HAProxy + PAF

ioguix

Démo interne

HAProxy + PAF

TITRE : HAProxy + PAF
SOUS-TITRE :

DATE: ioguix

SUJET

- *Cluster without a VIP*
  - gh #108
  - Google Cloud Engine
  - et probablement ailleurs
- Comment diriger les connexions vers l'instance de prod ?
- Comment configurer la réplication ?
- tentative avec HAProxy

Demande dans un ticket de PAF. Comment diriger les standby vers le primaire sans utiliser de vIP ? La personne dirigeait déjà les clients vers l'instance en utilisant HAProxy, chaque nœud PgSQL possédait un daemon HTTP pour exposer le statut de son instance locale.

ARCHI PRÉSENTÉE

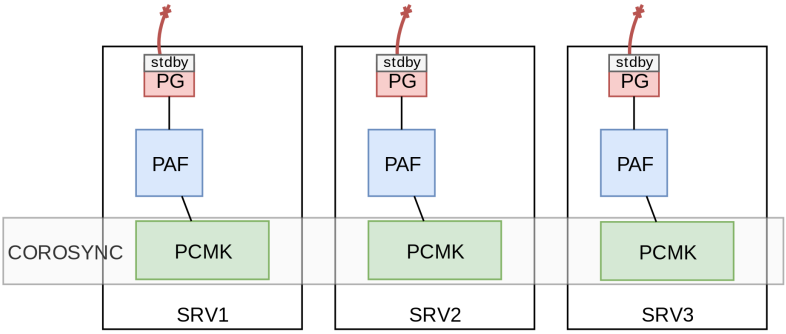
- OS : CentOS 7
- Pacemaker/PAF
- 4 nœuds: srv1, srv2, srv3 + log-sink
- HAProxy 1.5.18

La version d'HAProxy est celle distribuée sous CentOS 7. HAProxy 2.1 semble être la version stable actuelle. Voir: http://www.haproxy.org/

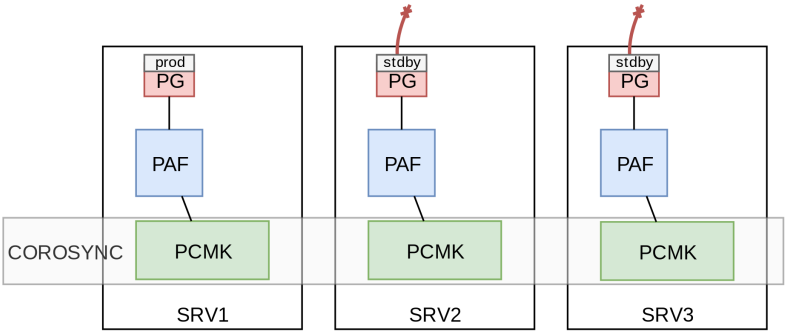
RAPPEL VIP

- détails du démarrage
- détails d'un switchover

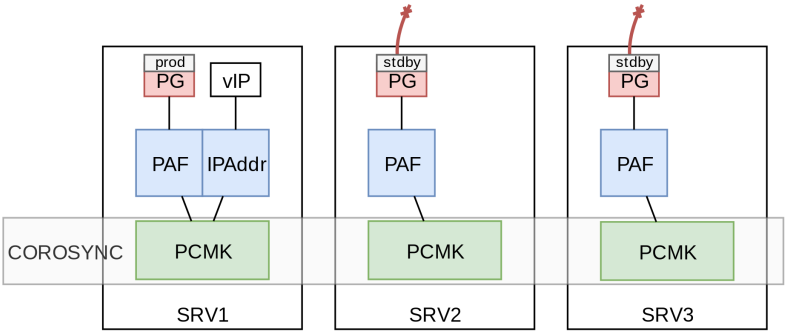
Afin de se remémorer les étapes importantes d'un cluster et ses pièges, revenons rapidement sur un cas simple: la vIP.



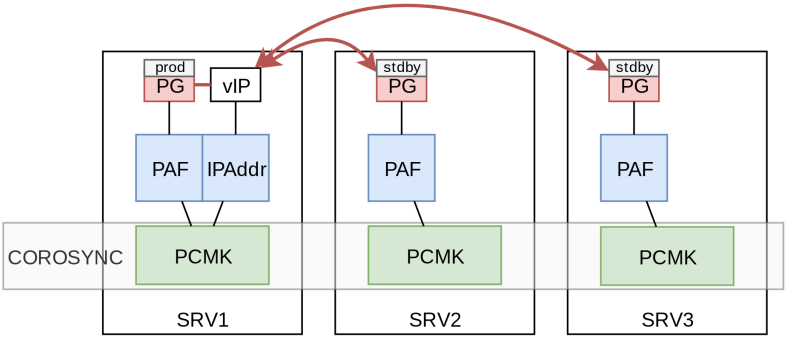
État du cluster après le démarrage. Toutes les instances sont en standby et n'ont nulle part où se connecter.



L'instance sur `srv1` est promue par le couche de HA.

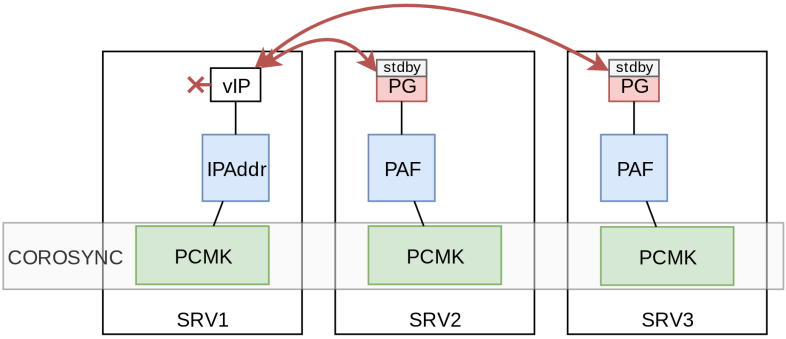


La vIP démarre sur `srv1` par adhérence au rôle master.



