Module R58

Patroni : Mise en œuvre

DALIBO
L'expertise PostgreSQL

24.04

Table des matières

Sur ce document

Formation	Module R58
Titre	Patroni : Mise en œuvre
Révision	24.04
PDF	https://dali.bo/r58_pdf
EPUB	https://dali.bo/r58_epub
HTML	https://dali.bo/r58_html
Slides	https://dali.bo/r58_slides
TP	https://dali.bo/r58_tp

Vous trouverez en ligne les différentes versions complètes de ce document.

Chers lectrices & lecteurs,

Nos formations PostgreSQL sont issues de nombreuses années d'études, d'expérience de terrain et de passion pour les logiciels libres. Pour Dalibo, l'utilisation de PostgreSQL n'est pas une marque d'opportunisme commercial, mais l'expression d'un engagement de longue date. Le choix de l'Open Source est aussi le choix de l'implication dans la communauté du logiciel.

Au-delà du contenu technique en lui-même, notre intention est de transmettre les valeurs qui animent et unissent les développeurs de PostgreSQL depuis toujours : partage, ouverture, transparence, créativité, dynamisme… Le but premier de nos formations est de vous aider à mieux exploiter toute la puissance de PostgreSQL mais nous espérons également qu'elles vous inciteront à devenir un membre actif de la communauté en partageant à votre tour le savoir-faire que vous aurez acquis avec nous.

Nous mettons un point d'honneur à maintenir nos manuels à jour, avec des informations précises et des exemples détaillés. Toutefois malgré nos efforts et nos multiples relectures, il est probable que ce document contienne des oublis, des coquilles, des imprécisions ou des erreurs. Si vous constatez un souci, n'hésitez pas à le signaler via l'adresse formation@dalibo.com[1] !

À propos de DALIBO

DALIBO est le spécialiste français de PostgreSQL. Nous proposons du support, de la formation et du conseil depuis 2005.

Retrouvez toutes nos formations sur https://dalibo.com/formations

[1] mailto:formation@dalibo.com

Remerciements

Ce manuel de formation est une aventure collective qui se transmet au sein de notre société depuis des années. Nous remercions chaleureusement ici toutes les personnes qui ont contribué directement ou indirectement à cet ouvrage, notamment :

Jean-Paul Argudo, Alexandre Anriot, Carole Arnaud, Alexandre Baron, David Bidoc, Sharon Bonan, Franck Boudehen, Arnaud Bruniquel, Pierrick Chovelon, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Jehan-Guillaume de Rorthais, Ronan Dunklau, Vik Fearing, Stefan Fercot, Pierre Giraud, Nicolas Gollet, Dimitri Fontaine, Florent Jardin, Virginie Jourdan, Luc Lamarle, Denis Laxalde, Guillaume Lelarge, Alain Lesage, Benoit Lobréau, Jean-Louis Louër, Thibaut Madelaine, Adrien Nayrat, Alexandre Pereira, Flavie Perette, Robin Portigliatti, Thomas Reiss, Maël Rimbault, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Cédric Villemain, Thibaud Walkowiak, Frédéric Yhuel.

Forme de ce manuel

Les versions PDF, EPUB ou HTML de ce document sont structurées autour des slides de nos formations. Le texte suivant chaque slide contient le cours et de nombreux détails qui ne peuvent être données à l'oral.

Licence Creative Commons CC-BY-NC-SA

Cela inclut les diapositives, les manuels eux-mêmes et les travaux pratiques. Cette formation peut également contenir quelques images et schémas dont la redistribution est soumise à des licences différentes qui sont alors précisées.

Marques déposées

PostgreSQL® Postgres® et le logo Slonik sont des marques déposées[3] par PostgreSQL Community Association of Canada.

Versions de PostgreSQL couvertes

Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 12 à 16.

Sur les versions précédentes susceptibles d'être encore rencontrées en production, seuls quelques points très importants sont évoqués, en plus éventuellement de quelques éléments historiques.

Sauf précision contraire, le système d'exploitation utilisé est Linux.

[3] https://www.postgresql.org/about/policies/trademarks/

1/ Patroni : Mise en œuvre

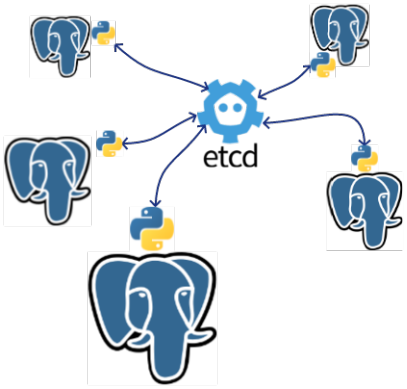


Figure 1/ .1: PostgreSQL

1.1 AU MENU

- Architecture générale
- Patroni
- Proxy, VIP et poolers de connexions

Sur les bases de la réplication physique de PostgreSQL, Patroni fournit un modèle de gestion d'instances, avec bascule automatique et configuration distribuée.

Les tâches nécessaires pour la bascule ou la promotion, l'ajout d'un nouveau nœud, la resynchronisation suite à une défaillance, deviennent la responsabilité de Patroni qui veille à ce que ces actions soient effectuées de manière fiable, en évitant toujours la multiplicité de nœuds primaires (*split-brain*).

Dans ce module, nous décrivons l'architecture de cette solution de haute disponibilité de service et sa mise en œuvre.

1.2 ARCHITECTURE GÉNÉRALE

- Haute disponibilité de service
- Instances gérées **uniquement** par Patroni
- Nécessite deux agrégats de serveurs :
  - DCS (etcd)
  - Patroni
- Synchronisation des horloges
  - attention aux *snapshots* !

