian1:~$ psql -q base1
base1=# SELECT count(*) FROM t1;
 count
-------
     0
(1 ligne)
base1=# SELECT count(*) FROM t1;
 count
-------
     0
(1 ligne)
```

C'est normal. Nous n'avons restauré que le schéma, pas les données et debian2 n'est toujours pas abonné aux modifications du set dont debian1 est le maître. Abonnons debian2. Nous devons de nouveau créer un petit script Slonik.

```
#--
# PREAMBULE
#--

# définit le nom du schéma que Slony utilise pour stocker ses données
cluster name = replication;

# spécification des DSN permettant d'accéder à chaque serveur
node 1 admin conninfo = 'dbname=base1 host=debian1 user=slony';
node 2 admin conninfo = 'dbname=base1 host=debian2 user=slony';

#--
# INSTRUCTIONS D'ABONNEMENT DU NOEUD 2 AU SET 1
#--

# abonnement du noeud 2
subscribe set (id=1, provider=1, receiver=2, forward = yes);
```

Le préambule est toujours là. La seule différence réside dans l'instruction « subscribe set ». Cette instruction permet d'ajouter dans le schéma _replication l'information que le nœud 1 (provider, soit un fournisseur de données) a comme nouvel abonné le nœud 2 (receiver, donc le récepteur des données). Exécutons ce script:

```
postgres@debian1:~$ cat mer.slonik | slonik
```

Là-aussi, aucun message d'erreur, tout va bien. Regardons ce qu'il vient de se passer au niveau du nœud 2:

```
base1=# SELECT count(*) FROM t1;
 count
--------
 100000
(1 ligne)
base1=# SELECT count(*) FROM t1;
```

```
 count
-------
 20000
(1 ligne)
```

Et voilà, nos données sont répliquées. Faisons quelques actions sur debian1 et vérifions que debian2 est bien mis à jour.

```
postgres@debian1:~$ psql -q base1
base1=# INSERT INTO t1 (texte) VALUES ('un texte') RETURNING id, texte;
   id   |     texte
--------------------------
 100001 | un super texte
(1 ligne)

INSERT 0 1
```

Vérifions l'insertion sur debian2:

```
postgres@debian2:~$ psql -q base1
base1=# SELECT * FROM t1 WHERE id=100001;
   id   |     texte
--------------------------
 100001 | un super texte
(1 ligne)
```

Les modifications et les suppressions sont aussi possibles sur debian1, et seront aussitôt répliquées sur debian2.

« aussitôt » est certainement un peu simpliste. Vous pourrez constater un certain délai dans la mise à jour du serveur esclave. C'est tout à fait logique. Slony est une réplication asynchrone, il ne faut donc pas s'attendre à ce que les données soient immédiatement transmises. Le délai de réplication des données dépend d'un paramètre qu'il sera possible de modifier suivant les besoins. En fait, cela s'avèrera soit suffisant soit handicapant.

Notez par contre qu'il est impossible de modifier des données sur l'esclave:

```
postgres@debian2:~$ psql -q base1
base1=# INSERT INTO t1 (texte) VALUES ('un autre texte');
ERREUR:  Slony-I: Table t1 is replicated and cannot be modified on a subscribed node
```

Le message d'erreur est clair : la table t1 est répliquée et ne peut donc pas être modifié à partir d'un nœud abonné.

Voyons maintenant les limites de Slony. Bien que vous ayez la possibilité d'utiliser l'instruction COPY (qui déclenche en fait le trigger INSERT), l'instruction TRUNCATE ne déclenche aucun trigger. Seule la version 8.4 le fait. Testons cela:

```
postgres@debian1:~$ psql -q base1
base1=# TRUNCATE t1;
TRUNCATE TABLE
base1=# SELECT count(*) FROM t1;
 count
-------
     0
(1 ligne)
base1=# \q
postgres@debian2:~$ psql -q -h debian2 base1
base1=# SELECT count(*) FROM t1;
 count
--------
 100001
(1 ligne)
```

La table t1 de debian1 est vide alors que la table t1 de debian2 contient un grand nombre de lignes. Il faut donc oublier TRUNCATE lorsque vous utilisez Slony.

Maintenant, ajoutons une colonne à la table t1 de debian1:

```
postgres@debian1:~$ psql -q base1
base1=# ALTER TABLE t2 ADD COLUMN test boolean;
ALTER TABLE
base1=# INSERT INTO t2 (texte, autre, test) VALUES ('yop', true, false);
INSERT 0 1
base1=# SELECT * FROM t2 WHERE texte='yop';
   id  | texte | autre | test
--------------------------------
 20001 | yop   | t     | f
(1 ligne)
base1=# \q
postgres@debian2:~$ psql -q -h debian2 base1
base1=# SELECT * FROM t2 WHERE texte='yop';
(0 lignes)
base1=# \d t2
                      Table « public.t2 »
 Colonne |  Type   |                  Modificateurs
---------+---------+-------------------------------------------------
 id      | integer | not null default nextval('t2_id_seq'::regclass)
 texte   | text    |
 autre   | boolean |
Index :
    « t2_pkey » PRIMARY KEY, btree (id)
```

Il n'y a pas la nouvelle colonne. Il n'y a pas non plus la nouvelle ligne. Il faut ajouter la nouvelle colonne aux esclaves, manuellement:

```
postgres@debian2:~$ psql -q base1
base1=# ALTER TABLE t2 ADD COLUMN test boolean;
ALTER TABLE
base1=# SELECT * FROM t2 WHERE texte='yop';
   id  | texte | autre | test
--------------------------------
 20001 | yop   | t     | f
(1 ligne)
```

Et voilà.

Le schéma de réplication

Maintenant que Slony est fonctionnel et qu'on a vu ses limitations, jetons un œil au schéma de Slony.

```
postgres@debian1:~$ psql -q base1
base1=# \dn
        Liste des schémas
       Nom         | Propriétaire
--------------------+---------------
 _replication       | slony
 information_schema | guillaume
 pg_catalog         | guillaume
 pg_toast           | guillaume
 pg_toast_temp_1    | guillaume
 public             | guillaume
(5 lignes)
```

Il y a bien un schéma _replication. Il a été créé par l'instruction Slonik « init cluster ». Son propriétaire est donc l'utilisateur slony, étant donné que toute connexion des démons et outils Slony se fait avec cet utilisateur. Cette dernière a aussi ajouté quelques tables, vues et séquences.

