Introduction à la réplication de bases de données

Etre capable de faire de la haute disponibilité de machines est une chose qui présente de nombreux avantages. Malheureusement, elle ne résout pas à elle seule la disponibilité des services hébergés, et la solution peut parfois être fort complexe dans le cadre des bases de données. Ceci est dû au fait que, contrairement à un serveur Web, un site FTP ou tout autre service, il s'agit de gérer un jeu de données qui peuvent être modifiées à fréquence élevée.

Définition

Une application de base de données repose sur un modèle client-serveur. Dans ce modèle, le client se connecte au SGBD pour passer des ordres. Ces ordres sont de deux natures : lecture (on parle alors de requêtes) ou mise à jour (on parle alors de transactions).

Pour les transactions, il y a une modification des données sur le serveur, mais cela reste des ordres de courte durée. A l'inverse, dans le cas d'une lecture, il n'y a pas de modification des données mais les traitements peuvent être longs et porter sur une grande masse de données.

On comprend donc aisément que, dans le cadre d'un site Web par exemple, un nombre important de requêtes peut emboliser partiellement (ou complètement) le serveur. Il existe plusieurs solutions pour pallier ce genre de problèmes et, ça tombe bien, la réplication en est une ;).

L'objectif principal de la réplication est de faciliter l'accès aux données en augmentant la disponibilité. Soit parce que les données sont copiées sur différents sites permettant de répartir les requêtes, soit parce qu'un site peut prendre la relève lorsque le serveur principal s'écroule.

Une autre application tout aussi importante est la synchronisation des systèmes embarqués non connectés en permanence.

Ce qui peut se résumer à l'aide des trois types de scénarii suivants :

- Deux serveurs distants sur lesquels les données doivent être consistantes ;
- Deux serveurs, un comme serveur principal, l'autre comme serveur de *backup* à chaud ;
- Plusieurs serveurs en cluster utilisés pour de l'équilibrage de charge et de la tolérance à la panne.

Principe

Le principe de la réplication, qui met en jeu au minimum deux SGBD, est assez simple et se déroule en trois temps :

1. La base maître reçoit un ordre de mise à jour (INSERT, UPDATE ou DELETE).

2. Les modifications faites sur les données sont détectées et stockées (dans une table, un fichier, une queue) en vue de leur propagation.

3. Un processus de réplication prend en charge la propagation des modifications à faire sur une seconde base dite esclave. Il peut bien entendu y avoir plus d'une base esclave.

Bien entendu, il est tout à fait possible de faire de la réplication dans les deux sens (de l'esclave vers le maître et inversement).

On parlera dans ce cas-là de réplication bidirectionnelle ou symétrique. Dans le cas contraire, la réplication est unidirectionnelle (seulement du maître vers l'esclave), et on parle de réplication en lecture seule ou asymétrique.

De plus, la réplication peut être faite de manière synchrone ou asynchrone. Dans le premier cas, la résolution des conflits éventuels entre deux sites intervient avant la validation des transactions ; dans le second cas, la résolution est faite dans des transactions séparées. Il est donc possible d'avoir quatre modèles de réplication :

- Réplication asymétrique (maître/esclave) avec propagation asynchrone ;
- Réplication asymétrique (maître/esclave) avec propagation synchrone ;
- Réplication symétrique ou *peer-to-peer* (*update everywhere*) avec propagation asynchrone ;
- Réplication symétrique avec propagation synchrone.

La mise à jour synchrone

La réplication synchrone est aussi appelée "réplication en temps réel". La synchronisation est effectuée en temps réel puisque chaque requête est déployée sur l'ensemble des bases de données avant la validation (*commit*) de la requête sur le serveur où la requête est exécutée.

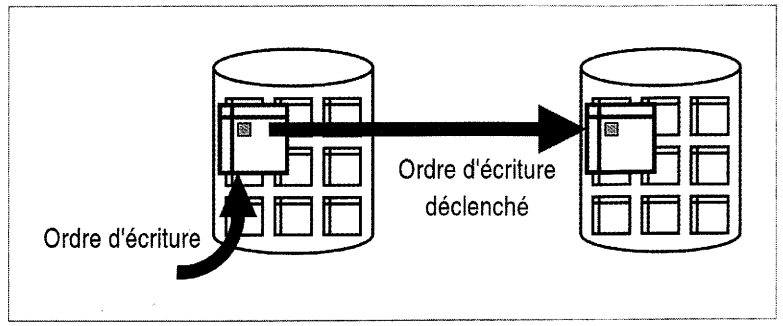
Ce type de réplication assure un haut degré d'intégrité des données mais requiert une disponibilité permanente des serveurs et de la bande passante. Ce type de réplication, fortement dépendant des pannes du système, nécessite de gérer des transactions multisites coûteuses en ressources.

La réplication synchrone asymétrique

Une modification sur le site primaire sera propagée aux sites secondaires à l'aide par exemple d'un *trigger* sur la table modifiée.

La table est modifiée en temps réel sur les autres sites, ces modifications faisant partie de la transaction.

Si le site distant est victime d'une panne, l'absence de synchronisation n'empêche pas la consistance de la base maître.



La réplication synchrone symétrique

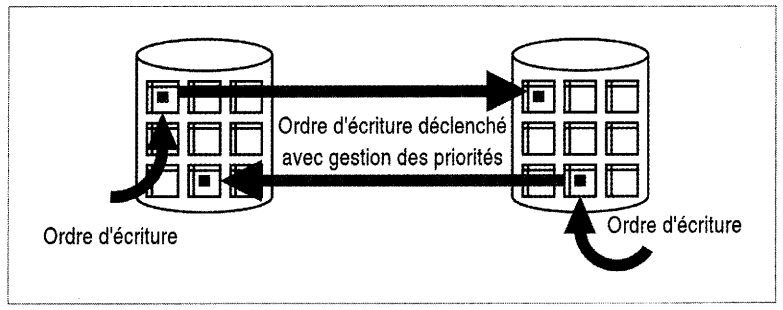
Dans ce cas, il n'y a pas de table maître, mais chaque table possède ses triggers, déclenchés lors d'une modification.

Il est alors nécessaire de définir des priorités et de gérer les blocages des tables en attendant qu'une modification soit entièrement propagée. D'autre part, les triggers doivent différencier les mises à jour issues d'une réplication des mises à jour de requête directes.

De même, une panne de réseau laissera les deux bases, maître et esclave, dans des états de consistance.

Les opérations sur les données sont plus rapides, puisqu'une requête n'est pas immédiatement déployée. Le trafic sur le réseau est de ce fait plus compact.

Par contre, le planning de réplication est dans ce cas plus complexe, puisqu'il s'agit de gérer les conflits émanant d'un éventuel accès en écriture sur une base esclave entre deux mises à jour.