Patroni

Patroni assure la haute disponibilité d'un service PostgreSQL.

L'application des modifications de la configuration de PostgreSQL est effectuée par Patroni qui se charge de la répercuter sur tous les nœuds.

Le démarrage et l'arrêt du service PostgreSQL sur chaque nœud ne doivent plus être gérés par le système et doivent être désactivés. Toutes les actions de maintenances (arrêt, démarrage, rechargement de configuration, promotion) doivent être faites en utilisant Patroni plutôt que les moyens traditionnels (`pg_ctl` , `systemctl` , etc).

DCS

Patroni s'appuie sur un gestionnaire de configuration distribuée (DCS) pour partager l'état des nœuds de son agrégat et leur configuration. Dans ce module, nous utilisons etcd comme DCS.

Notre but étant la haute disponibilité, etcd ne doit pas devenir un SPOF (*single point of failure*). Il doit donc lui aussi être déployé en agrégat afin d'assurer une tolérance aux pannes et une disponibilité maximale du service. Un agrégat etcd utilise le protocole RAFT pour assurer la réplication et cohérence des données entre ses nœuds.

Nous sommes donc en présence de deux agrégats de serveurs différents :

- l'un pour les nœuds etcd ;
- l'autre pour les nœuds Patroni gérant chacun une instance PostgreSQL.

Horloges système

La synchronisation des horloges de tous les serveurs Patroni et etcd est primordiale. Elle doit idéalement être assurée par le protocole NTP[1].

Les ralentissements causés par les *snapshot* de sauvegardes des machines virtuelles illustrent ce type de problème. Une indisponibilité (*freeze*) trop grande peut entraîner un décalage d'horloge trop important, une bascule automatique et une désynchronisation du nœud retardataire. Celui-ci est alors dans l'impossibilité de se raccrocher au nouveau primaire et sa reconstruction est inévitable.

L'anomalie se signale par exemple dans les traces d'etcd par le message :

```
etcd: the clock difference […] is too high
```

[1] *Network Time Protocole*

1.3 DÉFINITIONS

Patroni permet de :

- créer / mettre en réplication
- maintenir
- superviser

des serveurs PostgreSQL en haute disponibilité.

Patroni est un script écrit en Python. Il permet de maintenir un agrégat d'instances PostgreSQL en condition opérationnelle et de le superviser afin de provoquer une bascule automatique en cas d'incident sur le primaire.

Il s'agit donc d'un outil permettant de garantir la haute disponibilité du service de bases de données.

1.3.1 Mécanismes mis en œuvre

- Réplication physique et outils standards (`pg_rewind` , `pg_basebackup`)
- Sauvegarde PITR (barman, pgBackRest…)
- Gestionnaire de service : Patroni
- DCS : Serveur de configurations distribuées (etcd ou autres)

Patroni est un script Python qui s'appuie sur la capacité de l'écosystème PostgreSQL à répliquer les modifications et les rejouer sur un stockage clé valeur distribué pour garantir la haute disponibilité de PostgreSQL.

Outils :

La réplication physique fournie avec PostgreSQL assure la haute disponibilité des données. Des outils fournis avec PostgreSQL comme pg_rewind[2] et pg_basebackup[3] sont utilisés pour construire ou reconstruire les instances secondaires. Nous les décrivons dans le module de formation I4[4].

[2] https://www.postgresql.org/docs/current/app-pgrewind.html
[3] https://www.postgresql.org/docs/current/app-pgbasebackup.html
[4] https://dali.bo/i4_html

Cette capacité est étendue grâce à la possibilité d'utiliser des outils de la communauté comme barman[5], pgBackRest[6] ou WAL-G[7].

DCS (etcd) :

Patroni s'appuie sur un gestionnaire de configuration distribué (DCS) pour partager l'état des nœuds de son agrégat et leur configuration commune.

Nous ne mentionnerons dans ce document que etcd, mais il est cependant possible d'utiliser un autre DCS tel que Consul, ZooKeeper ou même Kubernetes. Tous peuvent être déployés en haute disponibilité de service.

Raft **dans** Patroni :
L'algorithme Raft utilisé dans etcd a également été implémenté dans Patroni en version 2. Cependant, cette option est déjà dépréciée dans la version 3 de Patroni et ne doit pas être utilisée.

1.3.2 Bascule automatique

- *leader lock*
- *heart beat*
- *split-brain*
- *followers* demandant une élection au quorum
- Promotion automatique

L'automatisation de la prise de décision de bascule est protégée par un mécanisme de verrou partagé appelé *leader lock*. Ce verrou est attribué à une seule instance secondaire suite à une élection basée sur sa disponibilité et son LSN courant (*Log Sequence Number*, ou position dans le flux des journaux de transaction).

La présence de deux primaires dans un agrégat d'instance nous amène à une situation que nous voulons éviter à tout prix : le *split-brain*.

[5] https://www.pgbarman.org/
[6] https://pgbackrest.org/
[7] https://github.com/wal-g/wal-g

1.3.3 Définition : *split-brain*

- 2 primaires sollicités en écriture
- Arbitrage très difficile
- Perte de données
- Indisponibilité du service

La situation d'un *split-brain* est obtenue lorsque l'élection automatique d'un primaire est possible sur deux nœuds différents, au même moment.

Les données insérées sur les deux nœuds doivent faire l'objet d'un arbitrage pour départager lesquelles seront gardées lors du rétablissement d'une situation normale (un seul primaire).

La perte de données est plus probable lors de cet arbitrage, suivant la quantité et la méthode d'arbitrage.

Le service peut souffrir d'une indisponibilité s'il y a nécessité de restauration partielle ou totale des données.