Les autres standby ont enfin la possibilité de se connecter au primaire et de répliquer.

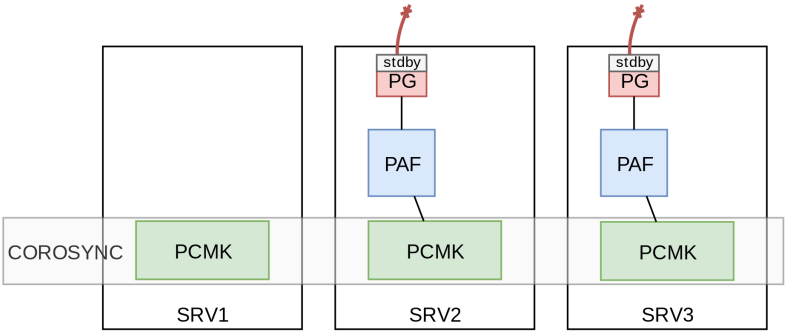
La vIP ne démarre qu'une fois le primaire promu. Ce dernier est donc prêt à accueillir les connexions entrante en streaming, ainsi que les clients souhaitant accéder à l'instance en écriture.



Le démote du primaire se fait en deux étapes: l'arrêt de l'instance, puis son démarrage en mode standby.

Durant toute cette phase, il faut faire attention aux flux de réplication avec les secondaires !

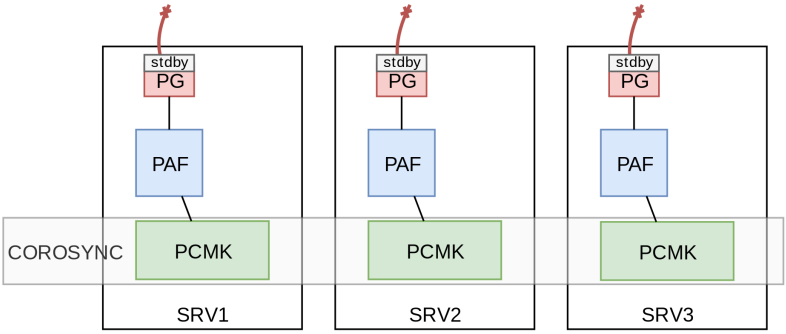
Ici, la 1ère étape: stop.



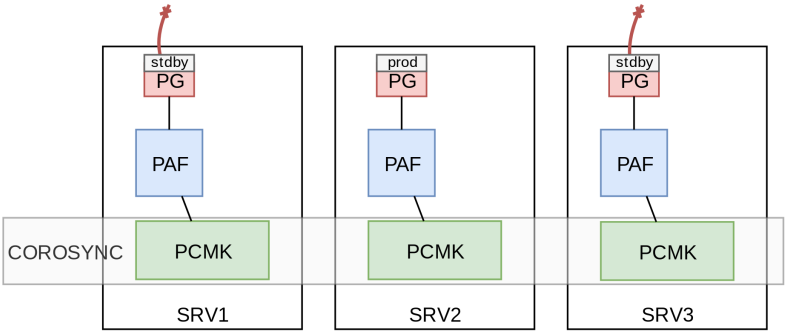
Arrêt de la vIP pour cause d'absence du rôle master dans le cluster.

L'adresse IP est supprimée du primaire qu'une fois ce dernier totalement éteint !

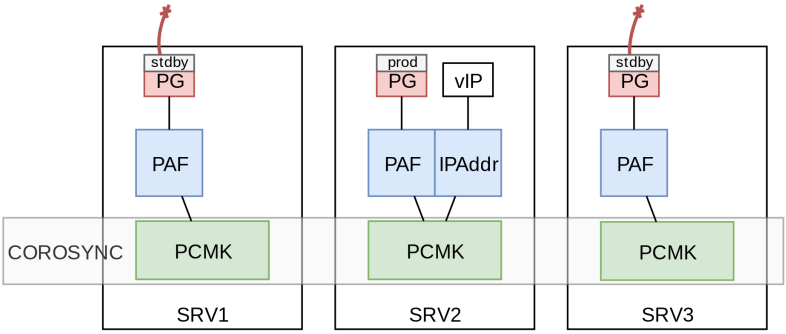
Elle sera donc resté jusqu'au bout afin que le primaire puisse envoyer tous ses WAL aux différents standby avant de s'arrêter.



Démote du primaire, 2ème étape: start. L'instance redémarre en standby et n'a nulle part où se connecter

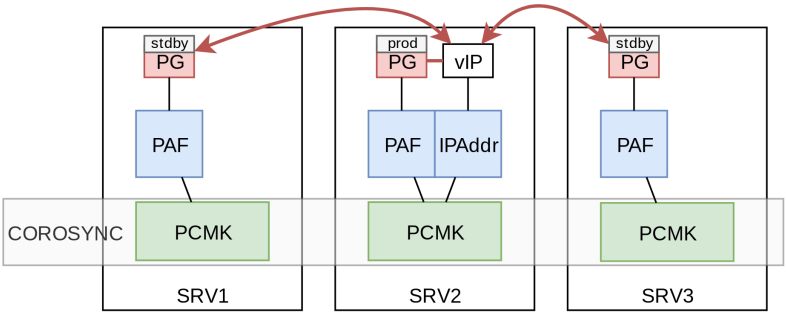


Promotion du nouveau primaire.