La mise à jour asynchrone

La réplication asynchrone (aussi appelée "*Store and Forward*" pour "stocker et propager") stocke les opérations intervenues sur une base de données dans une queue locale pour les propager plus tard à l'aide d'un processus de synchronisation.

Ce type de réplication est plus flexible que la réplication synchrone. Il permet en effet de définir les intervalles de synchronisation, ce qui permet à un site de fonctionner même si un site de réplication n'est pas accessible.

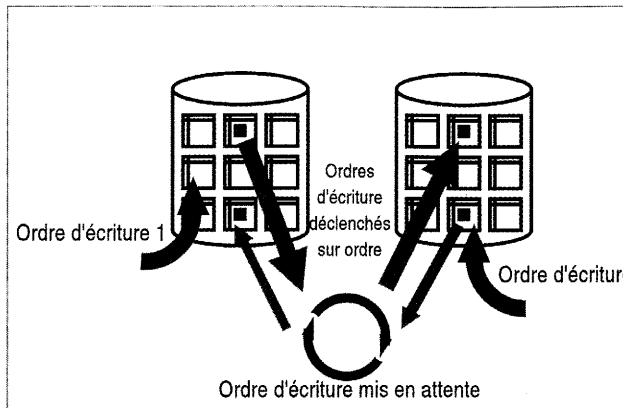
L'utilisateur peut ainsi déclarer que la synchronisation sera effectuée la nuit, à une heure de faible affluence.

La réplication asynchrone asymétrique

Les mises à jour sont stockées dans une file d'attente et ne seront propagées que lors d'un déclenchement programmé.

La réplication asynchrone symétrique

Toute modification sur toute table de toute base est stockée dans une file pour être rejouée ultérieurement. De fortes incohérences des données sont à craindre.



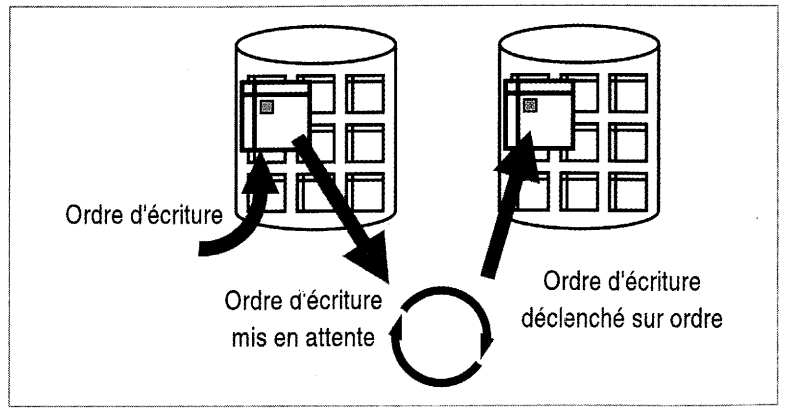
En théorie et en pratique

Dans les deux articles suivants, nous allons vous présenter les solutions existantes pour faire de la réplication avec PostgreSQL avant d'aborder la pratique avec MySQL.

Sachez cependant que la réplication est possible avec n'importe quelle base (libre ou non), mais que le prix de sa mise en place varie énormément d'un système à l'autre.

Stéphane Schildknecht
dbadialog@free.fr

Grégoire Lejeune
greg@webtime-project.net



La haute disponiblité sous PostgreSQL

Dans cet article nous dresserons un inventaire des solutions de réplication sous PostgreSQL, mais nous aborderons aussi deux solutions permettant d'améliorer les connexions aux bases de données. Nous avons pris le parti d'ignorer les autres bases de données et leurs systèmes de réplication pour la simple raison que, sous MySQL, comme sous SAPDB, la réplication est une possibilité offerte nativement par la base de données. Deux ou trois options de configuration et le tour est joué ;-). La réplication au sein de MySQL fera l'objet de l'article suivant.

Les solutions de réplication sous PostgreSQL

Plusieurs solutions de réplication de bases PostgreSQL sont actuellement à l'étude. La page officielle regroupant ces projets peut être consultée à l'URL :

http://gborg.postgresql.org/genpage?replication_research

Parmi les projets principaux, on trouve :

- Postgres-R
- DRAGON
- DBBalancer
- PostgreSQL-Replicator
- eRServer
- DBMirror
- Usogres

Projets

La principale difficulté à la mise en place d'une solution de réplication est l'absence d'une telle solution en natif au sein du projet PostgreSQL, contrairement aux projets SAPDB ou MySQL (si nous ne considérons que les projets OpenSource d'importance).

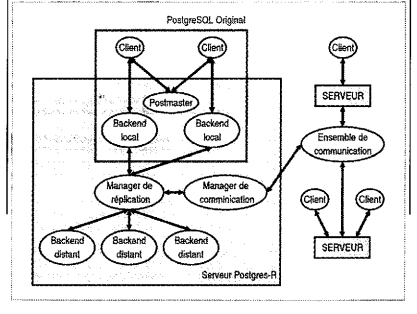
Nous présenterons succinctement chacun des projets précités auxquels nous ajouterons deux ou trois projets d'équilibrage de charge.

Postgres-R

Le projet Postgres-R a été initié par le "groupe des systèmes d'information et de communication" de l'ETH, à Zürich, en Suisse. Postgres-R est basé sur une approche synchrone. Le modèle de travail, servant à valider le principe, est basé sur une version 6.4.2 de PostgreSQL, l'intégration de Postgres-R devrait être réalisable avec la version 7.2 de PostgreSQL. Ce projet n'a toujours pas de version stable, n'étant encore qu'à l'état de prototype.

Principe

Postgres-R a été développé comme une extension à PostgreSQL, version 6.4.2. Ainsi, les fonctionnalités de PostgreSQL sont toujours présentes dans Postgres-R, mais avec en plus la prise en compte de la réplication. Postgres-R peut faire de la réplication complète ou partielle, mais Postgres-R ne permet de répliquer qu'une seule base par instance.



Fonctionnement

Fonctionnement de Postgres-R (d'après le schéma paru dans l'article *Don't be lazy, be consistent: Postgres-R, a new way to implement Database Replication*, B. Kemme et G. Alonso, "Proc. of the 26th International Conference on Very Large Databases (VLDB)", Cairo, Egypt, September 2000.)

Comme montré sur la figure précédente, une base répliquée comprend plusieurs serveurs, chacun gérant une instance de Postgres-R.

PostgreSQL est basé sur les processus. Quand un client veut se connecter, il envoie une requête au *postmaster* (le *listener*). Celui-ci va créer un processus de *backend* pour chaque client et démarrer une communication entre le backend et le client. Pour mettre en place la réplication, il faut ajouter quelques modules de façon à gérer la communication et à prendre en compte les transactions distantes.