1.3.4 Leader lock de Patroni

- Verrou attribué au primaire de manière unique
- Communication entre les nœuds Patroni
  - comparaison des LSN
- Nouveau primaire :
  - nouvelle *timeline*
  - nouvelle chaîne de connexions
  - les secondaires se raccrochent au primaire

L'attribution unique d'un verrou appelé *leader lock* par Patroni permet de se prémunir d'un *split-brain*. Ce verrou est distribué et stocké dans le DCS.

Une fois ce verrou obtenu, le futur primaire dialogue alors avec les autres nœuds Patroni référencés dans le DCS, et valide sa promotion en comparant leur type de réplication (synchrone ou asynchrone) et leur LSN courant.

La promotion provoque la création d'une nouvelle *timeline* sur l'instance primaire et la chaîne de connexion (`primary_conninfo`) utilisée par les instances secondaires pour se connecter au primaire est mise à jour.

Les secondaires se raccrochent ensuite à la *timeline* du primaire.

1.3.5 Heartbeat

- Tous les nœuds Patroni s'annoncent à etcd
  - primaire ou *follower*
- si perte de contact avec le leader :
  - *timeout* sur les *followers* et bascule

Chaque nœud est en communication régulière avec l'agrégat etcd afin d'informer le système de sa bonne santé. Le primaire confirme son statut de leader et les secondaires celui de *follower*.

Lorsque la confirmation du leader ne vient pas, un *timeout* est atteint sur les *followers* Patroni, déclenchant alors une procédure de bascule.

1.3.6 Bootstrap de nœud

- Création ou recréation de nœud
- Reconstruction :
  - pg_basebackup depuis primaire
  - pgBackRest depuis sauvegarde PITR (delta !)

L'opération de *bootstrap* consiste à recréer l'instance PostgreSQL d'un nœud, à partir du nœud primaire ou d'une sauvegarde. Par défaut, Patroni lance un `pg_basebackup` pour récupérer l'instance du primaire.

Cependant, cela n'est pas toujours souhaitable, notamment sur des gros volumes. Il est alors possible de paramétrer Patroni pour utiliser un autre outil de restauration de sauvegarde PITR, tel que pgBackrest.

Pour reconstruire un nœud ayant pris trop de retard, pgBackrest permet de raccrocher rapidement une instance ayant un volume de données important grâce à la restauration en mode delta.

Si l'archivage ou la sauvegarde sont endommagés, Patroni utilise l'alternative `pg_basebackup` pour recréer l'instance.

1.3.7 Répartition sur deux sites

- Prévoir la perte d'un des 2 sites
- Quorum impossible
  - au pire, passage en *read only* !

Le but de configurer un deuxième site est de disposer d'une tolérance de panne à l'échelle d'un site. En cas d'incident majeur, le deuxième site est censé prendre la relève.

Cependant, dans le cas d'un agrégat étendu sur deux sites, il devient impossible de différencier la perte totale d'une salle distante d'une coupure réseau locale. Il est tentant de « favoriser » une salle en y positionnant une majorité de nœuds de l'agrégat, mais cette approche réduit au final la disponibilité de service. Détaillons :

- Cas 1 : primaire et majorité sur le site A
  - si perte du site A : le site B n'a pas le quorum, pas de bascule
  - si perte du site B : aucun impact sur le primaire
- Cas 2 : primaire sur le site A, majorité sur le site B
  - si perte du site A : bascule vers le site B
  - si perte du site B : perte du quorum, **l'instance passe en mode *stand-by* sur le site 0**

Dans le cas d'un agrégat multi-site, la réponse à cette problématique est d'utiliser au minimum trois sites. En cas de perte de réseau d'un des sites, ce dernier perd alors le quorum et les instances PostgreSQL ne peuvent y être qu'en *standby*. Les deux autres sites continuent à communiquer entre eux et conservent ainsi le quorum. L'instance primaire peut être démarrée ou maintenue sur l'un de ces deux sites sans risque de *split-brain*.

En conclusion : sans troisième site, il n'est pas possible de mettre en œuvre une mécanique de bascule automatique fiable et anti *split-brain*.

1.3.8 Répartition sur trois sites

- 3 sites pour un quorum
- Tolérance de panne accrue
- Changement de site lors d'une bascule
- 3e site en *standby* en cas de perte de 2 sites

Quorum de sites

La présence de trois sites permet de disposer de suffisamment de nœuds pour construire le quorum nécessaire à la bascule automatique.

En effet, on a vu que pour départager deux sites en concurrence, il est nécessaire de disposer d'un arbitre externe et donc d'un troisième site.

Tolérance de panne accrue

La tolérance de panne peut être bien plus importante si l'on décide de prévoir la perte totale de deux sites sur les trois, mais elle apporte une complexité supplémentaire et ne permet pas un automatisme complet jusqu'au bout (après la perte de deux sites).

À tout moment, le maintien du service nécessite de disposer de :

- un nombre suffisant de nœuds pour le DCS, plus précisément d'un quorum de nœuds fonctionnels, ce qui dépend du nombre total de nœuds déclarés dans l'agrégat et de leur répartition entre les trois sites ;
- suffisamment de nœuds Patroni sur chaque site, pour que malgré la perte totale d'un ou deux sites, nous puissions disposer d'une tolérance de panne minimale, sur chaque site.

En conséquence, une solution hautement disponible peut être de disposer de trois nœuds pour le DCS et de deux nœuds Patroni au minimum, ceci pour chaque site.

