−Table des matières

Pgpool, le pooler multi-tâche

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.

Les programmes de pooling de connexions ont un but simple : supprimer le temps de connexion pour améliorer la rapidité des programmes se connectant très souvent et n'exécutant que peu de requêtes pendant une même connexion. C'est principalement le cas des applications web. Une page PHP va généralement exécuter peu de requêtes tout en étant fréquemment sollicitée. Dans ce cas, le temps de la connexion devient un facteur limitant, surtout pour les applications web devant gérer un très grand nombre d'utilisateurs (et donc de connexions). PostgreSQL dispose de deux solutions. La première est pgPool-II.

Ce logiciel est le successeur de pgPool. Développé au Japon, il est très connu car il a longtemps été le seul outil de ce type. De plus, il ne fait pas que du pooling. Il est aussi capable de faire de la répartition de charge ainsi que de la réplication (par envoi des requêtes sur les différents serveurs).

L'utilisation habituelle de pgPool-II est de conserver les connexions déjà établies pour les ré-utiliser après coup. Le gain peut être vraiment important.

Voici comment PostgreSQL établit une connexion. Un client va se connecter sur le port TCP/IP 5432 du serveur PostgreSQL. Un dialogue s'établit entre le processus serveur et le client. Le but est de réaliser l'authentification du client. Une fois que l'authentification est réussie, le processus postgres père lance un processus fils qui sera chargé de toute la suite du dialogue avec le client, donc principalement de l'exécution des requêtes.

Avec pgPool-II, la connexion se trouve être souvent accélérée. Le client se connecte sur un port TCP/IP du serveur où se trouve pgPool-II. Ce dernier vérifie l'authentification et vérifie dans son pool de connexions existantes s'il s'y trouve une connexion libre mais déjà établie rassemblant un certain nombre de critères: même utilisateur, même base de données, même protocole de connexion. Si c'est le cas, il l'utilise. Du coup, pas besoin de créer un nouveau processus fils. Pas besoin que le serveur PostgreSQL se charge de l'authentification. Cela peut sembler limité mais dans le cas d'une application web, où les connexions sont généralement très nombreuses et très limitées dans le temps, cela peut revêtir une importance considérable. Par contre, si pgPool-II ne dispose pas d'une connexion semblable, il va devoir demander à PostgreSQL d'en créer une nouvelle. Cependant, cette nouvelle connexion va rejoindre le pool une fois qu'elle sera libérée par le client.

La simplification de la connexion n'est pas le seul intérêt de pgPool-II. Quand le nombre de connexions au serveur PostgreSQL dépasse la valeur du paramètre max_connections (voir le fichier postgresql.conf), PostgreSQL refuse tout simplement les nouvelles connexions. PgPool-II n'a pas ce comportement. Au contraire, quand tous les processus pgpool s'occupent déjà d'une connexion au serveur PostgreSQL, il met en attente les nouvelles connexions. Comme l'intérêt de pgPool-II est justement de gérer des sessions très limitées dans le temps, il y a fort à parier que les connexions seront en attente très peu de temps.

Prérequis

Pour le mode pooling de connexions, pgPool-II n'a qu'une seule limitation : la méthode d'authentification sélectionnée. Vous devez utiliser une des méthodes suivantes : trust, password, crypt, md5 et/ou pam. La seule méthode qui vous protège d'une écoute basique est md5. Le choix de la méthode est donc assez simple.

D'autre part, les connexions SSL et IPv6 ne sont pas supportées.

Installation

Cet outil est déjà packagé pour Debian. Cependant, vous pouvez oublier ce paquet dès maintenant. Il correspond à la version 1.3.2 alors que la dernière version de pgpool-II est la 2.2.4. Et les backports ne vous seront d'aucune utilité, étant donné que ce paquet ne s'y trouve pas. Quant aux RPM, vous devriez pouvoir trouver une version récente (la 2.2.2 étant la plus récente que nous avons pu trouver).