La vIP démarre par adhérence au primaire.

La vIP ne démarre qu'une fois le primaire promu. Ce dernier est donc prêt à accueillir les connexions entrantes en streaming.



La réplication commence.

HAPROXY

- proxy TCP couche 4…
- les clients “parlent” à HAProxy et ne voient pas les instances pgsql
- les instances “parlent” à HAProxy et ne voient pas directement les clients
- méthodes de vérifications des serveurs variées
- différents algos d’équilibrage de charge
- capacités d’analyse jusqu’en couche 7 eg. pour le HTTP

HAProxy est un proxy TCP. Il ne fait pas de routage. Un peu le même principe que le NAT. Les paquets réseaux sont donc ré-écrits.

Du point de vue de PostgreSQL, tous les clients proviennent des même adresses IP: celles des HAProxy.

HAProxy est capable de distribuer les paquets TCP reçus sur un port vers un ou plusieurs backend, avec ou sans équilibrage de charge. Une notion de health check régulier lui permet de déterminer si chaque backend de sa liste est disponible ou non.

Lors de la ré-écriture des paquets réseau, il est capable d’analyser et/ou modifier des données jusque dans la couche applicative (couche 7).

INTÉGRATION DANS LA STACK

- HAProxy écoute sur 5432 pour l'instance primaire
- HAProxy écoute sur 5433 pour les standby
- PostgreSQL écoute sur 5434
- démarré sur tous les serveurs
  - redondance (no SPoF)
  - pas en HA => simplicité

Dans notre configuration, HAProxy gère deux proxy, chacun avec plusieurs backends. L'un pointant vers le primaire en backend, l'autre pointant vers les standby en roundrobin.

HAProxy aurait pu être intégré dans la conf Pacemaker, mais l'intérêt est minime. Sans compter que ça allongerait encore la conf de Pacemaker avec des dépendances supplémentaires pour démarrer les services dans le bon ordre. HAProxy étant stateless, il suffit de le démarrer partout pour en assurer la redondance.

Qu'il soit démarré avant que PostgreSQL soit disponible ne change rien: toute tentative de connexion des clients reçoit la même erreur, en passant par HAProxy ou non: aucun service disponible sur ce port.

CONFIGURATION SIMPLIFIÉE DE HAPROXY

```
listen prod
    bind            *:5432
    server          srv1 srv1:5434
    server          srv2 srv2:5434
    server          srv3 srv3:5434

listen stdby
    bind            *:5433
    server          srv1 srv1:5434
    server          srv2 srv2:5434
    server          srv3 srv3:5434
```

Voici la configuration minimale décrite, avec les deux proxy “prd” et “stb”. Nous allons par la suite y ajouter les health check nécessaires en fonction de l’état de l’instance (down, standby, primaire).

Chacun des deux proxy définis ici écoute sur toutes les interfaces disponibles.

Ils connaissent chacun trois backends (les lignes débutant par `serveur`) vers lesquels ils pourront distribuer les connexions TCP. Chaque backend est désigné par un nom unique au sein de son proxy, précisé ici dans la seconde colonne. La troisième colonne défini l’adresse et le port de connexion vers le backend.

Nous ne faisons pas apparaitre ici les paramètres globaux ou de log de HAProxy.

STATUT DES INSTANCES

- par défaut, *roundrobin* vers tous les nœuds !
- supporte des *health check* pour filtrer les instances
- l'API ReST de Patroni expose le rôle de chaque instance
- rien dans Pacemaker/PAF
- "développement" nécessaire: pgsql-state

Les *health check* de HAProxy permettent de déterminer si un backend est disponible ou pas et si des connexions peuvent lui être confiées.

Nous pouvons utiliser cette mécanique pour n'aiguiller les connexions vers un serveur que s'il possède une instance primaire. De même pour les standby.

Pour cela, nous avons besoin d'interroger chaque serveur. HAProxy a bien un module pgsql-check, mais il est similaire à pg_isready: il n'est pas capable de faire du SQL, ni d'authentification. Il nous faut donc un service supplémentaire dans l'architecture.

SYSTEMD / PGSQL-STATE

```
cat<<'EOF' > /etc/systemd/system/pgsql-state.socket
[Unit]
Description=Local PostgreSQL state

[Socket]
ListenStream=5431
Accept=yes

[Install]
WantedBy=sockets.target
EOF
```

La personne sur github avait créé un service HTTP. Je propose plus simple, plus rapide, mieux intégré et peut-être plus performant: Systemd.

Ou comment développer et lancer un daemon en trois commandes.

SYSTEMD / PGSQL-STATE

```
cat<<'EOF' > /etc/systemd/system/pgsql-state@.service
[Unit]
Description=Local PostgreSQL state

[Service]
User=postgres
ExecStart=/usr/pgsql-12/bin/psql -p 5434 -Atc          \
  "SELECT CASE WHEN pg_is_in_recovery() THEN 'standby' \
              ELSE 'production' END"
StandardOutput=socket
EOF

systemctl --now enable pgsql-state.socket
```

C'est un descendant des vieux inetd et xinetd.

Avec la dernière commande, le "daemon" est démarré et activé au démarrage du serveur, reste à l'interroger.

SYSTEMD / PGSQL-STATE

Test simpliste depuis l'un des serveurs:

```
srv2:~# for s in srv{1..3}; do
    echo -ne "$s: "
    nc --recv-only "$s" 5431
done
srv1: production
srv2: standby
srv3: standby
```

Il suffit de se connecter au port pour recevoir le statut de l'instance sans sommation. Le socket se referme immédiatement ensuite.