Changement de site lors des bascules

Lors de la bascule automatique, Patroni décide de privilégier les nœuds les plus à jour avec le primaire, puis les plus réactifs. Il n'y a donc aucune garantie que dans certaines conditions particulières, un nœud d'un autre site soit promu plutôt qu'un nœud géographiquement local.

Perte de deux sites sur trois

S'il ne reste plus qu'un site disponible, promouvoir un de ses nœuds secondaires ne pourra être fait qu'après plusieurs opérations **manuelles** visant à **reconstruire un quorum** [8] **de nœuds dans le DCS** en enlevant les nœuds défaillants de sa liste (les nœuds des deux autres sites).

[8] Le quorum se calcule ainsi : (3 sites × 3 nœuds / 2) +1

Pour rappel, il faut toujours retirer les nœuds défaillants du DCS avant d'en ajouter pour les remplacer, sous peine de ne jamais récupérer de quorum malgré l'ajout.

Troisième site en *standby*

Une autre possibilité consiste à garder le troisième site en dehors du mécanisme de bascule automatique. Il est prévu de ne le solliciter que manuellement, après avoir constaté la perte totale des deux premiers sites.

1.4 INSTALLATION

- Paquets disponibles pour les distributions EL
- Paquets disponibles pour les distributions Debian
- Installez PostgreSQL avec Patroni !

Patroni est empaqueté pour les principales distributions Linux existantes, qu'elles soient dérivées d'Entreprise Linux (Red Hat, Rocky Linux…) ou de Debian. L'installation ne dure que quelques minutes.

Bien entendu, Patroni a besoin des binaires de PostgreSQL afin d'administrer une instance locale. Utilisez votre méthode favorite ou consultez les méthodes recommandées dans notre module sur l'installation[9].

1.4.1 Sur Entreprise Linux

- Utilisation des dépôts communautaires PGDG
- Nécessite d'activer le dépôt EPEL

Les paquets Patroni sont disponibles pour les distributions Entreprise Linux depuis les dépôts communautaires PGDG. Ceux-ci utilisent par ailleurs des dépendances provenant du dépôt EPEL.

Il faut donc au préalable installer les paquets suivants sur tous les nœuds :

- `epel-release`
- sur EL 7 : https://download.postgresql.org/pub/repos/yum/reporpms/EL-7-x86_64/pgdg-redhat-repo-latest.noarch.rpm
- sur EL 8 : https://download.postgresql.org/pub/repos/yum/reporpms/EL-8-x86_64/pgdg-redhat-repo-latest.noarch.rpm
- sur EL 9 : https://download.postgresql.org/pub/repos/yum/reporpms/EL-9-x86_64/pgdg-redhat-repo-latest.noarch.rpm

Pour plus d'information à propos des dépôts PGDG, voir : https://yum.postgresql.org/howto/

Sur chaque nœud PostgreSQL, nous pouvons désormais installer Patroni :

```
dnf install -y patroni-etcd
```

[9] https://dali.bo/b_html#installation-de-postgresql-depuis-les-paquets-communautaires

Notez que ce paquet installe les paquets `patroni` et `python3-etcd` par dépendance, ce dernier étant la librairie cliente d'etcd en python. Il n'installe pas la partie serveur d'etcd, cette-ci devant idéalement être située sur des nœuds distincts.

Pour plus d'information à propos de l'installation d'un cluster etcd, voir le module de formation etcd : Architecture et fonctionnement[10].

1.4.2 Sur Debian et dérivés

- Paquet `patroni` disponible
- Versions plus à jour dans les dépôts PGDG communautaires
  - gérées par les mainteneurs Debian officiels

Debian et ses dérivés incluent directement Patroni et etcd dans ses dépôts officiels. Néanmoins, étant donné la politique de gestion des versions des paquets Debian, les versions de Patroni peuvent rapidement être désuètes.

De nouvelles versions de Patroni sont régulièrement publiées, incluant des corrections, améliorations et nouveautés. C'est pourquoi nous vous recommandons d'utiliser autant que faire se peut une version récente.

Pour se faire, nous proposons d'installer les dépôts PGDG avant d'installer Patroni sur les nœuds PostgreSQL :

```
# apt install postgresql-common gnupg
# /usr/share/postgresql-common/pgdg/apt.postgresql.org.sh -y
# apt install -y patroni
```

1.4.3 Installation manuelle

- Utilisation du gestionnaire de paquets Python `pip`
- Installe Patroni mais aussi ses dépendances

Pour les autres distributions Linux ou si certaines contraintes vous empêchent d'utiliser les méthodes précédentes, il est possible d'installer Patroni grâce au gestionnaire de paquets Python `pip`.

[10] https://dali.bo/r57_html

Le nom du paquet Python à installer est `patroni`. Cependant, si vous utilisez cette méthode manuelle, il vous sera probablement aussi nécessaire de spécifier la dépendance sur le DCS à utiliser, ici etcd, avec le format `patroni[etcd]`. Par exemple :

```
pip install patroni[etcd3]
```

Vous trouverez la liste des dépendances disponibles dans la documentation de Patroni : https://patroni.readthedocs.io/en/latest/installation.html#general-installation-for-pip

1.5 CONFIGURATIONS

- Configuration statique
  - stockée dans le fichier de configuration YAML de chaque nœud
  - recharge par `patronictl reload`
- Configuration dynamique
  - stockée dans le DCS
  - initialisée depuis la section `bootstrap.dcs` du fichier de configuration YAML
  - modifiable ensuite par `patronictl edit-config`
  - prise en compte immédiatement si possible
  - copiée dans `$PGDATA/patroni.dynamic.json` à intervalle régulier
- Variables d'environnement

La configuration de patroni[11] est répartie sur trois niveaux différents.