```
base1=# SET search_path TO _replication;
base1=# \d
glmf=# \d
                  Liste des relations
    Schéma    |            Nom             |   Type   | Propriétaire
--------------+----------------------------+----------+--------------
 _replication | sl_action_seq              | séquence | slony
 _replication | sl_archive_counter         | table    | slony
 _replication | sl_config_lock             | table    | slony
 _replication | sl_confirm                 | table    | slony
 _replication | sl_event                   | table    | slony
 _replication | sl_event_seq               | séquence | slony
 _replication | sl_listen                  | table    | slony
 _replication | sl_local_node_id           | séquence | slony
 _replication | sl_log_1                   | table    | slony
 _replication | sl_log_2                   | vue      | slony
 _replication | sl_log_status              | séquence | slony
 _replication | sl_node                    | table    | slony
 _replication | sl_nodelock                | table    | slony
 _replication | sl_nodelock_nl_conncnt_seq | séquence | slony
 _replication | sl_path                    | table    | slony
 _replication | sl_registry                | table    | slony
 _replication | sl_rowid_seq               | séquence | slony
 _replication | sl_seqlastvalue            | vue      | slony
 _replication | sl_seqlog                  | table    | slony
 _replication | sl_sequence                | table    | slony
 _replication | sl_set                     | table    | slony
 _replication | sl_setsync                 | table    | slony
 _replication | sl_status                  | vue      | slony
 _replication | sl_subscribe               | table    | slony
 _replication | sl_table                   | table    | slony
 _replication | sl_trigger                 | table    | slony
(26 lignes)
```

Le tableau Tab1 liste les tables et vues importantes du schéma de réplication.

Tab1. Liste des tables et vues importantes du schéma de Slony

Table/Vue	Commentaires
sl_node	liste des nœuds
sl_path	liste des chemins entre les différents nœuds
sl_set	liste des sets
sl_subscribe	liste des abonnements
sl_table	liste des tables ajoutées aux différents sets
sl_sequence	liste des séquences ajoutées aux différents sets
sl_log1 et sl_log2	liste des modifications pour chaque paire nœud/set
sl_status	statut de la réplication

Il y a bien plus à voir dans ce schéma. Le lecteur curieux pourra regarder le contenu de ce schéma, ce qui lui donnera une meilleure compréhension du fonctionnement de Slony.

Comment se faciliter la vie

Comme pour le Log Shipping, certains utilisateurs ont préféré écrire des outils pour faciliter l'utilisation de Slony.

Les outils altperl

Ce sont des scripts écrits en Perl par la communauté. Ils sont fournis avec les paquets Slony déjà installés.

Ils ont principalement pour but d'avoir à se passer de l'écriture des scripts Slonik, au prix de la modification d'un fichier de configuration.

slonik_build_env est un peu particulier dans le sens où il ne prépare pas un script Slonik, mais aide à la configuration. Voici un exemple de son utilisation:

```
postgres@debian1:~$ slonik_build_env -node debian1:base1:slony -node debian2:base1:slony
&add_node(host => 'debian1', dbname => 'base1', port =>5432,
        user=>'slony', password=>'', node=>1 );
&add_node(host => 'debian2', dbname => 'base1', port =>5432,
        user=>'slony', password=>'', node=>2 , parent=>1);
@KEYEDTABLES=(
        "public.t1",
        "public.t2",
);
@SEQUENCES=(
        "public.t1_id_seq",
        "public.t2_id_seq",
```

```
);
```

Étrangement, cela ne génère pas une sortie compatible directement avec le fichier de configuration. Il est généralement conseillé de concaténer cette sortie à la fin du fichier de configuration, puis d'éditer le fichier ainsi obtenu. Le très gros intérêt de ce script est de récupérer la liste des tables. Dans notre exemple précédent, le fichier de configuration est très simple à faire car il n'y a que deux tables. Avec une base bien plus conséquente de plusieurs centaines de tables, c'est clairement une autre affaire et une automatisation, même partielle, est bien agréable.

Les scripts slonik_init_cluster, slonik_create_set, slonik_subscribe_set permettent la mise en place de la réplication. Si nous devions refaire la réplication précédente avec les outils Slonik, cela donnerait ceci.

Le fichier de configuration /etc/slony1/slon_tools.conf est le suivant:

```
if ($ENV{"SLONYNODES"}) {
    require $ENV{"SLONYNODES"};
} else {
    $CLUSTER_NAME = 'replication';
    $LOGDIR = '/var/log/slony1';
    # $SYNC_CHECK_INTERVAL = 1000;
    $MASTERNODE = 1;

    add_node(node     => 1,
             host     => 'debian1',
             dbname   => 'base1',
             port     => 5432,
             user     => 'slony',
             password => 'slony');

    add_node(node     => 2,
             host     => 'debian2',
             dbname   => 'base1',
             port     => 5432,
             user     => 'slony',
             password => 'slony');

}

$SLONY_SETS = {
    "set1" => {
        "set_id" => 1,
        # "origin" => 1,
        # foldCase => 0,
        "table_id"    => 1,
        "sequence_id" => 1,

        "pkeyedtables" => [
                           'public.t1',
                           'public.t2',
                           ],