Nous allons donc passer par la compilation. Il est tout à fait possible d'utiliser le package source de Debian. Cependant, pour cette fois, nous ne l'utiliserons pas. Les étapes de compilation sont nombreuses, mais tout à fait traditionnelles. Il est à noter que cet outil n'a aucune dépendance, en dehors du paquet développement de PostgreSQL que nous allons installer immédiatement:

```
debian1:~# aptitude install postgresql-server-dev-8.4
```

Maintenant, la compilation/installation de pgPool-II ne doit poser aucun problème.

```
debian1:~# wget -q http://pgfoundry.org/frs/download.php/2362/pgpool-II-2.2.4.tar.gz
debian1:~# tar xfz pgpool-II-2.2.4.tar.gz
debian1:~# cd pgpool-II-2.2.4
debian1:~/pgpool-II-2.2.4# ./configure --prefix=/opt/pgpool-II-2.2
[... messages de progression …]
```

Pour l'étape configure, il existe une option très intéressante si vous avez compilé et installé votre version de PostgreSQL à un emplacement inhabituel. –with-pgsql permet d'indiquer le répertoire d'installation. Dans notre cas, la bibliothèque libpq se trouvant dans un répertoire standard (/usr/lib), nous n'en avons pas besoin.

```
debian1:~/pgpool-II-2.2.4# make
[... messages de progression ...]
debian1:~/pgpool-II-2.2.4# make install
[... messages de progression …]
```

Configuration

Des exemples de fichiers de configuration doivent se trouver dans le sous-répertoire etc du répertoire d'installation (/usr/local/etc si vous n'avez pas utilisé l'option –prefix). Nous allons préparer le fichier de configuration du processus pgpool.

```
debian1:~/pgpool-II-2.2.4# cd /opt/pgpool-II-2.2.4/etc/
debian1:/opt/pgpool-II-2.2.4/etc# cp pgpool.conf.sample pgpool.conf
```

La configuration de base de pgPool-II active le mode pooling de connexions (paramètre connection_cache). Il attend les connexions sur l'interface localhost, port 9999, pour ce qui concerne TCP/IP. Pour les sockets de domaine Unix, le répertoire /tmp est utilisé.

Généralement, on modifiera l'interface d'écoute 'localhost' par toutes les interfaces (donc '*'). Il s'agit du paramètre listen_addresses (oui, le même que pour le fichier postgresql.conf). Néanmoins, si le serveur web et pgpool se trouvent sur le même serveur, il n'est pas nécessaire de modifier ce paramètre.

Il est généralement intéressant de modifier le numéro du port par 5432. C'est le numéro du port de connexion de PostgreSQL. Si les deux serveurs sont sur la même machine, il faudra alors bien faire attention à modifier le port de connexion de PostgreSQL. Le gros intérêt de cette manœuvre est de faire en sorte que la chaîne de connexion des clients ne soit pas à modifier du fait que nous ajoutons un pooler de connexions. Cela permet de rendre transparent l'utilisation de cet outil.

Enfin, pour les connexions par socket de domaine Unix, le paramètre socket_dir vaut par défaut /tmp. Par défaut, les outils PostgreSQL sous Debian s'attendent à le trouver dans /var/run/postgresql. Il peut donc être intéressant de changer /tmp par ce chemin.

Maintenant que nous avons configuré la façon dont les clients allaient se connecter à pgPool-II, nous devons indiquer comment ce dernier peut se connecter au serveur PostgreSQL. Pour les connexions TCP/IP, vous devez renseigner deux paramètres:

- backend_hostname0, pour indiquer l'alias ou l'adresse IP du serveur (localhost par défaut);
- backend_port0, pour indiquer le port de connexion (5432 par défaut).

Il existe une troisième valeur ayant le même préfixe backend_, à savoir backend_weight0. Elle n'a aucune importance dans le cadre strict du pooling de connexions. Cette variable a un intérêt lorsqu'on met en place la répartition de charge. Vous remarquerez qu'il est aussi possible de définir d'autres serveurs. La même remarque s'applique : aucun intérêt pour le pooling, mais essentiel pour la répartition de charge.

Quant aux sockets de domaine Unix, il s'agit du paramètre backend_socket_dir. Il vaut encore une fois /tmp. Sous Debian, si pgPool-II et PostgreSQL se trouvent sur le même serveur, vous devez le modifier par /var/run/postgresql.