La **configuration statique** peut désigner soit un fichier YAML, soit un répertoire. Les fichiers par défaut sont `/etc/patroni/config.yml` sous Debian et dérivés et `/etc/patroni/patroni.yml` sous les EL et dérivés.

Si un répertoire est spécifié, tous les fichiers YAML du répertoire seront chargés dans l'ordre où ils apparaissent. Si des paramètres sont définis deux fois, seule la dernière valeur est prise en compte.

Les modifications faites dans la configuration statique peuvent être chargées par l'une des méthodes suivantes :

- le rechargement du service (si disponible) ;
- la commande `patronictl reload` ;
- un signal `SIGHUP` (*reload*) au processus `patroni`.

Il est possible de générer un modèle de configuration avec la commande suivante :

```
patroni --generate-sample-config /chemin/vers/fichier.yml
```

Cette commande doit pouvoir accéder aux binaires de PostgreSQL. Il peut donc être nécessaire de positionner la variable d'environnement `PATH` en fonction de votre installation.

[11] https://patroni.readthedocs.io/en/latest/dynamic_configuration.html

Les éléments de configuration suivants sont présents dans ce fichier :

- configuration de l'API REST de Patroni (section `restapi`) ;
- configuration des interactions avec un des DCS supporté, dont etcd (section `etcd` ou `etcd3` pour ce dernier) ;
- configuration des interactions avec PostgreSQL (section `postgresql`) et de sa configuration (`postgresql.conf`, `pg_hba.conf`, `pg_ident.conf`) ;
- configuration du *watchdog* ;
- configuration concernant la création de l'instance (section `bootstrap`).

Toutes les informations présentes dans la section `bootstrap` sont uniquement utilisées lors de la création du cluster. Cela inclut la configuration dynamique qui est dans la section `bootstrap.dcs`, mais aussi la configuration de PostgreSQL effectuée dans les sections `bootstrap.initdb`, `bootstrap.method`, `bootstrap.pg_hba`, `bootstrap.users`.

Les modifications des autres sections de cette configuration doivent être reportées dans le fichier de configuration de tous les nœuds si l'on souhaite qu'elles soient prises en compte globalement. Cela peut être fait avec des outils d'industrialisation comme Ansible, Chef,…

Le fichier de configuration peut être testé avec l'exécutable de Patroni :

```
patroni --validate-config /etc/patroni/config.yml
```

Le second niveau de configuration est la **configuration dynamique**. Elle est initialisée grâce aux données de la section `bootstrap.dcs` du fichier de configuration YAML. Elle est chargée dans le DCS et n'est plus maintenue qu'à cet endroit. Les modifications doivent être faites avec la commande `patronictl edit-config`. Patroni copie cette configuration à intervalle régulier dans le fichier `patroni.dynamic.json` placé dans le répertoire de données de l'instance.

Pour finir, certains éléments de configuration peuvent être spécifiés *via* des **variables d'environnement**. Les variables d'environnement ont toujours précédence sur les fichiers de configuration.

L'application de la configuration locale ou dynamique de PostgreSQL suit l'algorithme suivant :

- **SI** `postgresql.custom_conf` est défini dans la configuration YAML de patroni
  - **ALORS** le fichier configuré dans `custom_conf` sera utilisé comme référence de configuration en lieu et place de `postgresql.base.conf` et `postgresql.conf`
  - **SINON SI** le fichier `postgresql.base.conf` existe
    * **ALORS** il sera utilisé comme référence de la configuration
    * **SINON** le fichier `postgresql.conf` est utilisé et renommé en `postgresql.base.conf`
- La configuration dynamique est écrite dans `postgresql.conf` à l'exception de certains paramètres qui seront passés directement en paramètre du `postmaster` (voir plus loin).
- Un `include` vers la configuration de référence est ajouté au fichier `postgresql.conf`
- **SI** des paramètres nécessitent un redémarrage (`pg_settings.context`)

  - **ALORS** un flag `pending_restart` est donné au nœud et sera retiré lors du prochain redémarrage.

L'ordre de prise en compte des paramètres de Postgresql est donc le suivant :

1. paramètres présents dans `postgresql.base.conf` ou spécifiés dans `custom_conf` ;
2. paramètres présents dans `postgresql.conf` ;
3. paramètres présents dans `postgresql.auto.conf` ;
4. paramètres défini au lancement de PostgreSQL avec l'option `-o --name=value`.

1.5.1 Paramètres globaux du cluster

Patroni définit trois paramètres globaux au cluster :

- `name`
- `namespace`
- `scope`

Les paramètres globaux (de premier niveau) de la configuration de Patroni sont :

`name` : Le nom de l'hôte, doit être unique au sein du cluster.

`namespace` : Le chemin dans lequel est stockée la configuration de Patroni dans le DCS, par défaut: `/service`.

`scope` : Le nom de l'agrégat, utilisé dans les commandes `patronictl` et pour stocker la configuration de l'agrégat dans le DCS. Ce paramètre est requis et doit être identique sur tous les nœuds.

Exemple de fichier de configuration :

```
name: p1
scope: acme
[…]
```

Et voici un exemple de l'arborescence créée dans le DCS pour la configuration ci-dessus et avec deux serveurs **p1** et **p2** :

```
$ ETCDCTL_API=3 etcdctl get --prefix / --keys-only
/service/acme/config
/service/acme/initialize
/service/acme/leader
/service/acme/members/pgsql0.hapat.vm
/service/acme/members/pgsql1.hapat.vm
/service/acme/status
```

1.5.2 Configuration du DCS

- DCS supportés : etcd, Consul, ZooKeeper, Exhibitor, Kubernetes
- avec `etcd` ou `etcd3` :
  - `host`
  - `protocol`
  - `username`, `password`
  - `cacert`, `cert`, `key`

Patroni supporte six types de DCS différents : etcd, Consul, Zookeeper, Exhibitor, Kubernetes.