La commande définie dans le paramètre `ExecStart` est exécutée à la demande et sa sortie standard est envoyé au client distant tel quel.

SYSTEMD / PGSQL-STATE

Tests de défaillance autour de `pgsql-state`:

```
# socket arrêté
srv2:~# nc --recv-only srv3 5431
Ncat: Connection refused.

# PostgreSQL arrêté
srv2:~# nc --recv-only srv3 5431
psql: error: could not connect to server: could not connect to server: No such file or
  Is the server running locally and accepting
  connections on Unix domain socket "/var/run/postgresql/.s.PGSQL.5434"?
```

En cas d'anomalie quelconque, le socket renverra toujours une erreur.

Nous avons donc plusieurs réponse possible:

- `standby`
- `production`
- tout autre chose, un message d'erreur, rien, etc.

INTÉGRATION DANS HAPROXY

```
listen prod
    bind            *:5432
    option          tcp-check
    tcp-check       connect port 5431
    tcp-check       expect string production
    default-server inter 2s fastinter 1s rise 2 fall 1 on-marked-down shutdown-session
    server          srv1 srv1:5434 check
    server          srv2 srv2:5434 check
    server          srv3 srv3:5434 check
```

On ajoute l'option `tcp-check` et ses différents sous arguments. Cette dernière est très fonctionnelle et permet de créer des health check TCP variés de toute pièce.

Voici les paramètres utilisés:

- `tcp-check connect port 5431`: connecte toi sur le port `5431` du backend à vérifier
- `tcp-check expect string production`: la réponse de celui ci doit contenir la chaîne `production`.

PARAMÈTRES

- `inter 2s`: vérifie toutes les 2s
- `fastinter 1s`: toutes les secondes en période de transition
- `rise 2`: deux check valides pour devenir accessible
- `fall 1`: inaccessible dès le premier check échoué
- `on-marked-down shutdown-sessions`: déconnecte tous les clients si le backend devient inaccessible

Les périodes de transition débutent lors d'un changement d'état et jusqu'à ce que le nombre de vérification atteigne la valeur de `rise` dans un sens, ou celle de `fall` dans l'autre.

POUR LES STANDBY

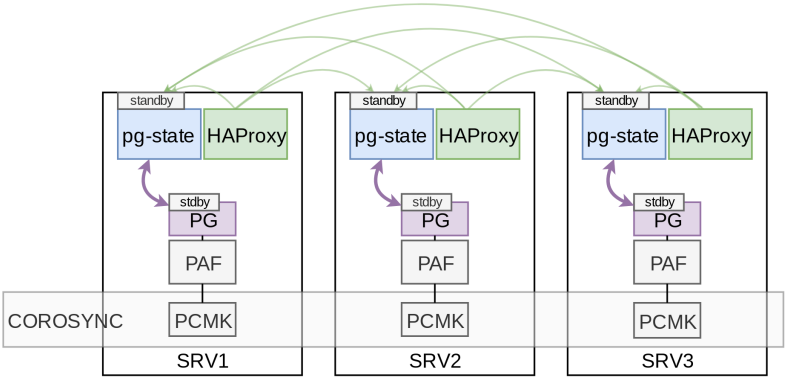
```
listen stdby
    bind            *:5433
    balance         leastconn
    option          tcp-check
    tcp-check       connect port 5431
    tcp-check       expect string standby
    default-server inter 2s fastinter 1s rise 2 fall 1 on-marked-down shutdown-session
    server          srv1 srv1:5434 check
    server          srv2 srv2:5434 check
    server          srv3 srv3:5434 check
```

Même configuration pour ce proxy. Seul le nom du proxy et les ports utilisés sont modifiés.

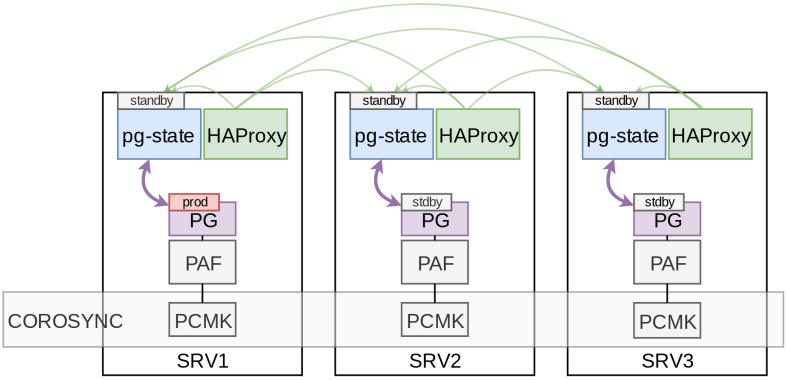
HAPROXY ET PGSQL-STATE

Récapitulons en image pour eg. le pool `prod`.