Pour résumer, nous n'allons modifier que cinq paramètres pour l'instant:

```
socket_dir = '/var/run/postgresql'
backend_socket_dir = '/var/run/postgresql'
backend_hostname0 = 'localhost'
backend_port0 = 5432
log_connections = true (pour avoir quelques traces supplémentaires)
```

pgPool-II va faire la connexion sur l'interface TCP/IP locale. Or, sous Debian, le fichier de configuration pg_hba.conf demande une authentification par mot de passe chiffré avec md5. Nous allons donc donner un mot de passe à l'utilisateur postgres:

```
postgres@debian1:~$ psql -c "ALTER USER postgres PASSWORD 'postgres'"
ALTER rôle
```

C'est évidemment un exemple. Nous vous conseillons de mettre un mot de passe bien plus fort pour cet utilisateur.

Dernière information d'importance. Au lancement de pgpool, ce dernier va créer un fichier PID. Le répertoire par défaut est /var/run/pgpool. Au cas où ce dernier n'existe pas, il est nécessaire de le créer soi-même:

```
debian1:/opt/pgpool-II-2.2.4/etc# mkdir /var/run/pgpool
```

Si vous ne souhaitez pas créer ce répertoire, vous pouvez aussi modifier cet emplacement grâce au paramètre `pid_file_name`.

Lancement

Le paquet Debian a déjà installé le script de démarrage. Pour ceux qui ont compilé eux-même pgPool-II, il existe un script de démarrage dans le sous-répertoire redhat du répertoire des sources. Il est nommé pgpool.init. Le plus simple est de copier ce fichier dans le répertoire /etc:

```
debian1:~$ cp redhat/pgpool.init /etc/init.d/pgpool
```

Ensuite, il faut l'activer:

```
debian1:~$ update-rc.d pgpool defaults
```

Si vous préférez faire plus simple dans un premier temps, vous pouvez aussi le lancer manuellement:

```
debian1:~$ pgpool -n -d > /tmp/pgpool.log 2>&1 &
```

Cette commande l'exécute en mode démon, avec une activation du débogage.

Utilisation

Si nous essayons de nous connecter à PostgreSQL via le port 9999 sans avoir lancé pgPool-II, voici ce que nous allons obtenir:

```
postgres@debian1:~$ psql -p 9999 postgres
psql: n'a pas pu se connecter au serveur : Aucun fichier ou dossier de ce type
        Le serveur est-il actif localement et accepte-t-il les connexions sur la
        socket Unix « /var/run/postgresql/.s.PGSQL.9999 » ?
```

Lançons donc pgPool-II:

```
debian1:~# cd /opt/pgpool-II-2.2.4/bin
debian1:/opt/pgpool-II-2.2.4/bin# ./pgpool -n
2009-08-27 02:43:38 LOG:   pid 1469: pgpool successfully started
```

Vérifions que le processus pgpool est bien lancé:

```
debian1:~# ps axjf | grep pgpool
 4900  1469  1469  4900 pts/0    1469 S+       0   0:00 |      \_ ./pgpool -n
 1469  1470  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1471  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1472  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1473  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1474  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1475  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1476  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1477  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1478  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1479  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1480  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1481  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1482  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1483  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1484  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1485  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1486  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1487  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1488  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1489  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1490  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1491  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1492  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1493  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1494  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1495  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1496  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1497  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1498  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1499  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1500  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1501  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: wait for connection request
 1469  1502  1469  4900 pts/0    1469 S+       0   0:00 |         \_ pgpool: PCP: wait for connection request
```

Ça fait du monde. « pgpool -n » est le processus père de tous les autres. Il y a 32 processus pgpool attendant une connexion d'un utilisateur et 1 processus pgpool PCP attendant lui-aussi la connexion d'un utilisateur. Ceci est tout à fait normal. Pgpool a lancé en avance un certain nombre de processus pgpool. Lors d'une connexion, un des processus pgpool en attente récupère la connexion et s'occupe de faire la passerelle pour le dialogue entre le client initial et le serveur PostgreSQL.