Dans cette formation, nous n'abordons que l'utilisation d'etcd, qui dispose de deux sections en fonction de la version du protocole d'etcd utilisée :

- `etcd` pour le protocole v2 ;
- `etcd3` pour le protocole v3.

L'API v3 est configurée par défaut dans etcd depuis qu'il est en version 3.4. Son implémentation dans Patroni est considérée comme stable pour la production depuis sa version 2.1.5. Il est recommandé de l'utiliser lorsque les versions utilisées le permettent car l'API v2 est progressivement dépréciée :

- elle est désactivée par défaut depuis la version 3.4 d'etcd
- elle sera supprimée dans la version 3.6 d'etcd

Les paramètres suivants permettent de configurer l'accès au DCS :

`host` et `hosts` : Permettent de définir au choix un ou une liste de nœud etcd au format `host:port`.

`protocol` : Le protocole utilisé parmi `http` et `https`, par défaut à `http`.

`username` : Utilisateur pour l'authentification à etcd.

`password` : Mot de passe pour l'authentification à etcd.

`cacert` : Le certificat du serveur d'autorité de certification, active le SSL si présent.

`cert` : Certificat du client.

`key` : Clé du client, peut être ignorée si elle fait partie du certificat.

D'autres paramètres sont décrits dans la documentation officielle.

Exemple :

```
[…]
etcd3:
  hosts:
   - 10.20.89.56:2379
   - 10.20.89.57:2379
   - 10.20.89.58:2379
  username: patroniaccess
  password: secret
[…]
```

1.5.3 Configuration de Patroni

Le paramétrage de Patroni est :

- initialisé depuis la section `bootstrap.dcs`
- conservé à la racine du YAML dynamique

Ci-après un exemple de configuration visible après l'initialisation de l'instance via la commande `patronictl show-config` :

```
$ patronictl -c /etc/patroni/config.yml show-config
loop_wait: 10
master_start_timeout: 300
postgresql:
   parameters:
      archive_command: /bin/true
      archive_mode: 'on'
   use_pg_rewind: false
   use_slot: true
retry_timeout: 10
ttl: 30
```

La configuration du démon Patroni est représentée à la racine de la section `bootstrap.dcs` puis dans YAML dynamique. Elle comprend les paramètres suivants :

`loop_wait` : Temps de pause maximal entre chaque boucle de vérification, par défaut `10s`. Cela correspond au temps nominal de la boucle de vérification.

`ttl` : Temps avant l'initialisation d'un *failover*, par défaut `30s`. Cela correspond au temps maximal entre deux mises à jour de la *leader key*.

`retry_timeout` : Temps avant de retenter une action échouée sur PostgreSQL ou le DCS, par défaut `10s`.

`maximum_lag_on_failover` : Limite supérieure du délai (*lag*), en octets, pour qu'un nœud *follower* puisse participer à une élection.

`max_timelines_history` : Quantité maximale de changements de *timeline* conservés dans l'historique stocké dans le DCS, par défaut `0`. Pour une valeur de `0`, Patroni conserve tout l'historique.

`master_start_timeout` : Le temps maximal autorisé pour qu'une instance primaire redevienne fonctionnelle suite à un incident, par défaut `300`. Si la valeur est nulle et que l'état du cluster le permet, Patroni déclenche un *failover* immédiat suite à un incident, ce qui peut provoquer une perte de donnée dans le cas d'une réplication asynchrone.

`master_stop_timeout` : Le temps maximal autorisé pour l'arrêt de PostgreSQL lorsque le mode synchrone est activé, par défaut à `0`. Si la valeur est supérieure à zéro et que le mode synchrone est activé, Patroni envoie un signal `SIGKILL` au postmaster s'il met trop de temps à s'arrêter.

`synchronous_mode` : Activation de la réplication synchrone. Dans ce mode, un ou des secondaires sont choisis comme *followers* synchrones. Seuls ces secondaires et le leader peuvent participer à une élection.

`synchronous_node_count` : Nombre de *followers* synchrones que Patroni doit chercher à maintenir. Pour cela, Patroni choisit des *followers* synchrones parmi les secondaires disponibles et valorise le paramètre `synchronous_standby_names` dans la configuration de PostgreSQL en listant uniquement ces nœuds (ex: `2(p2,p3)`). Si un nœud disparaît de l'agrégat, Patroni le retire de `synchronous_standby_names`. Par défaut, Patroni fait en sorte de ne pas bloquer les écritures en diminuant le nombre de nœuds synchrones jusqu'à désactiver la réplication synchrone si nécessaire. Les *followers* synchrones sont aussi listés dans l'entrée `/namespace/scope/sync` de etcd.

`synchronous_mode_strict` : Empêche la désactivation du mode synchrone dans le cas où il n'y a plus de *follower* synchrone. Pour cela, Patroni positionne le paramètre `synchronous_standby_names` à `*` quand il n'y a plus de candidats, ce qui bloque les écritures en attendant le retour d'un *follower* synchrone. Ce paramètre peut donc provoquer une indisponibilité du service.

`maximum_lag_on_syncnode` : Limite supérieure du *lag* en octets au delà de laquelle un nœud *follower* synchrone est considéré comme non sain et remplacé par un *follower* asynchrone sain. Une valeur inférieure ou égale à zéro désactive ce comportement. Une valeur trop basse peut provoquer des changements de *follower* synchrone trop fréquents. Par défaut à `-1`.