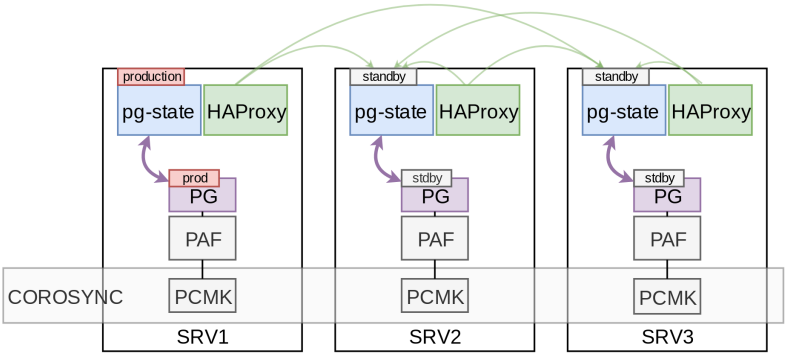
Ne concerne que les interactions entre HAProxy et pgsql-state



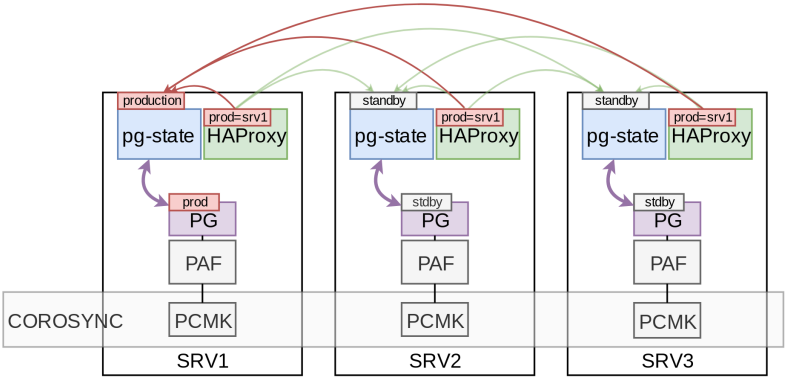
Démarrage initial



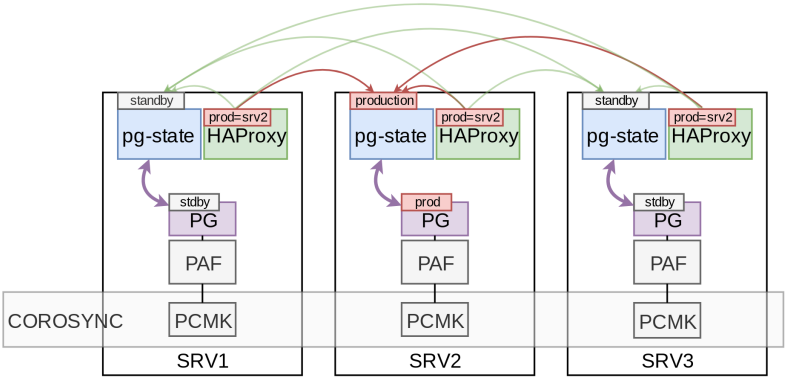
Une promotion



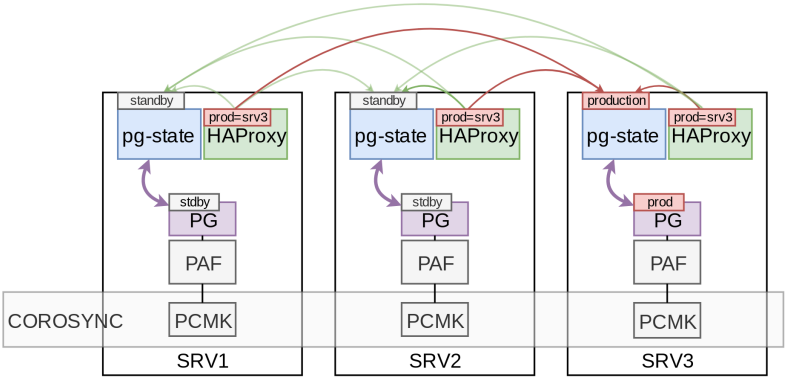
L'appelle à `pgsql-state` change de réponse



Les HAProxy activent `srv1` dans le pool `prod`…



...ou `srv2` après une bascule



…ou `srv3` après une autre bascule

HAPROXY ET PGSQL-STATE

Jamais plus d'une instance est désignée comme primaire…vraiment ?

- Pacemaker (ou Patroni) n'autorise qu'un primaire à la fois…
  - aucun problème du point de vue des données
- …mais HAProxy peut pointer une connexion vers un standby si mal configuré
- `default-server [...] rise 2 fall 1 [...]`

Une mauvaise configuration des `fall` et `rise` pourrait amener un primaire devenu secondaire de ne pas être immédiatement sorti des backends disponibles.

le `fall 1` permet d'exclure une instance dès la première erreur. Lors d'un switchover, ce moment n'intervient que lors de l'arrêt de l'instance, une fois que l'instance a terminé d'envoyer ses WAL aux standby. Comme avec une vIP.

Le `rise 2` (ou plus) nous assure qu'il est bien debout au moins depuis 2 secondes. Lorsque HAProxy désigne le nouveau master, ce dernier est donc prêt a accueillir les connexions en streaming de ses standby. Comme avec une vIP.

HAPROXY ET PGSQL-STATE

Mais pas suffisant:

- race condition possible entre deux health check
- un standby pourrait s'auto-repliquer !
- solution (comme avec une vIP):

```
local replication all              reject
host  replication all $NODENAME    reject
host  replication all 127.0.0.1/32 reject
host  replication all ::1/128      reject
```

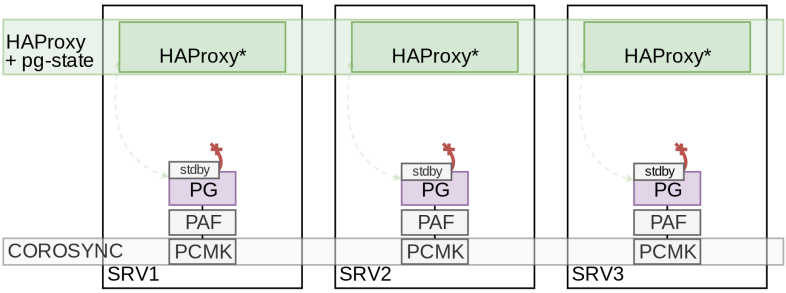
Entre deux health check, une instance primaire peut avoir reçu un demote. Son HAProxy local pointant toujours sur lui même, l'instance devenue standby se retrouve à répliquer avec elle même.