Créons quelques bases de données:

```
postgres@debian1:~$ for i in $(seq 1 9)
> do
> createdb b$i
> done
```

Attention à ne pas faire vos tests avec les bases template0, template1, postgres et regression. Ces quatre bases sont gérées séparément des autres car elles ne sont jamais prises en compte par les pools. La déconnexion d'un client de la base postgres fera que la connexion entre pgPool-II et PostgreSQL soit supprimée.

Connectons-nous à la base b1:

```
postgres@debian1:~$ psql -p 9999 b1
psql (8.4.0)
Saisissez « help » pour l'aide.

b1=#
```

Ça fonctionne. Les traces indiquent fièrement:

```
2009-08-27 02:43:40 LOG:   pid 1470: connection received: host=[local]
```

et les processus indiquent bien la connexion:

```
debian1:~# ps -ef | grep "[p]ostgres b1"
root      1470  1469  0 02:43 pts/0    00:00:00 pgpool: postgres b1 [local] idle
postgres  1504  3780  0 02:43 ?        00:00:00 postgres: postgres b1 127.0.0.1(54535) idle
```

La première correspond au processus pgpool chargé de la communication entre le client (qui accède via la socket de domaine Unix) et le processus

postgres. Le processus pgpool a établi la connexion, via TCP/IP (comme lui demandait le paramètre backend_hostname0).

Maintenant déconnectons-nous:

```
b1=# \q
postgres@debian1:~$
```

Parfait. Vérifions que le processus postgres est bien toujours là, alors que le processus pgpool, bien que toujours présent, est en attente d'une connexion.

```
debian1:~# ps -ef | grep "[p]ostgres b1"
postgres  1504  3780  0 02:43 ?        00:00:00 postgres: postgres b1 127.0.0.1(54535) idle
debian1:~# ps -ef | grep  1470
root      1470  1469  0 02:43 pts/0    00:00:00 pgpool: wait for connection request
```

Donc, bien que mon client se soit déconnecté, la connexion est toujours disponible. Essayons de nous connecter de nouveau sur b1:

```
debian1:~# ps -ef | grep "[p]ostgres b1"
root      1470  1469  0 02:43 pts/0    00:00:00 pgpool: postgres b1 [local] idle
postgres  1504  3780  0 02:43 ?        00:00:00 postgres: postgres b1 127.0.0.1(54535) idle
```

Nous utilisons toujours le même processus pgpool et la même connexion à PostgreSQL. Maintenant, connectons-nous sur plusieurs bases:

```
debian1:~# ps -ef | grep "[p]ostgres b"
root      1475  1469  0 02:43 pts/0    00:00:00 pgpool: postgres b2 [local] idle
root      1477  1469  0 02:43 pts/0    00:00:00 pgpool: postgres b4 [local] idle
root      1478  1469  0 02:43 pts/0    00:00:00 pgpool: postgres b3 [local] idle
root      1479  1469  0 02:43 pts/0    00:00:00 pgpool: postgres b1 [local] idle
postgres  1504  3780  0 02:43 ?        00:00:00 postgres: postgres b1 127.0.0.1(54535) idle
postgres  1541  3780  0 02:50 ?        00:00:00 postgres: postgres b1 127.0.0.1(57197) idle
postgres  1543  3780  0 02:50 ?        00:00:00 postgres: postgres b2 127.0.0.1(57198) idle
postgres  1545  3780  0 02:50 ?        00:00:00 postgres: postgres b3 127.0.0.1(57199) idle
postgres  1548  3780  0 02:50 ?        00:00:00 postgres: postgres b4 127.0.0.1(57200) idle
```

Chaque connexion est réalisée par un processus pgpool indépendant.

Arrêtons pgpool et diminuons le nombre de processus pgpool. Pour cela, nous devons modifier le paramètre num_init_children pour obtenir cette configuration:

```
num_init_children = 1
```

Relançons pgpool:

```
debian1:/opt/pgpool-II-2.2.4/bin# ./pgpool -n
2009-08-27 03:04:35 LOG:   pid 1625: pgpool successfully started
```