Contrairement à PostgreSQL, où un client est associé à un backend particulier, Postgres-R permet à un client de se connecter à n'importe quel serveur du groupe, chacun gérant un *pool* de processus de backend distants.

Le contrôle du protocole de réplication est effectué par le "manager de replication". C'est lui qui reçoit les messages émanant des backends locaux et distants et qui transmet ces messages aux autres sites.

C'est également lui qui reçoit les messages délivrés par le système de communication et qui les transmet aux backends correspondants.

Lorsque les transactions sont acceptées en local, un ensemble d'écriture est envoyé au gestionnaire de réplication. C'est un ensemble d'échanges entre le gestionnaire de réplication et les différents backends qui permet de gérer les *locks* et les écritures.

Contacts

Plus d'informations sont disponibles sur le site du projet : **http://gborg. postgresql.org/project/pgreplication/projdisplay.php**.

DRAGON : Database Replication based on Group Communication

Tout comme le projet Postgres-R, DRAGON est plus un projet de recherche théorique qu'un programme abouti de réplication. On y retrouve des personnes ayant pris part ensuite au développement de Postgres-R. Il semblerait que DRAGON ait été antérieur à Postgres-R. Peu d'information sont disponibles concernant ce système. Il y est fait référence dans cet article dans un seul but d'exhaustivité, parce que ce projet est cité comme projet officiel. Les derniers écrits remontent cependant à mai 1998.

DBBalancer

DBBalancer, qui se veut un pool de connexions, un système d'équilibrage de charge, et un système de réplication est bloqué à la version 0.4.1-1alpha depuis décembre 2001 pour la version binaire et à la version 0.4.4 de septembre 2002 pour la version compilable. Ce projet, qui ne gère pas la réplication asynchrone, est très simpliste et suffisamment peu sécurisé pour ne pas être utilisé en production.

Connection pool

Cette fonctionnalité opère selon le principe de tous les pools de connexions. Elle réserve un certain nombre de connexions ouvertes de façon à accélérer les requêtes à une base de données. Un démon de lecture est activé, qui *dispatche* les requêtes sur les différents serveurs.

Equilibrage de charge

L'équilibrage de charge est un des aspects de la haute disponibilité en matière de bases de données. Or, pour avoir la possibilité de partager des connexions entre différents serveurs, il était nécessaire de développer la troisième fonctionnalité de façon à avoir des bases de données cohérentes entre les différents serveurs utilisés.

Réplication

Cette fonction permet à un client d'interroger la base de données comme s'il ne voyait qu'un serveur, alors qu'en réalité, c'est à un *cluster* qu'il s'adresse. Ce fonctionnement a un coût lors d'écritures dans la base de données, mais représente un gain considérable lors de requêtes en lecture, ce qui signifie la majorité des requêtes sur une application Web. Le principe consiste à utiliser deux démons, un de lecture, un d'écriture. Le démon de lecture servira à dispatcher les requêtes de lecture, mais celui d'écriture devrait servir à dupliquer la même requête d'écriture sur les tous sites. Cela sans gestion des erreurs, ce qui peut amener à une perte complète des données en cas d'erreur d'écriture. De plus, aucune cohérence des données n'est assurée entre les différents sites.

Contacts

La page d'accueil du projet se trouve sur **http://dbbalancer.sourceforge.net**. Daniel Varela, administrateur du projet peut être joint à l'adresse **xperience@users.sourceforge.net**.

PostgreSQL Replicator ou pgReplicator

PostgresSQL Replicator est un système de réplication asynchrone. Il permet en outre la gestion des modèles de propriété des données et comporte des algorithmes de résolution des conflits.

La version actuelle, numérotée 1.1.0, date du 15 octobre 2001. Cette version a vu apparaître la réplication des LO (*Large Objects*), le support des contraintes d'intégrité, une alerte par mél lors d'une panne de réplication, un script de démarrage pour Debian, et la correction de bogues inhérente à toute nouvelle *release*.

Si les auteurs continuent d'utiliser ce projet en interne, le développement est effectivement temporairement interrompu.

Fonctionnalités

Comme dit précédemment, pgReplicator est un système de réplication asynchrone. Il ne nécessite aucun patchage de PostgreSQL et se base sur les catalogues et les héritages de PostgreSQL. Par contre, les auteurs affirment supporter les versions 6.5.x, 7.0.x et 7.1.x de PostgreSQL, mais la date de dernière release du projet ne laisse pas présager du support pour les versions de postgreSQL postérieures à 7.2.x.

Différents modèles de réplication sont disponibles, tels que la réplication maître-esclave, l'*update anywhere*, la répartition de charges... Chaque réplication est paramétrable. Le logiciel est pourvu d'algorithmes de résolution des conflits au niveau des tables. Il est également possible d'ajouter des algorithmes personnalisés.

Réplication

La réplication utilise des *sockets* client-serveur. Elle peut être programmée ou basée sur des *triggers*.

Les opérations qui ont été annulées par les algorithmes de résolution des conflits sont "loguées" et peuvent être resoumises. Les utilisateurs dont les opérations ont été annulées peuvent être prévenus par mél et recevoir les détails des transactions dans une forme lisible. Il est également possible de "monitorer" le processus de réplication.

Contacts

PostgreSQL Replicator est un logiciel libre publié sous licence GNU. Les membres du projet peuvent être contactés à l'adresse **pgreplicator@bp.lnf.it**. Plus d'informations sont disponibles sur **http://pgreplicator.sourceforge.net**.

DBMirror Postgres Database Mirroring

DBMirror est un système de réplication Maître-Esclave(s), développé par Steven Singer de la société Navtech Systems Support Inc. (Il peut être joint à l'adresse suivante : **ssinger@navtechinc.com**).

Fonctionnalités

DBMirror propose les fonctionnalités suivantes :

- Support d'esclaves multiples ;
- Maintien des transactions ;
- Choix des tables à répliquer.

Fonctionnalités

Réplication

DBMirror utilise un *daemon* en Perl qui sert de serveur de réplication et une fonction en C appelée par les triggers déclenchés lors de toute insertion, suppression ou *update* sur une des tables concernées par la réplication et gérant l'écriture des requêtes à "mirorrer". Les triggers sont placés sur chaque table prise

Le principe consiste à utiliser deux démons, un de lecture, et un d'écriture

en compte dans le processus de *mirroring*. Ces triggers vont permettre de loguer les informations d'édition des tables dans deux tables propres à DBMirror.

Un script Perl tourne en permanence pour chaque base esclave. Ce script examine le contenu des tables, à la recherche des transactions à déployer.

Limitations

DBMirror est relativement simple à mettre en œuvre. La plus grosse difficulté consiste à écrire le script SQL de mise en place des triggers pour chaque table que l'on souhaite mirrorer. Il n'est pas possible par exemple de simplement préciser la base à répliquer.