Facile à vérifier avec un helth check important.

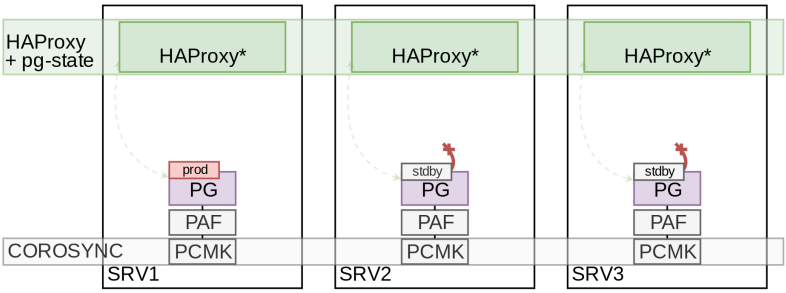
ARCHI FINALE

- HAProxy et pgsql-state fusionnés pour simplifier les diagrammes
- réplication au travers des HAProxy via localhost
- pgsql-state sur le port 5431
- clients rw sur le port 5432
- clients ro sur le port 5433
- PostgreSQL sur le port 5434

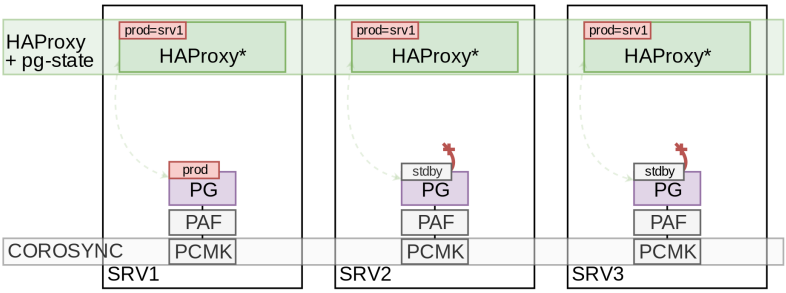
HAProxy et pgsql-state ont été fusionnés dans les diagrammes suivants et sont considéré comme une seule et même entité “intelligente” afin de simplifier la représentation.



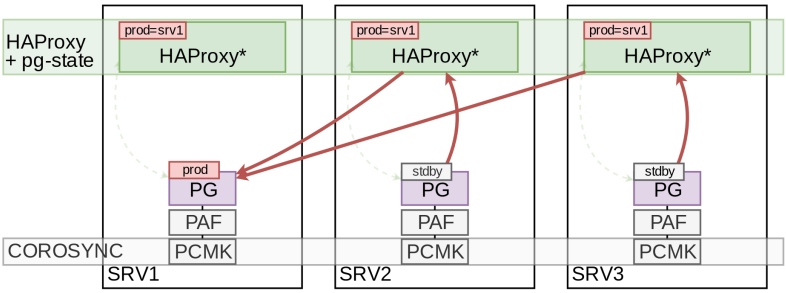
Démarrage initial, aucune réplication possible



Promotion d'une instance

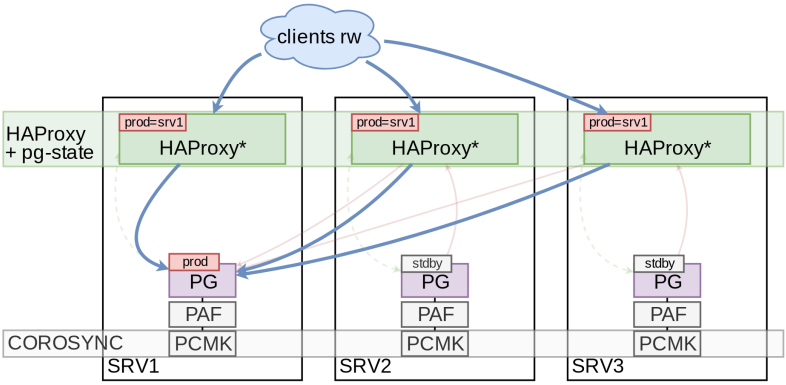


Convergence des HAProxy grâce aux health-check sur pgsql-state

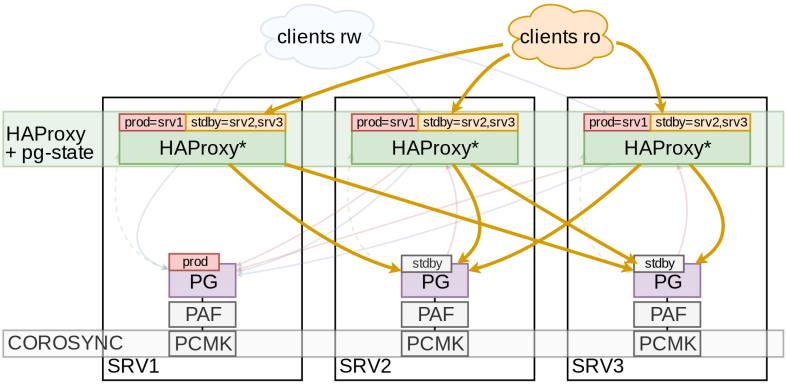


La réplication commence en passant par les HAProxy.