Vérifions que le nombre de processus pgpool en attente de connexion est bien de 1:

```
postgres@debian1:~$ ps axjf | grep [p]gpool
 4900  1625  1625  4900 pts/0     1625 S+      0   0:00  |      \_ ./pgpool -n
 1625  1626  1625  4900 pts/0     1625 S+      0   0:00  |         \_ pgpool: wait for connection request
 1625  1627  1625  4900 pts/0     1625 S+      0   0:00  |         \_ pgpool: PCP: wait for connection request
```

Parfait. Connectons-nous à b1, puis déconnectons-nous immédiatement de cette base:

```
postgres@debian1:~$ psql -p 9999 b1
psql (8.4.0)
Saisissez « help » pour l'aide.

b1=# \q
postgres@debian1:~$ ps -ef | grep "[p]ostgres b"
postgres  1668  3780  0 03:07 ?        00:00:00 postgres: postgres b1 127.0.0.1(32994) idle
```

La connexion sur db1 est toujours conservée. Faisons de même avec la base db2:

```
postgres@debian1:~$ psql -p 9999 b2
psql (8.4.0)
Saisissez « help » pour l'aide.

b2=# \q
postgres@debian1:~$ ps -ef | grep "[p]ostgres b"
postgres  1668  3780  0 03:07 ?        00:00:00 postgres: postgres b1 127.0.0.1(32994) idle
postgres  1676  3780  0 03:08 ?        00:00:00 postgres: postgres b2 127.0.0.1(34518) idle
```

Maintenant, nous avons une connexion sur b1 et b2. Faisons de même avec les bases b3 et b4:

```
postgres@debian1:~$ psql -p 9999 b3
psql (8.4.0)
Saisissez « help » pour l'aide.

b3=# \q
postgres@debian1:~$ psql -p 9999 b4
psql (8.4.0)
Saisissez « help » pour l'aide.

b4=# \q
postgres@debian1:~$ ps -ef | grep "[p]ostgres b"
postgres  1668  3780  0 03:07 ?        00:00:00 postgres: postgres b1 127.0.0.1(32994) idle
postgres  1676  3780  0 03:08 ?        00:00:00 postgres: postgres b2 127.0.0.1(34518) idle
postgres  1681  3780  0 03:08 ?        00:00:00 postgres: postgres b3 127.0.0.1(34519) idle
postgres  1686  3780  0 03:08 ?        00:00:00 postgres: postgres b4 127.0.0.1(34520) idle
```

Nous arrivons à quatre connexions PostgreSQL, tout ça pour un seul processus pgpool. Essayons maintenant avec b5:

```
postgres@debian1:~$ psql -p 9999 b5
```

```
psql (8.4.0)
Saisissez « help » pour l'aide.

b5=# \q
postgres@debian1:~$ ps -ef | grep "[p]ostgres b"
postgres  1676  3780  0 03:08 ?        00:00:00 postgres: postgres b2 127.0.0.1(34518) idle
postgres  1681  3780  0 03:08 ?        00:00:00 postgres: postgres b3 127.0.0.1(34519) idle
postgres  1686  3780  0 03:08 ?        00:00:00 postgres: postgres b4 127.0.0.1(34520) idle
postgres  1691  3780  1 03:08 ?        00:00:00 postgres: postgres b5 127.0.0.1(34521) idle
```

Nous constatons que la connexion sur b1 a été supprimée, et que celle sur b5 a été conservée. L'explication en est simple. Chaque processus pgpool va gérer un ensemble (appelé généralement pool) de quatre connexions à la base de données. Seule une d'entre elles pourra être active jusqu'à la déconnexion du client. Si le client se connecte toujours avec le même triplé utilisateur / base de données / protocole de connexion, il trouvera toujours sa connexion prête à être utilisé. Dans le cas contraire, il utilisera une autre connexion du pool. Si le pool est déjà complet, la connexion inactive la plus ancienne est supprimée et la nouvelle connexion peut se faire. Maintenant, ouvrons une connexion sur b5 sans la fermer et tentons d'ouvrir une autre connexion:

```
postgres@debian1:~$ psql -p 9999 b6
```