D'autre part, il ne semble pas prévu de lock particulier pour interdire les modifications sur la base esclave.

La gestion des accès devra se faire de façon logicielle au niveau des interfaces d'accès aux bases de données. Il devrait être possible ainsi de définir plusieurs esclaves pour un maître et d'utiliser une solution de répartition de charge sur les accès en lecture. Seul le maître devra être utilisé en écriture.

Usogres

Usogres (*Synchronizing Option for PostgreSQL*) est un utilitaire développé en C++ permettant la création d'un backup d'une base de données en temps réel. La version actuelle est la version 0.8.1. d'avril 2002. Son développeur l'estime stable à ce niveau.

Principe

Contrairement à la plupart des systèmes dits de réplication développés pour PostgreSQL, Usogres n'a pas la prétention de faire de la réplication dans le sens défini plus haut. Il permet simplement de copier une base de données maître sur une base esclave et de maintenir l'égalité des deux bases.

Fonctionnement

Peu d'indications sont fournies sur le site concernant le fonctionnement d'Usogres. Quoiqu'il en soit, il s'agit de répercuter les requêtes effectuées sur le maître vers l'esclave. La simplicité du projet et son apparente facilité de mise en application incitent à tester ce projet, qui est utilisé dans la société où travaille son concepteur.

Contacts

Pour plus d'informations, vous pouvez vous rendre sur le site **http://usogres.good-day.net**. Son auteur, Tetsuichi Hosokawa , est joignable à l'adresse **hosokawa@dear-jpn.com**.

eRserver : Enterprise Replication Server

eRServer est un système de réplication Maître unique vers Esclave(s) basé sur des triggers. Le projet est financé par PostgreSQL, Inc. ERServer, disponible en version 1.2, est un système de réplication commercial. Les sources d'eRserver ont été rendues disponibles à la communauté OpenSource en août 2003.

Fonctionnalités

Du fait du caractère essentiellement commercial de ce projet, peu d'informations sont disponibles sur le site. Parmi les fonctionnalités annoncées, on trouve :

- Distribution des requêtes et équilibrage de charge ;
- Réplication Maitre —> Esclave(s) ;
- Fusion de données (PostgreSQL vers PostgreSQL) ;
- Réplication périodique ou continue ;
- Mirroring et réparation de pannes ;
- Basculement à chaud/ Tolérance aux pannes (système, logiciel et réseau).

Fonctionnalités

Contact

Le site Internet du projet est : **http://www.erserver.com/eRServer.html**. Des informations peuvent également être obtenues par mél : **info@pgsql.com**.

Résumé

Le tableau suivant présente un résumé des différents projets de réplication dont nous avons parlé. Pour chacun, il est dit s'il est besoin de recompiler le serveur de base de données (code source intégré) ou si le projet est indépendant.

Projet	Type
Postgres-R	Intégré, Synchrone Symétrique
DBBalancer	Externe, Duplication de requêtes
pgReplicator	Externe, Asynchrone Symétrique/ Asymétrique
eRserver	Intégré, Asymétrique, Synchrone/ Asynchrone
DBMirror	Externe, Asynchrone Asymétrique
Usogres	Externe, Duplication de bases

Projets

Les solutions d'équilibrage de charge

SQLB : SQL Load Balancer

Au-delà d'un système de réplication, le projet SQLB se veut un améliorateur de requêtes SQL à une base de données. MySQL, Postgre SQL et Oracle sont supportés. Le projet est publié sous licence GPL.

Principe

Ce programme consiste en un ensemble de daemons effectuant des connexions permanentes multiples à un ou plusieurs SGBD au démarrage. Un programme externe lié à l'aide de la bibliothèque "client SQLB" peut alors envoyer des requêtes vers ces daemons et en récupérer le résultat sans avoir à effectuer de connexions/déconnexions.

Les requêtes sont améliorées grâce aux connexions permanentes. En fait, vous gagnez le temps des connexions/déconnexions à chaque requête. Le gain est soumis au type de SGBD utilisé et aux types de requêtes soumises. SQLB est le plus efficace dans deux cas bien particuliers :

- Lorsque le SGBD utilise un protocole de connexion complexe, tel Oracle ;
- Lorsque la base de données est sur un serveur distant.

Avec des bases de données telles que MySQL et PostgreSQL, hébergées sur le même serveur, le gain est négligeable, ces serveurs utilisant les sockets UNIX, tout comme SQLB.

Mais si les SGBD se trouvent sur des serveurs distants, vous pouvez arriver aux mêmes résultats qu'avec des SGBD locaux. SQLB peut être une bonne solution lorsque les serveurs doivent accéder à une base de données centralisée.

Fonctionnalités

Petite liste des fonctionnalités déclarées de SQLB :

- Support de MySQL, Postgre SQL et Oracle ;
- Définition d'un pool de connexions multiples à un ou plusieurs SGBD de type quelconque, un identifiant permet de spécifier l'accès à utiliser ;
- Possibilité de gestion de la charge : SQLB est capable de créer ou stopper de nouvelles connexions de façon à gérer un accroissement du nombre de requêtes ;
- Possibilité d'ajout d'un test d'intégrité : SQLB va vérifier régulièrement que les requêtes en attente ne sont pas corrompues puisqu'elles sont stockées dans un segment de mémoire partagée ;
- Vérification permanente des connexions et au besoin reconnexion ;
- Logs : Un des daemons est dédié au log de l'activité (erreurs, informations utiles, requêtes effectuées, messages de débogage) ;
- API très simple : l'API consiste en une seule fonction, les résultats sont stockés dans une structure ;
- La migration vers un autre SGBD peut se faire sans avoir à recoder tout le code ;
- Gestion du *timeout* : le client gère le timeout de sorte que le programme ne se bloque pas en cas de non-réponse de la base de données.

Intérêt

SQLB est utile dans deux cas particuliers. Le premier concerne les serveurs gérant un très grand nombre de requêtes et nécessitant de ce fait un temps d'accès aux données le plus court possible.