        "sequences" => ['public.t1_id_seq',
                        'public.t2_id_seq',
                        ],
    },
};

if ($ENV{"SLONYSET"}) {
    require $ENV{"SLONYSET"};
}

# Please do not add or change anything below this point.
1;
```

Ensuite, initialisons le cluster. En lançant la commande seule, elle affiche le résultat du script Slonik demandé:

```
postgres@debian1:~$ slonik_init_cluster

# INIT CLUSTER
cluster name = replication;
 node 1 admin conninfo='host=debian1 dbname=base1 user=slony port=5432 password=slony';
 node 2 admin conninfo='host=debian2 dbname=base1 user=slony port=5432 password=slony';
  init cluster (id = 1, comment = 'Node 1 - base1@debian1');

# STORE NODE
  store node (id = 2, event node = 1, comment = 'Node 2 - base1@debian2');
  echo 'Set up replication nodes';

# STORE PATH
  echo 'Next: configure paths for each node/origin';
  store path (server = 1, client = 2, conninfo = 'host=debian1 dbname=base1 user=slony port=5432 password=slony');
  store path (server = 2, client = 1, conninfo = 'host=debian2 dbname=base1 user=slony port=5432 password=slony');
  echo 'Replication nodes prepared';
  echo 'Please start a slon replication daemon for each node';
```

En dehors de quelques instructions echo supplémentaires, nous retrouvons le préambule et les instructions de préparation du cluster: init cluster, store node, store path. Maintenant, exécutons-le:

```
postgres@debian1:~$ slonik_init_cluster | slonik
<stdin>:10: Set up replication nodes
<stdin>:13: Next: configure paths for each node/origin
<stdin>:16: Replication nodes prepared
<stdin>:17: Please start a slon replication daemon for each node
```

À partir de ce moment, nous pouvons déjà lancer les démons sur chaque nœud. Cela se fait exactement de la même façon que précédemment:

```
postgres@debian1:~$ slon -f /etc/slony1/slon.conf &>slon.log
```

et

```
postgres@debian2:~$ slon -f /etc/slony1/slon.conf &>slon.log
```

Ensuite, nous pouvons créer le set grâce au script slonik_create_set à qui nous passons un seul argument, le numéro du set à créer:

```
postgres@debian1:~$ slonik_create_set 1 | slonik
<stdin>:16: Subscription set 1 created
<stdin>:17: Adding tables to the subscription set
<stdin>:21: Add primary keyed table public.t1
<stdin>:25: Add primary keyed table public.t2
<stdin>:28: Adding sequences to the subscription set
<stdin>:32: Add sequence public.t1_id_seq
<stdin>:36: Add sequence public.t2_id_seq
<stdin>:37: All tables added
```

Et enfin abonnons-nous le nœud 2 au set 1:

```
postgres@debian1:~$ slonik_subscribe_set 1 2 | slonik
<stdin>:10: Subscribed nodes to set 1
```

Et c'est terminé. La réplication fonctionne.

D'autres scripts existent, par exemple ceux de suppression/désinstallation (slonik_drop_node, slonik_drop_set, slonik_drop_table, slonik_uninstall_nodes, slonik_unsubscribe_set), ceux de bascule (slonik_move_set, slonik_failover, dont nous reparlerons un peu plus tard), ainsi que divers autres (slonik_print_preamble, slonik_merge_sets).

Le seul dont nous allons parler tout de suite est slonik_execute_script. Ce script est intéressant car il va permettre d'exécuter un script SQL sur les différents nœuds. C'est donc un moyen propre pour mettre à jour le schéma d'une base. Testons sur l'ajout d'une colonne à t1:

```
postgres@debian1:~$ slonik_execute_script 1 -c "ALTER TABLE t1 ADD COLUMN nouvellecolonne integer;" | slonik
DDL script consisting of 2 SQL statements
DDL Statement 0: (0,50) [ALTER TABLE t1 ADD COLUMN nouvellecolonne integer;]
DDL Statement 1: (50,52) []
Complete DDL Event...
DDL submission to initial node - PGRES_TUPLES_OK
postgres@debian1:~$ psql -h debian1 -U slony -c "\d t1" base1
                    Table « public.t1 »
    Colonne     |  Type   |                Modificateurs
----------------+---------+-----------------------------------------------
 id             | integer | not null default nextval('t1_id_seq'::regclass)
 texte          | text    |
 nouvellecolonne | integer |
Index :
    « t1_pkey » PRIMARY KEY, btree (id)
Triggers :
    _replication_logtrigger_1 AFTER INSERT OR DELETE OR UPDATE ON t1 FOR EACH ROW EXECUTE PROCEDURE _replication.logtrigger('_replication', '1', 'kvv')

postgres@debian1:~$ psql -h debian2 -U slony -c "\d t1" base1
                    Table « public.t1 »
    Colonne     |  Type   |                Modificateurs
----------------+---------+-----------------------------------------------
 id             | integer | not null default nextval('t1_id_seq'::regclass)
 texte          | text    |
 nouvellecolonne | integer |
Index :
    « t1_pkey » PRIMARY KEY, btree (id)
Triggers :
    _replication_denyaccess_1 BEFORE INSERT OR DELETE OR UPDATE ON t1 FOR EACH ROW EXECUTE PROCEDURE _replication.denyaccess('_replication')
```

La nouvelle colonne se trouve bien dans la table t1 pour les deux serveurs.

Ce qu'il faut retenir des outils AltPerl, c'est qu'ils nous évitent d'avoir à se transformer en expert du langage Slonik. Même s'il est toujours intéressant de vérifier le script Slonik avant de l'exécuter, une simple lecture permettra de s'assurer de son bon fonctionnement.

Le paquet slony_ctl

Développé pour ses besoins par Stéphane Schildknecht, de la société Dalibo, ce paquet a pour but de simplifier encore plus la mise en place d'une réplication. Exécuter deux scripts bash suffit à lancer la réplication.

Reprenons notre exemple et mettons en place une réplication Slony à partir de cet outil. Commençons par récupérer cet outil sur pgfoundry.org, la forge de PostgreSQL:

```
postgres@debian1:~$ wget -q http://pgfoundry.org/frs/download.php/2253/slony1-ctl-1.1.4.tar.gz
```

Nous allons le déballer et le configurer:

```
postgres@debian1:~$ tar xfz slony1-ctl-1.1.4.tar.gz
postgres@debian1:~$ cd slony1-ctl/slony-ctl
```

Il existe ici deux répertoires: etc contient les fichiers de configuration, outils contient tous les scripts bash. Allons dans etc. Le premier fichier à configurer est bases.h:

```
#INSTANCE      NODE    BASE          HOST          PORT
replibase1     1       base1         debian1       5432
replibase1     2       base1         debian2       5432
```

Ce fichier contient la liste des nœuds. Il indique l'instance de réplication, le numéro du nœud, le nom de la base, celui de l'hôte et enfin le numéro de port TCP/IP. Ici, nous avons deux nœuds, donc deux lignes.

Le second fichier à configurer est relations.h. Il précise la relation entre chaque nœud pour chaque instance:

```
#INSTANCE   SET     MAITRE  ESCLAVE
replibase1  1       1       2
```

Ici, nous indiquons que le nœud 1 est le fournisseur des données pour le nœud 2 dans le cadre du set1 pour l'instance replibase1.

Le dernier fichier contient des informations plus générales, comme l'emplacement des différents exécutables, fichiers de configuration, etc. Il s'appelle slony_include.h et voici son contenu dans notre cas:

```
# Configuration file for slony1-ctl
```

```
# Path to PG install_dir
PG=/usr

# Path to slony1 install dir
SL_PATH=/usr
SLON_CTL=/var/lib/postgresql/slony1-ctl/slony-ctl

SLON_BIN=${SL_PATH}/bin
SLON_TOOLS=${SL_PATH}/outils

SLON_ETC=$SLON_CTL/etc
SLON_OUTILS=$SLON_CTL/outils
SLON_LOG=$SLON_CTL/logs
SLON_TEMP=/tmp

SLONY_USER=slony
#SLONY_PWD=mypass

# Some binaries
slon_exec=${SLON_BIN}/slon
slonik_exec=${SLON_BIN}/slonik
psql_exec=$PG/bin/psql
pgdump_exec=$PG/bin/pg_dump
```

Le répertoire des traces n'existe pas, il faut le créer soi-même:

```
postgres@debian1:~/slony1-ctl/slony-ctl/etc$ cd ..
postgres@debian1:~/slony1-ctl/slony-ctl$ mkdir logs
```

Maintenant, nous allons utiliser le premier script base. Ce script va vérifier un certain nombre de points et créer un fichier de réplication qui sera exécuté par le second script.

```
postgres@debian1:~/slony1-ctl/slony-ctl$ cd outils
postgres@debian1:~/slony1-ctl/slony-ctl/outils$ ./01_create_init.sh -c replibase1
Création des fichiers de préambule
Création du fichier d'initialisation de la réplication des bases de replibase1
OK
```

Le « OK » indique que tout s'est bien passé. Le fichier /tmp/replibase1.slon a été créé. Le lire explique un peu comment fonctionne slony1-ctl. Lançons la réplication:

```
postgres@debian1:~/slony1-ctl/slony-ctl/outils$ ./02_exec_init.sh -c replibase1
Démarrage réplication replibase1...
OK
```

Et voilà. Difficile de faire plus simple.

Vérifions que tout fonctionne:

```
postgres@debian1:~/slony1-ctl/slony-ctl/outils$ psql -q -h debian2 base1
Mot de passe :
base1=# select count(*) from t1;
 count
--------
 100000
(1 ligne)

base1=# select count(*) from t2;
 count
-------
 20000
(1 ligne)

base1=# \q
postgres@debian1:~/slony1-ctl/slony-ctl/outils$ psql -q -h debian1 base1
Mot de passe :
base1=# insert into t2 (texte, autre) values ('yop', true);
INSERT 0 1
base1=# \q
postgres@debian1:~/slony1-ctl/slony-ctl/outils$ psql -q -h debian2 base1
base1=# select count(*) from t2;
 count
-------
 20001
(1 ligne)
```

L'outil est encore jeune, il reste encore quelques améliorations possibles pour en faire un produit mature. Néanmoins, la base est bonne et rend la mise en place de la réplication d'une base avec Slony extrêmement simple. De plus, il est facilement personnalisable étant donné qu'il s'agit de simples scripts bash.

Comment surveiller la réplication

Dans une réplication Slony, il est nécessaire de surveiller deux choses:

- que les données sont bien répliquées
- et que le schéma ne change pas.

Pour le premier, nous allons pouvoir nous aider d'une vue Slony appelée sl_status. Cette vue peut nous donner un grand nombre d'informations:

```
postgres@debian1:~$ psql -q base1
base1=# \x
base1=# SELECT * FROM _replication.sl_status;
-[RECORD 1]----------------------------------------------------
st_origin                | 1
st_received              | 2
st_last_event            | 1454
st_last_event_ts         | 2009-08-12 17:28:54.284162
st_last_received         | 1454
st_last_received_ts      | 2009-08-12 16:42:43.070423
```