Et oui, cette connexion est bloquée.

```
postgres@debian1:~$ ps -ef | grep "[p]ostgres b"
root      1657  1469  0 23:14 pts/0    00:00:00 pgpool: postgres b5 [local] idle
postgres  1676  3780  0 03:08 ?        00:00:00 postgres: postgres b2 127.0.0.1(34518) idle
postgres  1681  3780  0 03:08 ?        00:00:00 postgres: postgres b3 127.0.0.1(34519) idle
postgres  1686  3780  0 03:08 ?        00:00:00 postgres: postgres b4 127.0.0.1(34520) idle
postgres  1691  3780  1 03:08 ?        00:00:00 postgres: postgres b5 127.0.0.1(34521) idle
```

Remarquez bien que la connexion n'est pas refusée. Elle est simplement en attente. Dès que le client sur b5 se déconnectera, la connexion sera disponible pour le client qui veut aller sur b6.

Si on essaie de résumer, il y a au maximum num_init_children processus pgpool. Chacun d'entre eux peut ouvrir au maximum max_pool connexions au serveur. Du coup, nous avons un maximum de num_init_children * max_pool connexions à un temps t. Cependant, seules num_ini_children connexions peuvent être actives à un instant t. Toute demande de connexion supplémentaire sera placée en attente. Chaque client qui se déconnecte permettra à la première connexion en attente de se faire. Si quelqu'un essaie de se connecter et que le processus pgpool n'a plus de place dans son pool, la plus ancienne connexion est dégagée.

Pour une configuration plus poussée

Nous n'avons pas encore vu les paramètres avancées. Ils permettent de mieux contrôler le comportement de pgPool-II.

Au lancement de pgPool-II, ce dernier, à l'instar d'Apache, lance un certain nombre de processus fils. Ce nombre dépend directement du paramètre num_init_children.

```
$ ps -ef | grep "[p]gpool: wait for connection request" | wc -l
32
```

En augmentant ce nombre, vous augmentez le nombre de processus exécutés au démarrage de pgPool-II, qui correspond aussi au nombre maximum de connexions actives à un instant t sur le serveur PostgreSQL.

max_pool indique le nombre maximum de connexions mises en cache par un processus pgpool. La valeur par défaut étant de 4, vous aurez donc un maximum de 4*32 (soit 128) connexions simultanées au serveur PostgreSQL, avec seulement 32 réellement actives.

Comme les processus pgpool peuvent rester en exécution assez longtemps, les développeurs de pgPool-II ont ajouté deux variables permettant de limiter leur durée de vie. child_life_time va permettre d'indiquer le nombre maximum de secondes d'inactivité avant de tuer un processus pgpool. Si jamais le système est trop occupé pour que ce nombre de secondes ne soit jamais dépassé, un autre paramètre va permettre de tuer un processus pgpool : child_max_connections. Un processus pgpool ayant reçu ce nombre de connexions sera automatiquement arrêté, pour être remplacé par un nouveau processus. Ainsi, en permettant un arrêt des processus après une certaine activité, il est possible de se prémunir contre des risques comme les pertes mémoire.

client_idle_limit permet d'arrêter une connexion si le client a été inactif pendant ce nombre de secondes. Ceci est valable même si le client se trouve au milieu d'une transaction, ce qui peut être assez dangereux. Imaginez que vous avez commencé une transaction, fait un certain nombre d'insertions, modifications et/ou suppressions. Vous êtes parti déjeuner sans valider la transaction. Et hop, au retour du déjeuner, vous vous apercevez que la connexion a été stoppée, annulant du coup votre transaction. Cet exemple est évidemment un peu grossier, mais il aide à comprendre pourquoi cette option doit être désactivée. Elle l'est par défaut et nous vous recommandons de la laisser ainsi. Il est préférable d'utiliser connection_life_time. Ce paramètre permet de spécifier la durée de vie d'une connexion. La déconnexion ne survient que si cette période s'est écoulée alors que la connexion au serveur PostgreSQL était inactive.

Comme dans la configuration de PostgreSQL, il est aussi possible de configurer un délai pour l'authentification avec le paramètre authentication_timeout. Sa valeur par défaut est de 60 secondes, tout comme sur PostgreSQL.