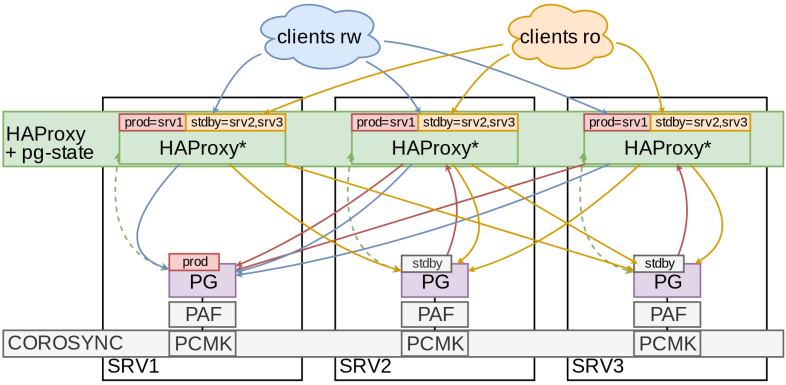
Chaque standby se connecte sur son localhost pour entrer en réplication !



Accès applicatif en RW. Quelque soit l'IP sur laquelle se connectent les clients, ils atterrissent sur l'instance primaire au travers du proxy "prod" défini dans HAProxy.



Accès applicatif en RO. Quelque soit l'IP sur laquelle se connectent les clients, ils atterrissent sur l'une des deux instances en standby au travers du proxy "stdby" défini dans HAProxy.



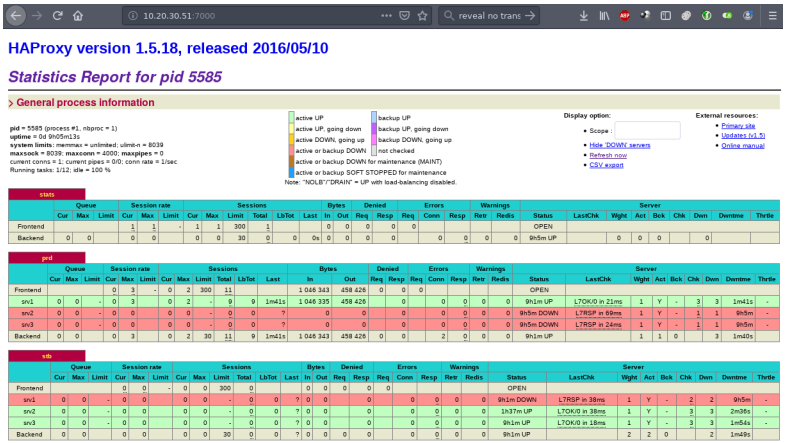
C'est bon les spaghettis.

STATS HAPROXY

```
listen stats
    mode http
    bind *:7000
    stats enable
    stats uri /
    timeout connect 15s
    timeout client  15s
    timeout server  15s
```

Possibilité d'accéder aux stats d'activité de HAProxy en HTML…

STATS HAPROXY



HAProxy version 1.5.18, released 2016/05/10

Statistics Report for pid 5585

› General process information

pid = 5585 (process #1, nbproc = 1)
uptime = 0d 9h05m13s
system limits: memmax = unlimited; ulimit-n = 8039
maxsock = 8039; maxconn = 4000; maxpipes = 0
current conns = 1; current pipes = 0/0; conn rate = 1/sec
Running tasks: 1/12; idle = 100 %

active UP
active UP, going down
active DOWN, going up
active or backup DOWN
active or backup DOWN for maintenance (MAINT)
active or backup SOFT STOPPED for maintenance
backup UP
backup UP, going down
backup DOWN, going up
not checked
Note: "NOLB"/"DRAIN" = UP with load-balancing disabled.

Display option:
- Scope :
- Hide 'DOWN' servers
- Refresh now
- CSV export

External resources:
- Primary site
- Updates (v1.5)
- Online manual

stats

	Queue			Session rate			Sessions					Bytes		Denied		Errors			Warnings		Server							
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				1	1	-	1	1	300	1			0	0	0	0	0					OPEN							
Backend	0	0		0	0		0	0	30	0	0	0s	0	0	0	0		0	0	0	0	9h5m UP		0	0	0		0		

prd

	Queue			Session rate			Sessions					Bytes		Denied		Errors			Warnings		Server							
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				0	3	-	0	2	300	11			1 046 343	458 426	0	0	0					OPEN							
srv1	0	0	-	0	3		0	2	-	9	9	1m41s	1 046 335	458 426		0		0	0	0	0	9h1m UP	L7OK/0 in 21ms	1	Y	-	3	3	1m41s	-
srv2	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	9h5m DOWN	L7RSP in 69ms	1	Y	-	1	1	9h5m	-
srv3	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	9h5m DOWN	L7RSP in 24ms	1	Y	-	1	1	9h5m	-
Backend	0	0		0	3		0	2	30	11	9	1m41s	1 046 343	458 426	0	0		2	0	0	0	9h1m UP		1	1	0		3	1m40s	

stb

	Queue			Session rate			Sessions					Bytes		Denied		Errors			Warnings		Server							
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				0	0	-	0	0	300	0			0	0	0	0	0					OPEN							
srv1	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	9h1m DOWN	L7RSP in 38ms	1	Y	-	2	2	9h5m	-
srv2	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	1h37m UP	L7OK/0 in 38ms	1	Y	-	3	3	2m36s	-
srv3	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	9h1m UP	L7OK/0 in 18ms	1	Y	-	3	3	1m54s	-
Backend	0	0		0	0		0	0	30	0	0	?	0	0	0	0		0	0	0	0	9h1m UP		2	2	0		2	1m49s	

STATS HAPROXY

Configuration:

```
global
    stats socket ipv4@*:9999
    stats socket /var/lib/haproxy/stats
```

Utilisation:

```
$ echo "show stat" | nc -U /var/lib/haproxy/stats | column -s, -t | less -S
$ echo "show stat" | ncat 10.20.30.51 9999 | column -s, -t | less -S
```

...ou en mode cli. Il est donc possible de collecter ces statistiques en CSV et de les exploiter eg. dans grafana.