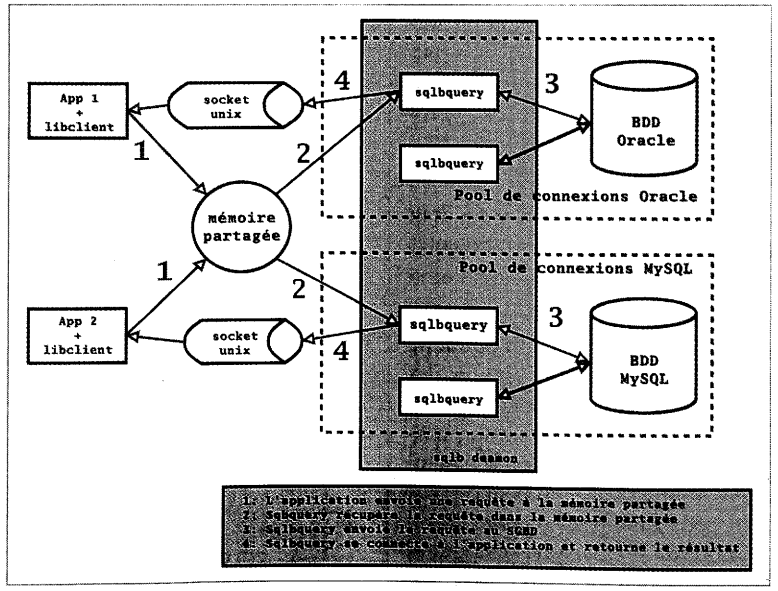
Ceci est le cas notamment des sites Web qui requièrent un grand nombre de requêtes SELECT et très peu de requêtes INSERT/ UPDATE.

Concernant les données sensibles, il est préférable d'éviter d'utiliser SQLB, celui-ci ne gérant pas toutes les requêtes se terminant mal. Il informera le programme externe du timeout, mais ne donnera pas d'informations sur l'exécution ou non de la requête.

L'autre intérêt est pour les développeurs qui souhaitent un développement rapide et n'ont pas particulièrement envie de redévelopper une application lorsqu'il s'agit de changer de SGBD.

Fonctionnement

La figure suivante présente un diagramme de fonctionnement lorsque deux applications externes sont liées à l'aide de la bibliothèque `sqlbclient`, et envoient une requête au daemon SQLB configuré pour avoir deux pools de connexion, l'un avec un SGBD MySQL, l'autre avec un SGBD Oracle. SQLB est également configuré pour avoir 2 connexions permanentes par pool.



Lorsque l'application doit exécuter une requête SQL, la requête est placée dans la mémoire partagée, `sqlbquery` est prévenu et l'application attend la libération d'une connexion. Un des daemons se libère, il récupère le *job*.

Le job est envoyé à travers la connexion permanente à la base. Le résultat est retourné, une connexion à la socket UNIX de l'application qui a soumis le job est ouverte et le résultat y est envoyé.

Contacts

Frédéric Ruaudel, responsable du projet peut être contacté à l'adresse suivante : **grumz@users.sourceforge.net**.

Le site : **http://sqlb.sourceforge.net**.

SQL Relay

SQL Relay n'est pas non plus un système de réplication de bases de données, mais un système d'équilibrage de charge permettant de gérer les connexions vers des bases répliquées. Plus avancé que SQLB, il en est aussi plus complexe, il supporte un plus grand nombre de SGBD, plus de langages et une API plus complexe. Enfin, la documentation est plus fournie.

Principe

SQL Relay est un système de *pooling* de connexions persistantes à une base de données, de *proxying* et d'équilibrage de charge pour Linux et UNIX.

Il supporte un grand nombre de SGBD (ODBC, ORACLE, MySQL, PostgreSQL, etc.), avec des API pour le C, C++, Perl... et comprend une interface graphique de configuration.

Les API supportent des fonctions avancées d'opérations sur les bases de données. SQL Relay est très utile pour accélérer les applications Web se basant sur des bases de données, les connexions aux bases de données à partir d'OS non supportés, la migration entre différentes bases de données, l'accès distribué à des bases de données répliquées et le contrôle des accès.

Fonctionnement

Les daemons de connexion de SQL Relay se connectent et maintiennent des sessions avec les bases de données. Un de ces daemons (appelé `listenert`) est chargé de surveiller les ports UNIX/INET dans l'attente de connexions de clients.

Lorsqu'un client se connecte au serveur, si un daemon de connexion est disponible, le listener relie le client et ce daemon.

Si aucun daemon n'est disponible, le client est placé dans la file d'attente. Lorsque le client est connecté, il communique avec la base à travers la session maintenue par le daemon.

SQL Relay facilite la distribution de la charge entre des bases répliquées. Une solution courante pour résoudre les problèmes d'accès à un serveur Web consiste à déployer plusieurs serveurs, chacun attaquant une base de données, ces bases étant mises à jour à intervalles réguliers.

Pour pouvoir faire cela, il faudrait que les machines soit équivalentes en termes de puissance.

L'utilisation de SQL Relay permet de s'affranchir de cette contrainte puisqu'il peut se connecter à des serveurs de bases de données multiples, répliqués ou en cluster, permettant de se connecter à la machine la moins chargée. Il peut de plus être configuré pour maintenir un nombre de connexions plus grand sur les machines les plus puissantes.

Contacts

David Muse, administrateur du projet, peut être joint à l'adresse **dmuse@firstworks.com**. Le site, assez complet, est disponible sur **http://sqlrelay.sourceforge.net**.

Pour aller plus loin

Par cet article, nous avons présenté succinctement les différentes possibilités offertes en matière de réplication de bases de données PostgreSQL. PostgreSQL, en effet, contrairement à ses concurrents ne dispose pas encore d'une solution interne de réplication.

Certaines des solutions décrites dans cet article se présentent tout de même comme des modules compilables avec le code du serveur. Cela impliquera certes une recompilation du code source, ce qui n'est pas forcément la première chose que l'on souhaite entreprendre lorsque l'on veut tester une solution de réplication, mais présentera l'avantage de s'intercaler plus finement au sein d'une transaction.

D'autre part, la possibilité de tester des solutions externes au serveur permettent par l'analyse du source ou des mécanismes mis en œuvre d'approcher la philosophie de la réplication des bases de données. Et l'on s'aperçoit rapidement que la réplication n'est pas une mince affaire ;-)

Il ne reste plus qu'à tester toutes ces solutions !

Stéphane Schildknecht
dbadialog@free.frs

Les définitions suivantes vous permettront peut-être de pénétrer un peu le jargon des DBA !

ACID	Propriétés d'une transaction (Atomicité - Consistance - Isolation - Durabilité)
Abort	Transaction non "commitée"
Asynchrone	Envoi post-commit d'informations aux autres systèmes impliqués dans la distribution
Atomicité	Succès du commit d'une transaction
Consistance	Une transaction ne modifie pas la consistance de la base
Commit	Transaction correctement écrite dans la base
Durabilité	Le résultat d'une transaction commitée est stockée de manière pérenne
Isolation	Toute transaction est exécutée comme si elle était unique
Optimiste	Réplication synchrone où seules les écritures sont propagées, dans le même ordre ; les lectures sont locales
Peer-to-Peer	Réplication où les lectures-écritures sont locales et chaque système est responsable de la propagation de ses données, synchrone ou asynchrone
Pessimiste	Réplication synchrone dans laquelle chaque transaction (en lecture et écriture) est propagée dans le même ordre à l'ensemble du système
Post database	Réplication où les lectures-écritures sur le maître sont envoyées aux esclaves à l'aide de triggers ou de logs
Pre database	Réplication de type lecture seule où un processus intercepte les données du client et lui envoie les écritures
Read-only	Réplication où les systèmes esclaves lisent sur leur bases locales et envoient les écritures sur le maître tandis que le maître met les esclaves à jour
Synchrone	Envoi pre-commit des informations aux autres systèmes en vérifiant le commit ou *rollback* sur chaque transaction pour la distribution complète
Transaction	Groupe de commandes SQL dont le résultat sera visible par l'ensemble du système comme une entité après commit (rien ne sera vu en cas de rollback) ; exécution d'un programme qui effectue ne tâche administrative en accédant à une base partagée
Transaction Processing Application	Ensemble de transactions destinées à automatiser une activité
Tuple	Objet spécifique stocké dans une table et représentant une version d'une colonne logique. Une colonne peut exister dans différentes versions simultanément.