```
st_last_received_event_ts | 2009-08-12 17:28:54.284162
st_lag_num_events         | 0
st_lag_time               | 00:00:06.284245
```

Voici le descriptif de chaque colonne:

- st_origin est l'identifiant du nœud origine (le maître);
- st_received est l'identifiant du nœud abonné (l'esclave);
- st_last_event est l'identifiant du dernier événement survenu;
- st_last_event_ts est l'horodatage du dernier événement survenu;
- st_last_received est l'identifiant du dernier événement reçu;
- st_last_received_ts est l'horodatage du dernier événement reçu;
- st_last_received_event_ts est l'horodatage du dernier événement reçu;
- st_lag_num_events est le lag en nombre d'événements (autrement dit st_last_received – st_last_event);
- st_lag_time est le lag en heures/minutes/secondes (autrement dit l'heure actuelle – st_last_received_event_ts).

Les colonnes les plus intéressantes sont donc les deux dernières, le but étant d'avoir le lag minimum possible. Généralement, le nombre st_lag_num_events devra se trouver à 0 ou à 1. Plus il est petit, et plus les données sont synchronisées. Si les deux serveurs sont de plus en plus désynchronisés, il va falloir trouver la raison de ce problème et pour cela, les traces sont votre meilleur ami.

Pour s'assurer que le schéma ne change pas, vous pouvez utiliser un outil comme check_postgres. Écrit par Greg Sabino Mulane, autre grand nom dans le monde PostgreSQL et Perl, de la société End Point Corporation, ce script Perl est principalement conçu pour s'intégrer à un système Nagios, voire même à un système MRTG. Il est capable de réaliser différentes actions. Celle qui nous intéresse ici est same_schema. Cette action est capable de vérifier les différences de schéma entre deux bases de données. Testons cela rapidement:

```
postgres@debian1:~$ wget -q http://bucardo.org/check_postgres/check_postgres.pl
postgres@debian1:~$ chmod +x check_postgres.pl
postgres@debian1:~$ ./check_postgres.pl --action same_schema \
> --host debian1--dbname base1 --dbuser slony \
> --host2 debian2 --dbname2 base1 --dbuser2 slony \
> --warning "notriger=replication"
POSTGRES_SAME_SCHEMA OK: DB "base1" (host:debian1 => debian2) Les bases de données ont les mêmes éléments | time=0.20
```

Modifions base1 pour qu'il y ait une différence. Nous allons par exemple ajouter une table t3, ajouter un index à t2 et une colonne à t1:

```
postgres@debian1:~$ psql -q base1
base1=# ALTER TABLE t1 ADD COLUMN pascool varchar(255);
base1=# CREATE INDEX beurk ON t2(autre);
base1=# CREATE TABLE t3(id serial, contenu text);
NOTICE:  CREATE TABLE créera des séquences implicites « t3_id_seq » pour la colonne serial « t3.id »
base1=#\q
postgres@debian1:~$ ./check_postgres.pl --action same_schema \
> --host debian1 --dbname base1 --dbuser slony \
> --host2 debian2 --dbname2 base1 --dbuser2 slony \
> --warning "notriger=replication" \
> --verbose
POSTGRES_SAME_SCHEMA CRITICAL: DB "base1" (host:debian1 => debian2) Les bases de données sont différentes. éléments différents : 7 | time=0.17
Table in 1 but not 2: public.t3
Sequence in 1 but not 2: public.t3_id_seq
Trigger in 1 but not 2: _replication_logtrigger_1
Trigger in 1 but not 2: _replication_logtrigger_2
Trigger in 2 but not 1: _replication_denyaccess_1
Trigger in 2 but not 1: _replication_denyaccess_2
Table "public.t1" on 1 has column "pascool", but 2 does not.
```

Notez l'option –verbose permettant d'obtenir les objets, chacun sur une ligne.

En exécutant ce script régulièrement (soit par Nagios, soit par un cron), vous pourrez être averti d'une différence de schéma pouvant être un soucis pour votre réplication.

Comment basculer le maître

Il est possible de faire un switchover comme un failover avec Slony.

Ces deux opérations passent par des instructions Slonik bien qu'il soit possible de passer par des outils plus évolués.

slonik_move_set permettra de faire un switchover alors que slonik_failover, comme son nom l'indique, fera un failover.

Du côté slony1-ctl, il faut utiliser 82_exec_switch.sh pour le switchover et 92_exec_fail.sh pour le failover. Prenons un exemple pour cet outil. Disons que nous voulons basculer le maître sur le nœud 2 bien que le 1 soit toujours vivant:

```
postgres@debian1:~/slony1-ctl/slony-ctl/outils$ ./82_exec_switch.sh -c replibase1 -f 1 -b 2
Bascule de replibase1 : 1 vers 2...
OK
postgres@debian1:~/slony1-ctl/slony-ctl/outils$ psql -q -h debian1 base1
base1=# INSERT INTO t2 (texte, autre) VALUES ('yop2', false);
ERREUR:  Slony-I: Table t1 is replicated and cannot be modified on a subscribed node
base1=# \q
postgres@debian1:~/slony1-ctl/slony-ctl/outils$ psql -q -h debian2 base1
base1=# INSERT INTO t2 (texte, autre) VALUES ('yop2', false);
base1=# SELECT * FROM t2 WHERE texte='yop2';
   id  | texte | autre
-----------------------
 20002 | yop2  | f
(1 ligne)
base1=# \q
postgres@debian1:~/slony1-ctl/slony-ctl/outils$ psql -q -h debian1 base1
base1=# SELECT * FROM t2 WHERE texte='yop2';
   id  | texte | autre
-----------------------
 20002 | yop2  | f
(1 ligne)
```

La bascule est bien faite. Il n'est plus possible de modifier la base sur debian1, et les modifications sur debian2 sont répliquées sur debian1.

Faisons maintenant un failover sur debian1. Commençons par arrêter debian2 (un simple « shutdown -h now » suffit). Puis retournons sur debian1 et utilisons les scripts AltPerl:

```
postgres@debian1:~$ ssh debian2
ssh: connect to host debian2 port 22: No route to host
```

Nous ne pouvons plus nous connecter sur debian2. Nous devons donc basculer le service en urgence sur debian1.

```
postgres@debian1:~/slony1-ctl/slony-ctl/outils$ psql -q -h debian1 base1
base1=# INSERT INTO t2 (texte, autre) VALUES ('yop2', false);
ERREUR:  Slony-I: Table t1 is replicated and cannot be modified on a subscribed node
base1=# \q
```

Ce dernier se considère toujours comme un nœud esclave. Lançons dès maintenant le script Slonik de bascule pour failover:

```
postgres@debian1:~/slony1-ctl/slony-ctl/outils$ cd
postgres@debian1:~$ slonik_failover 2 1 | slonik
<stdin>:4: NOTICE:  failed node has no direct receivers – move now
<stdin>:10: Replication sets originating on 2 failed over to 1
postgres@debian1:~$ psql -q -h debian1 base1
base1=# INSERT INTO t2 (texte, autre) VALUES ('yop2', false);
INSERT 0 1
```

Parfait, le serveur debian1 est maître dès maintenant.

Petit récapitulatif

Avantages majeurs

- simple à mettre en place, mais un peu complexe à appréhender/maîtriser
- très grande granularité
- les esclaves en lecture seule
- mise à jour de PostgreSQL rapide et avec le moins possible d'arrêt de production

Inconvénients majeurs

- pas de réplication automatique des objets
- pas de réplication du TRUNCATE avant la 8.4
- une documentation qui, bien que complète, laisse à désirer

Conclusion

Slony est un outil très intéressant. Il est fiable. Son gros soucis est le travail qu'il demande à l'administrateur. Le moindre oubli d'une table à déclarer, et les données de cette table ne sont pas disponibles sur les esclaves. De plus, certains problèmes, relevant plutôt du détail que d'un problème de conception, entachent cet outil. Il mériterait certainement que ses développeurs s'attachent un peu plus à supprimer les petits détails qui le rendent encore un peu rugueux.

Heureusement, il existe quelques outils qui facilitent la vie, lors de son installation, comme lors de la surveillance de la réplication.