Comme plusieurs clients peuvent ré-utiliser la même connexion, il est important que le client précédent n'ait pas fait de modification spécifique à la session. Cela peut être un curseur, une table temporaire (qui n'est supprimée qu'à la fin de la session), voire même la modification d'un paramètre de configuration. Pour éviter cela, pgPool-II va exécuter une requête SQL lors de la déconnexion d'un client pour réinitialiser la session. Cette requête est précisée par le paramètre reset_query_list. D'ailleurs, il ne s'agit pas forcément d'une seule requête. Vous pouvez en indiquer plusieurs à condition de les séparer par des points-virgules. La valeur par défaut dépend de la version de PostgreSQL: pour les versions antérieures à la 8.3, vous devez utiliser : ABORT; RESET ALL; SET SESSION AUTHORIZATION DEFAULT pour les versions 8.3 et ultérieures, vous pouvez utiliser : ABORT; DISCARD ALL

Le dernier paramètre particulièrement important est log_connections. Ce dernier vous permet d'enregistrer dans les traces de pgpool chaque connexion d'un client. C'est une aide très appréciable au début de l'utilisation de l'outil.

Filtrer les connexions par rapport aux adresses réseau

Si vous avez correctement configuré votre serveur PostgreSQL, vous avez dû passer un bon moment sur le fichier pg_hba.conf. Ce fichier sert à indiquer quel utilisateur a droit à se connecter à quelle base, en passant par telle adresse IP ou sous-réseau.

En utilisant pgPool-II, pour PostgreSQL, toutes les connexions ont pour origine le serveur pgPool-II. Du coup, toute la jolie configuration du pg_hba.conf que vous aviez patiemment affinée n'a plus d'intérêt.

Heureusement, pgPool-II propose une solution à ce problème. Le filtrage des accès par leur adresse IP d'origine devra se faire par une configuration de pgPool-II. Le fichier ne se nomme plus pg_hba.conf, mais pool_hba.conf. Le contenu est pratiquement identique. La meilleure chose à faire est de copier votre fichier pg_hba.conf en pool_hba.conf. Ensuite, vérifiez que votre configuration n'entre pas en conflit avec les limitations suivantes:

- comme les connexions SSL ne sont pas supportés, le type de connexion hostssl ne peut pas être utilisé;
- de même, IPv6 n'étant pas supporté, les adresses de type IPv6 ne peuvent pas être utilisées;
- l'option samegroup n'est pas supportée pour la colonne des bases de données (en effet, pgPool-II ne connaît pas les utilisateurs et les groupes disponibles dans l'instance PostgreSQL… du coup, il ne peut pas en déduire le nom de la base de connexion);
- les groupes suivis d'un + dans la colonne des utilisateurs ne sont pas supportées pour la même raison;
- les méthodes d'authentification trust, reject et pam sont les seules autorisées.

Ce dernier point est vraiment le gros inconvénient de l'utilisation du fichier pool_hba.conf.

Quelques informations supplémentaires

Assurez-vous que la valeur du paramètre max_connections de votre serveur PostgreSQL soit bien supérieure à la multiplication des paramètres num_init_children et max_pool de pgPool-II. Dans le cas contraire, vous aurez, dans les traces du serveur PostgreSQL, des messages de type FATAL indiquant que certaines connexions ont échouées. Notez bien que les connexions supplémentaires du côté pgPool-II sont mises en attente, et non pas simplement abandonnées.

Un autre point est à considérer si vous voulez pouvoir annuler des reqsuêtes. L'annulation d'une requête va générer une nouvelle connexion au serveur. Du coup, si tous les pools annoncent qu'ils sont complets, la requête ne peut pas être annulée. Pour s'assurer de la possibilité d'annuler une requête, il faut doubler le nombre de connexions attendues au niveau du paramètre max_connections du serveur PostgreSQL.

Conclusion

PgPool-II est un excellent outil, bien maintenu et disposant d'une documentation de plutôt bonne qualité. Son seul inconvénient est son côté multi-tâches lorsqu'on a uniquement besoin d'un pooler de connexions. En effet, l'empreinte mémoire est du coup plus importante, les possibilités de bugs plus nombreuses. Donc on tâchera plutôt de conserver cet outil si on veut utiliser aussi d'autres modes (comme la répartition de charge ou la réplication). Dans les autres cas, il peut être intéressant de jeter un œil à un autre outil, pgbouncer.