Glossaire

Réplication avec MySQL

MySQL est certainement la base de données la plus utilisée par le quidam souhaitant créer son propre site Web ou simplement découvrir rapidement le monde des SGBD. Elle doit très certainement sa notoriété à sa grande simplicité d'utilisation et d'administration. Et bien que les amoureux du *tuning* la trouvent souvent trop simpliste, il n'en reste pas moins qu'elle permet de mettre en place la réplication avec un minimum d'efforts là où d'autres obligent d'avoir sous la main une brochette d'experts.

Dans cet article nous allons mettre en place une réplication entre deux bases MySQL en mode unidirectionnel/asynchrone. En fait, la propagation synchrone n'est pas possible avec MySQL - tout au moins pas avec le système de réplication fourni en standard avec la base.

Installation

L'architecture minimale pour faire de la réplication est composée de deux machines. Il est tout à fait possible de faire de la réplication entre deux instances de bases placées sur la même machine, mais l'intérêt est limité.

Dans le cas présent, nous aurons la machine "uranium" servant de *master* et "helium" pour le *slave*. Ces deux machines sont installées sous Linux Debian Sid.

Nous allons faire simple pour le moment et considérer que ni la base master, ni la base slave ne sont installées sur les machines. Nous utiliserons la version 4.0.16 de MySQL.

Installation du master

Vous l'avez certainement compris, la première chose à faire consiste à installer la base. Pour cela, je vous laisse faire en fonction de vos habitudes.

Une fois la base prête, nous devons créer un utilisateur spécial qui va contrôler la réplication. Il doit pouvoir accéder à la base master depuis la machine où se trouvera le slave.

Pour savoir où se trouve le répertoire de données, regarder l'entrée `datadir` dans le fichier de configuration de la base (`.cnf`).

```
uranium:~# locate my.cnf
/etc/mysql/my.cnf
uranium:~# cat /etc/mysql/my.cnf | grep datadir
datadir         = /var/lib/mysql
uranium:~# tar zcf /tmp/mysql-data-master.tgz /var/lib/mysql/
uranium:~#
```

Avant de pouvoir relancer la base, vous devez modifier le fichier de configuration. Ouvrez ce fichier avec l'éditeur de votre choix et ajoutez les deux lignes suivantes à la fin de la section `[mysqld]` :

```
/etc/mysql/my.cnf :
...
[mysqld]
...
server-id = 1
log-bin   = /var/log/mysql-bin.log
...
```

```
uranium:~# mysql -uroot -p mysql
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 78 to server version: 4.0.16-log

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

master> grant FILE, REPLICATION SLAVE on *.* to repl@'%' IDENTIFIED BY 'repl';
Query OK, 1 rows affected (0.14 sec)

master>
```

Nous attribuons à cet utilisateur "repl" les droits `FILE` et `REPLICATION SLAVE`. Si vous utilisez une version antérieure à la 3.23.29, le droit `REPLICATION SLAVE` n'existe pas et `FILE` suffit donc. Si votre version est supérieure (ou égale) à la 3.23.29, vous pouvez ignorer le droit `FILE`.

Pour des raisons de sécurité évidentes, nous ne donnerons pas plus de droits à cet utilisateur en le cantonnant ainsi dans son rôle de contrôleur de la réplication.

La seconde étape implique d'arrêter la base.

```
master> exit
Bye


uranium:~# /etc/init.d/mysql stop
Stopping MySQL database server: mysqld.
uranium:~#
```

Il faut maintenant sauvegarder les données de la base afin de pouvoir les transférer, plus tard, sur le slave.

L'entrée `server-id` sert à identifier de manière unique une base dans un groupe de réplication. Vous pouvez lui attribuer la valeur de votre choix en fonction de votre logique.

`log-bin` permet d'activer le *log* binaire tout en précisant l'emplacement et le nom de ce fichier de log.

C'est dans ce dernier que vont être stockés les ordres de modifications faites sur le master avant d'être envoyés au slave.

C'est terminé pour le master, vous pouvez redémarrer la base.

Installation du slave

Comme pour le master, il faut bien entendu commencer par installer la base. Ma remarque sur le sujet sera la même que pour l'installation de la base master. Une seule différence cependant, il est inutile de démarrer la base (ou si votre système de *package* ne vous laisse pas le choix, arrêtez-la).

Editez le fichier de configuration de la base et ajoutez à la fin de la section `[mysqld]` les lignes suivantes :

```
/etc/mysql/my.cnf :
...
[mysqld]
...
server-id       = 2
log-bin         = /var/log/mysql-bin.log
master-host     = uranium
master-user     = repl
master-password = repl
master-port     = 3306
master-connect-retry = 10
...
```

Nous retrouvons le `server-id`. Faites bien attention de lui attribuer une valeur différente de celle du master.

Les entrées `master-host`, `master-user`, `master-password` et `master-port` sont suffisamment explicites pour que je ne m'étende pas dessus. La valeur de `master-connect-retry` indique, en secondes, le délai entre deux tentatives de connexion au master en cas d'échec.

Vous pouvez vous demander pourquoi nous activons le log binaire ici. Il s'agit d'un principe de protection qui prendra tout son intérêt lorsque nous parlerons de la tolérance de panne.

Il faut ensuite récupérer l'archive de sauvegarde des données que nous avons produite lors de l'installation du master.

Une fois ceci fait, vous pouvez supprimer le répertoire de données qui a été créé lors de l'installation de la base slave (ou en faire une copie de sauvegarde) et le remplacer par les données issues du master.

```
helium:~# cd /
helium:/# scp root@uranium:/tmp/mysql-data-master.tgz /tmp/
root@uranium's password:
mysql-data-master.tgz                         100%  3.7K   1.0MB/s   00:02
helium:/# mv /var/lib/mysql /var/lib/mysql_old
helium:/# tar zxf /tmp/mysql-data-master.tgz
helium:/#
```

C'est terminé pour le slave, vous pouvez démarrer la base.