`check_timeline` : Vérifie que la *timeline* du candidat est bien la plus élevée avant d'effectuer la promotion. Désactivé par défaut.

`failsafe_mode` : Ce mode permet d'éviter le déclenchement d'une opération de démote sur l'instance primaire lorsque le DCS est indisponible. Pour cela, une nouvelle clé `/failsafe` est ajoutée au DCS. Elle est maintenue par l'instance primaire et contient la liste des membres autorisés à participer à une élection. En cas de perte du DCS, si tous les membres listés dans la clé `failsafe` sont joignables, l'instance primaire garde son rôle. Si l'un d'entre eux ne répond pas, l'instance primaire perd son rôle de leader. Désactivé par défaut.

1.5.4 Création des instances

Il est possible d'indiquer à Patroni comment construire les instances primaires et secondaires d'un agrégat. Les sections concernées sont :

- `bootstrap.method` et `bootstrap.initdb`
- `postgresql.create_replica_methods`
- `bootstrap.users`
- scripts `bootstrap.post_bootstrap` et `bootstrap.post_init`

La section `bootstrap.method` permet de décrire la manière dont l'instance PostgreSQL principale doit être créée. La méthode `initdb` est la méthode par défaut, mais il est possible d'utiliser d'autres commandes[12].

L'exemple suivant utilise une sauvegarde pgBackRest pour initialiser le cluster. La commande à utiliser est spécifiée avec le paramètre `command`. Si l'option `no_params` est à `False` les paramètres `--scope` et `--datadir` qui définissent respectivement le nom du cluster et le chemin de l'instance seront ajoutés à la commande. Pour finir, le paramètre `keep_existing_recovery_conf` permet de conserver le `recovery.conf` généré par pgBackRest.

```
bootstrap:
   […]
   method: pgbackrest
   pgbackrest:
      command: /bin/bash -c "pgbackrest --stanza=test restore"
      keep_existing_recovery_conf: True
      no_params: True
[…]
```

Si la méthode `initdb` est choisie, une section spécifique doit être renseignée pour en modifier les paramètres. Cela permet par exemple de spécifier si les sommes de contrôles doivent être activées (`data-checksums`) et de définir une locale et un encodage :

```
bootstrap:
   […]
   # ici la méthode est facultative, initdb est la valeur par défaut
   method: initdb
   initdb:
     - data-checksums
     - encoding: UTF8
     - locale: UTF8
<...>
```

Les instances secondaires doivent être construites comme des clones de la primaire, une commande spécifique est donc dédié à leur création. Comme ces secondaires peuvent être

[12] https://patroni.readthedocs.io/en/latest/replica_bootstrap.html#custom-bootstrap

créés à n'importe quel moment de la vie du cluster, cette commande se situe dans la section `postgresql.create_replica_method`, conservée dans le DCS.

Cette section peut contenir plusieurs méthodes qui seront testées dans l'ordre d'apparition. Patroni permet l'utilisation de `pg_basebackup` (par défaut), pgBackRest, WAL-G, barman ou d'un script utilisateur pour réaliser cette opération.

Il est possible de fournir des paramètres à l'outil que l'on souhaite utiliser. Tous les paramètres configurés seront communiqués sous la forme `--<nom>=<valeur>` ou `--<nom>`. Trois paramètres sont cependant réservés et ne seront pas passés aux scripts :

`no_master` : Permet d'utiliser une méthode même si aucune instance n'est démarrée dans l'agrégat. Désactivé par défaut.

`no_param` : Permet de ne pas utiliser les paramètres supplémentaires décrits ci-après. Désactivé par défaut.

`keep_data` : Indique à Patroni de ne pas vider le répertoire de données de l'instance avant la réinitialisation. Désactivé par défaut.

Les paramètres suivant seront également fourni au script si l'option `no_params` reste a `False` :

`scope` : Nom de l'agrégat.

`datadir` : Chemin vers le répertoire de donnée de l'instance secondaire.

`role` : Toujours valorisé à `replica`.

`connstring` : Chaîne de connexion vers le nœud depuis lequel la copie va être réalisée.

Exemple de configuration (dynamique) des méthodes de création des réplicas :

```
[…]
postgresql:
    […]
    create_replica_methods:
        - pgbackrest
        - basebackup
    pgbackrest:
        command: /usr/bin/pgbackrest --stanza=patroni_demo --delta restore
        keep_data: True
        no_master: True
        no_params: True
    basebackup:
        keep_data: False
        no_master: False
        no_params: False
        verbose
        max-rate: '100M'
        waldir: /pg_wal/14/pg_wal
    […]
```

La documentation est disponible à cet emplacement : https://patroni.readthedocs.io/en/latest/replica_bootstrap.html#custom-replica-creation.

Vient ensuite la section `bootstap.users`, permettant de créer et configurer des rôles PostgreSQL supplémentaires. Par exemple :

```
bootstrap:
   […]
   users:
      admin:
         password: password_admin
         options:
          - createrole
          - createdb
[…]
```

Enfin, des scripts peuvent être déclenchés après l'initialisation de l'instance avec les sections `bootstrap.post_bootstrap` ou `bootstrap.post_init`. Le script reçoit en paramètre une URL de connexion avec comme utilisateur le super utilisateur de l'instance et est appelé avec la variable d'environnement `PGPASSFILE` positionnée.