À noter qu'il est aussi possible d'effectuer quelques actions d'administration au travers de ces sockets s'ils l'autorisent. Dans la configuration, ça donne:

```
stats socket ipv4@*:9999 level admin
```

Il est alors possible par exemple de déclarer un backend inaccessible administrativement:

```
$ echo show stat|ncat 10.20.30.51 9999|cut -d, -f 1,2,18,28,29|column -s, -t
# pxname  svname    status  iid  sid
stats     FRONTEND  OPEN    2    0
stats     BACKEND   UP      2    0
prd       FRONTEND  OPEN    3    0
prd       srv1      DOWN    3    1
prd       srv2      DOWN    3    2
prd       srv3      UP      3    3
prd       BACKEND   UP      3    0
stb       FRONTEND  OPEN    4    0
stb       srv1      UP      4    1
stb       srv2      UP      4    2
stb       srv3      DOWN    4    3
stb       BACKEND   UP      4    0

$ echo show stat 4 4 2|ncat 10.20.30.51 9999|cut -d, -f 1,2,18,28,29|column -s, -t
# pxname  svname  status  iid  sid
stb       srv2    UP      4    2

$ echo disable server stb/srv2|ncat 10.20.30.51 9999
```

```
$ echo show stat 4 4 2|ncat 10.20.30.51 9999|cut -d, -f 1,2,18,28,29|column -s, -t
# pxname  svname  status  iid  sid
stb       srv2    MAINT   4    2
```

TRAVAIL À LA MAISON

- Vagrantfile dispo pour reconstruire cette archi
- cf. README
  - remplacez `3nodes-vip` par `3nodes-haproxy`
- faites moi des retours

PROS

- pas de vIP, la contrainte de base
- équilibrage de charge possible vers les standby
- pgsql-state ouvert à tous
  - information dispo aux couches applicatives
- accès direct aux instances toujours possible !

Pour peu que les applications aient un peu d'intelligence, elles peuvent elles même choisir sur quel nœud se connecter en interrogeant `pgsql-state`, si elles veulent éviter le rebond de HAProxy.

Elles peuvent même contourner le HAProxy si nécessaire. Ce dernier se résume alors simplement aux flux de réplications et éventuellement à l'équilibrage de charge vers les standby.

CONS

- `pgsql-state` ouvert à tous
  - sécurité ? filtrer par un firewall ?
  - information dispo aux couches applicatives, pas nécessairement acceptable
- archi plus complexe qu'avec une vIP
- dépend de la disponibilité de Systemd
  - peu grave
- systemd.socket accepte par défaut 100cnx/s
  - le service passe en `failed`
  - doit être redémarré (automatiquement ?)
  - attention au DoS
  - se configure aisément

En cas de défaillance de systemd, ce qui est fort improbable, au pire haproxy coupera les connexions clientes et de réplication. Nous ne sommes donc pas dans une situation extrême comme dans le cas d'un *split brain*.

Concernant les 100cnx/s, les deux paramètres à modifier sont `TriggerLimitIntervalSec` et `TriggerLimitBurst`. Voir la page de manuel `systemd.socket` à ce propos.

PISTES DE TRAVAIL

- health checks
  - utilisation du `check-pgsql` intégré
    - pas d'authent sécurisée possible
  - édition du pg_hba lors de la bascule…
  - fermeture de port avec le RA `portblock`
  - autres idées ?
- keepalived/LVS

Le `tcp-check` permet d'envoyer et valider des données binaires. Voici un exemple qui se connecte, sans authentification, et effectue une requête SQL pour déterminer si l'instance est primaire ou non, le tout en binaire pour reproduire le protocole de PostgreSQL:

https://gist.github.com/arkady-emelyanov/af2993ab242f9a1ec0427159434488c4

Dans un article, Percona propose de modifier le contenu du pg_hba lors des bascules pour rejeter un rôle spécifique en fonction de l'état de chaque instance. Plutôt hacky.

https://www.percona.com/blog/2019/11/08/configure-haproxy-with-postgresql-using-built-in-pgsql-check/

Il semble que le projet keepalived soit capable de configurer du LVS et effectuer de l'équilibrage de charge. À creuser, notamment pour ce qui est des health check. Il pourrait être plus performant dans les modes tunnel ou routage, bien moins lourd que la ré-écriture de paquet comme du NAT.

Voir à ce propos le chapitre "LVS CONFIGURATION" dans la documentation officielle: https://www.keepalived.org/manpage.html

DÉMO

```
$ ncat --recv-only 10.20.30.53 5431
standby

$ pgbench -i -s 10 -h 10.20.30.53
dropping old tables...
creating tables...
generating data...
100000 of 1000000 tuples (10%) done (elapsed 0.04 s, remaining 0.36 s)
[...]

$ for i in {1..6}; do psql -XAtc 'show cluster_name' "host=10.20.30.53 port=5432" & do
pgsql-srv1
pgsql-srv1
pgsql-srv1
pgsql-srv1
pgsql-srv1
pgsql-srv1

$ for i in {1..6}; do psql -XAtc 'show cluster_name' "host=10.20.30.53 port=5433" & do
pgsql-srv2
pgsql-srv3
pgsql-srv2
pgsql-srv2
pgsql-srv3
pgsql-srv3
```