Tester la réplication

Nous sommes maintenant prêts à vérifier si cela fonctionne correctement.

Nous profiterons de ces tests pour voir quelques commandes utiles pour l'administration d'un tel système et pour expliquer plus précisément comment fonctionne la réplication avec MySQL.

Etant partis d'un "système vide", nous n'avons pour le moment que deux bases : `mysql` et `test`. Nous allons créer une base de test "lmtest" dans laquelle nous allons créer une unique table "tbltest". Bien entendu, nous faisons tout cela depuis le master :

```
master> create database lmtest;
Query OK, 1 row affected (0.00 sec)

master> connect lmtest;
Connection id:    9
Current database: lmtest

master> CREATE TABLE tbltest (
    ->   id       INT(11) AUTO_INCREMENT PRIMARY KEY,
    ->   username VARCHAR(128) NOT NULL,
    ->   age      INT(11),
    ->   ts       TIMESTAMP DEFAULT 'CURRENT_TIMESTAMP'
    -> );
Query OK, 0 rows affected (0.00 sec)

master> show tables;
+-----------------+
| Tables_in_lmtest |
+-----------------+
| tbltest         |
+-----------------+
1 row in set (0.00 sec)

master>
```

> Remarque : Pour éviter la confusion, j'utilise `master` comme *prompt* sur le master et `slave` pour le slave.

Bon, regardons maintenant ce que nous avons sur le slave. Si vous ne vous êtes pas trompé lors de l'installation, voici ce que vous devez obtenir :

```
slave> show databases;
+----------+
| Database |
+----------+
| lmtest   |
| mysql    |
| test     |
+----------+
5 rows in set (0.00 sec)

slave> connect lmtest;
Connection id:    10
Current database: lmtest

slave> show tables;
+-----------------+
| Tables_in_lmtest |
+-----------------+
| tbltest         |
+-----------------+
1 row in set (0.00 sec)

slave> desc tbltest;
+----------+-----------------+------+-----+---------+----------------+
| Field    | Type            | Null | Key | Default | Extra          |
+----------+-----------------+------+-----+---------+----------------+
| id       | int(11)         |      | PRI | NULL    | auto_increment |
| username | varchar(128)    |      |     |         |                |
| age      | int(11)         | YES  |     | NULL    |                |
| ts       | timestamp(14)   | YES  |     | NULL    |                |
+----------+-----------------+------+-----+---------+----------------+
4 rows in set (0.00 sec)

slave>
```

Première constatation, le système de réplication de MySQL se démarque des autres par le fait que tout est répliqué.

En effet, avec des bases telles que Oracle ou DB2, vous avez la possibilité de répliquer seulement une base, voire même d'être plus fin en ne sélectionnant que certaines tables.

Avec MySQL c'est tout ou rien. Dans la plupart des cas, on serait tenté de dire que cela n'est pas grave. Pourtant, cela n'est pas neutre.

Bon, poussons un peu plus loin et insérons des données :

```
master> INSERT INTO tbltest( username, age ) VALUES (
'greg', 30 );
Query OK, 1 row affected (0.11 sec)

master> select * from tbltest;
+----+----------+------+----------------+
| id | username | age  | ts             |
+----+----------+------+----------------+
|  1 | greg     |   30 | 20040110152747 |
+----+----------+------+----------------+
1 row in set (0.00 sec)

master>
```

Vérification sur le slave :

```
slave> select * from tbltest;
+----+----------+------+----------------+
| id | username | age  | ts             |
+----+----------+------+----------------+
|  1 | greg     |   30 | 20040110152747 |
+----+----------+------+----------------+
1 row in set (0.01 sec)

slave>
```

Tout fonctionne sans problèmes.

Ecrire sur le slave

Pour aller plus loin et comprendre comment fonctionne la réplication, vérifions ce qui se passe si nous insérons une ligne sur le slave puis quand nous faisons une modification sur le master.

```
slave> INSERT INTO tbltest( username, age ) VALUES ( 'arthur', 5 );
Query OK, 1 row affected (0.00 sec)

slave> select * from tbltest;
+----+----------+------+----------------+
| id | username | age  | ts             |
+----+----------+------+----------------+
|  1 | greg     |   30 | 20040110152747 |
|  2 | arthur   |    5 | 20040110180618 |
+----+----------+------+----------------+
2 rows in set (0.00 sec)

slave>
```

Jusque-là tout semble normal ;) Par contre, si vous faites un `select` sur le master, vous pouvez attendre aussi longtemps que vous le voulez, jamais vous ne verrez s'afficher les nouvelles données.

Pire, ajoutons une ligne sur le master :

```
master> INSERT INTO tbltest( username, age ) VALUES ( 'colyne', 2 );
Query OK, 1 row affected (0.00 sec)

master> select * from tbltest;
+----+----------+------+----------------+
| id | username | age  | ts             |
+----+----------+------+----------------+
|  1 | greg     |   30 | 20040110152747 |
|  2 | colyne   |    2 | 20040110153627 |
+----+----------+------+----------------+
2 rows in set (0.00 sec)

master>
```

Voici ce que nous avons ensuite sur le slave :

```
slave> select * from tbltest;
+----+----------+------+----------------+
| id | username | age  | ts             |
+----+----------+------+----------------+
|  1 | greg     |   30 | 20040110152747 |
|  2 | arthur   |    5 | 20040110180618 |
+----+----------+------+----------------+
2 rows in set (0.00 sec)

slave>
```

Harg ! La ligne n'a pas été répliquée !

Et vous pouvez ajouter autant de lignes que vous le souhaitez sur le master, jamais vous ne les verrez apparaître sur le slave.

Bien entendu, ce problème n'est heureusement pas insoluble.

Mais encore faut-il savoir où est le problème. Pour cela, sur le slave, tapez la commande suivante :

```
slave> SHOW SLAVE STATUS;
... | Last_errno | Last_error                                  | ...
... | 1062       | Error 'Duplicate entry '2' for key 1' on query | ...
```

J'ai légèrement tronqué la sortie pour ne vous montrer que la partie qui nous intéresse. Comme on pouvait s'y attendre, le fait d'avoir inséré des données sur le slave bloque la réplication.

C'est somme toute logique dans une politique de conservation des données. En effet, le système n'est pas à même de choisir si c'est le slave qui a raison ou le master.

Il attend donc une intervention humaine (bonne nouvelle pour les parano de SF ;))

Vu la faible quantité de données impactées dans le cas présent, le plus simple consiste à identifier qui a raison (slave ou master) et modifier les données en conséquence.

Donc, si nous considérons dans le cas présent que la ligne sur le slave est mauvaise, il suffit tout simplement de la supprimer.

```
slave> delete from tbltest where id='2';
```

Cependant, cela ne suffit pas. Si vous essayez, vous constaterez que la réplication ne se fera pas plus malgré la suppression de la ligne responsable du conflit.

Il faut en effet relancer la réplication. Pour cela, rien de plus simple, il suffit d'arrêter puis de redémarrer l'esclave. Bien entendu, si vous avez une base en production, cela peut poser des problèmes.

En général, on ne coupe pas une base de données. Eh bien là, aussi c'est la même chose.

Nous n'allons pas arrêter la base, juste stopper le mode esclave et le relancer. Pour cela, tapez les commandes suivantes :

```
slave> slave stop;
Query OK, 0 rows affected (0.00 sec)

slave> slave start;
Query OK, 0 rows affected (0.00 sec)

slave> select * from tbltest;
+----+----------+------+----------------+
| id | username | age  | ts             |
+----+----------+------+----------------+
|  1 | greg     |   30 | 20040110152747 |
|  2 | colyne   |    2 | 20040110153627 |
+----+----------+------+----------------+
2 rows in set (0.00 sec)

slave>
```

Simple, non ? En fait, dans le cas présent, c'est effectivement très simple, mais si vous vous lancez vraiment dans la réplication, attendez-vous à devoir traiter des cas beaucoup plus épineux.

Tolérance de panne

Maintenant que vous savez que tout cela fonctionne, vous pouvez envisager de faire de la tolérance de panne.

Imaginons pour cela que vous ayez un site Web avec lequel vous utilisez une base de données : rien de plus classique en fait. Maintenant, imaginez que la machine hébergeant votre base de données décide de rendre l'âme.

Là généralement c'est la catastrophe. Mais non puisque vous avez mis votre base en réplication !

En effet, toutes vos données sont sur le slave, donc il suffit de demander à vos scripts d'attaquer le slave.

Bien entendu, tout ceci se prépare. En effet, il ne faut pas attendre que vous ayez remarqué que le master est dans les choux pour basculer sur le slave.

Donc, il peut être intéressant d'écrire un script vérifiant l'état du master et d'utiliser un DNS que vous pourrez alors modifier dynamiquement grâce à `nsupdate` si vous utilisez `bind` par exemple.

Le second problème interviendra lorsque vous aurez remonté votre (ancien) master. En effet, pendant un certain temps, c'est le slave qui a recueilli les données.

Bien entendu, vous pouvez faire une sauvegarde de vos données du slave, l'arrêter, tout remettre sur le master et refaire la configuration.

Malheureusement, cela vous oblige à stopper la base et donc à rendre votre service indisponible pendant un certain temps (qui sera généralement plus long que celui que vous avez planifié), ce qui ne plaira certainement pas à tout le monde :)

La "bonne" solution consisterait à "inverser" slave et master. C'est tout à fait possible car nous avons activé le log binaire sur le slave.

Il n'en faut pas plus pour le transformer en master. Par contre, il est inutile de le laisser en mode esclave et vous pouvez donc faire un `SLAVE STOP` .

Ensuite, nous devons récupérer les données pour les copier vers l'ancien master (qui devient donc le slave - eh oui, faut suivre !)

Pour cela, il vous faut créer une archive des données comme nous l'avons vu lors de la procédure d'installation.

Il est impératif que les données ne soient pas modifiées lors de cette opération.

Pour cela, il suffit d'utiliser la commande `FLUSH TABLES WITH READ LOCK` , de créer l'archive (avec `tar` par exemple), puis de faire un `UNLOCK TABLES` à la fin :

```
slave> FLUSH TABLES WITH READ LOCK;
Query OK, 0 rows affected (0.00 sec)

slave> quit
Bye
helium:~# tar zcf /tmp/mysql-data-slave.tgz /var/lib/mysql/
helium:~# mysql mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 24 to server version: 4.0.16-log

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

slave> SHOW MASTER STATUS;
+--------------+----------+--------------+------------------+
| File         | Position | Binlog_do_db | Binlog_ignore_db |
+--------------+----------+--------------+------------------+
| mysql-bin.012 | 1212     |              |                  |
+--------------+----------+--------------+------------------+
1 row in set (0.00 sec)

slave> UNLOCK TABLES;
Query OK, 0 rows affected (0.00 sec)

slave>
```

Il suffit maintenant de réinstaller la base sur (l'ancien) master comme nous l'avons vu plus haut (installation du slave). Il faut ensuite ajuster la position de lecture du log binaire, c'est pour cela que nous avons fait un `SHOW MASTER STATUS` avant de "délocker" les tables :

```
master> slave stop;
Query OK, 0 rows affected (0.00 sec)

master> CHANGE MASTER TO
    -> MASTER_HOST='uranium',
    -> MASTER_PORT=3306,
    -> MASTER_USER='repl',
    -> MASTER_PASSWORD='repl',
    -> MASTER_LOG_FILE='mysql-bin.012',
    -> MASTER_LOG_POS=1212;
Query OK, 0 rows affected (0.00 sec)

master> slave start;
Query OK, 0 rows affected (0.01 sec)

master>
```

A partir de là, tout doit fonctionner comme avant :

```
slave> INSERT INTO tbltest( username, age ) VALUES ( 'arthur', 5 );
Query OK, 1 row affected (0.00 sec)

slave> select * from tbltest;
+----+----------+------+----------------+
| id | username | age  | ts             |
+----+----------+------+----------------+
|  1 | greg     |   30 | 20040110152747 |
|  2 | colyne   |    2 | 20040110153627 |
|  3 | morgane  |   30 | 20040110153820 |
|  4 | maguy    |   60 | 20040110215139 |
|  5 | arthur   |    5 | 20040111011340 |
+----+----------+------+----------------+
5 rows in set (0.00 sec)

slave>
```

Et sur le master devenu slave :

```
master> select * from tbltest;
+----+----------+------+----------------+
| id | username | age  | ts             |
+----+----------+------+----------------+
|  1 | greg     |   30 | 20040110152747 |
|  2 | colyne   |    2 | 20040110153627 |
|  3 | morgane  |   30 | 20040110153820 |
|  4 | maguy    |   60 | 20040110215139 |
|  5 | arthur   |    5 | 20040111011340 |
+----+----------+------+----------------+
5 rows in set (0.00 sec)

master>
```

Aller plus loin

Maintenant que vous avez les connaissances nécessaires pour mettre en place la réplication, vous pouvez envisager des architectures plus complexes avec plusieurs esclaves.

Remarquez que vous pouvez parfaitement avoir une machine esclave qui sert de maître à d'autres.

Cela permet de mettre en place une architecture sous forme d'arbre.

Dans tous les cas, n'oubliez jamais que vous devez toujours écrire sur la machine racine.

Grégoire Lejeune
greg@webtime-project.net