1.5.5 Configuration de PostgreSQL

- Configuration de l'agrégat et des instances
- Initialisée depuis la section `bootstrap.dcs.postgresql`
- Conservée dans la section `postgresql` du YAML dynamique
  - `postgresql.authentication`
  - `postgresql.parameters`
  - `postgresql.pg_hba`
  - `postgresql.pg_ident`
  - `postgresql.callbacks`

La configuration de l'agrégat PostgreSQL à déployer et/ou maintenir peut être chargée dans le DCS au moment de l'initialisation depuis la section `bootstrap.dcs.postgresql`. Elle est ensuite conservée dans le YAML dynamique dans la section `postgresql`. Cette section permet d'avoir une configuration commune pour toutes les instances. Les paramètres suivants sont disponibles à ce niveau :

`connect_address` : Adresse IP locale sur laquelle PostgreSQL est accessible pour les autres nœuds et applications utilisés au sein du cluster, sous la forme `[IP|nom d'hôte]:port`.

`data_dir` : Emplacement du répertoire de données.

`config_dir` : L'emplacement des fichiers de configuration, par défaut défini à la valeur de `data_dir`.

`bin_dir` : Chemin vers les binaires de PostgreSQL : `pg_ctl`, `pg_rewind`, `pg_basebackup` et `postgres`. Si cette valeur n'est pas configurée, la variable d'environnement `PATH` est utilisée pour trouver les exécutables.

`listen` : Liste d'adresses sur lesquelles PostgreSQL écoute. Elles doivent être accessibles par les autres nœuds. Il est possible d'utiliser une liste sous la forme `{IP|nom d'hôte}[,…]:port`, dans ce cas la première adresse sera utilisée par Patroni pour ses connexions au nœud local. Ce paramètre est utilisé pour valoriser les paramètres `listen_addresses` et `port` de PostgreSQL. Il est possible d'utiliser `*` plutôt qu'une liste d'IP.

`use_unix_socket` : Indique à Patroni de préférer une connexion par socket pour accéder au nœud local. Désactivé par défaut.

`use_unix_socket_repl` : Permet de dire à Patroni de préférer une connexion par socket pour la réplication. Désactivé par défaut.

`pgpass` : Chemin vers un fichier `.pgpass`. Il est préférable de laisser Patroni gérer ce fichier et de ne pas ajouter nos propres entrées car elles pourraient être supprimées.

`recovery_conf` : Configuration supplémentaire à ajouter lors de la configuration d'un nœud *follower*. Même si le fichier `recovery.conf` disparaît à partir de la version 12 de PostgreSQL, la section porte toujours ce nom.

`custom_conf` : Chemin vers un fichier de configuration a utiliser à la place de `postgresql.base.conf`. Le fichier doit exister et être accessible par Patroni et PostgreSQL. Patroni ne surveille pas ce fichier pour détecter des modifications et ne le sauvegarde pas. Il est simplement inclus depuis le fichier `postgresql.conf`.

`pg_ctl_timeout` Temps d'attente maximale pour les actions effectuées avec `pg_ctl` (`start`, `stop`, `restart`), par défaut à `60`.

`use_slots` Utilisation des slots de réplication, activé par défaut pour les versions de PostgreSQL supérieures à la 9.4.

`use_pg_rewind` : Utilise `pg_rewind` pour reconstruire l'ancien primaire en tant que secondaire après un *failover*, par défaut à `false`.

`remove_data_directory_on_rewind_failure` : Force la suppression du répertoire de donnée de l'instance en cas d'erreur lors du `pg_rewind`. Désactivé par défaut.

`remove_data_directory_on_diverged_timelines` : Supprime le répertoire de données si les *timelines* divergent entre le secondaire et le primaire.

`pre_promote` : Permet d'exécuter un script pendant un *failover* après l'acquisition du *leader lock* et avant la promotion d'un réplica. Si le script renvoie un code différent de zéro, Patroni n'effectue pas la promotion et relâche la *leader key*. Ce paramètre est principalement utile pour mettre en œuvre un mécanisme de *fencing*.

Exemple de configuration basique :

```
<...>
postgresql:
  listen: "*:5432"
  connect_address: 10.20.89.54:5432
  data_dir: /var/lib/pgsql/11/data
  bin_dir: /usr/pgsql-11/bin
  pgpass: /var/lib/pgsql/.pgpass_patroni
  <...>
```

Plusieurs sous-sections complètent cette configuration de PostgreSQL.

La sous-section `postgresql.authentication` permet d'indiquer à Patroni quels utilisateurs choisir pour :

- `superuser` : ses tâches courantes ;
- `replication` : la configuration de la réplication ;
- `rewind` : l'exécution de la commande `pg_rewind`.

Pour chaque type d'utilisateur, les éléments de configuration suivants sont disponibles et correspondent aux paramètres de connexion éponymes : `username`, `password`, `sslmode`, `sslkey`, `sslpassword`, `sslcert`, `sslrootcert`, `sslcrl`, `sslcrldir`, `gssencmode` et `channel_binding`.

Exemple de section `authentication` :

```
[…]
postgresql:
   […]
   authentication:
      superuser:
         username: patronidba
         password: secret
      replication:
         username: replicator
         password: secretaussi
      rewind:
         username: rewinder
         password: sectrettoujours
   […]
```

La sous-section `postgresql.parameters`, contient les paramètres PostgreSQL maintenus par Patroni et stockés dans le DCS de manière inconditionnelle, soit parce que PostgreSQL requiert qu'ils soient identiques partout, soit par choix d'implémentation. Voici la liste de ces paramètres :

`max_connections` : Nombre maximal de connexions simultanées, par défaut à `100`.

`max_locks_per_transaction` : Nombre maximal de verrous par transaction, par défaut à `64`.

`max_worker_processes` : Nombre maximum de processus *worker*, par défaut à `8`.

`max_prepared_transactions` : Nombre maximal de transactions préparées, par défaut à `0`.

`wal_level` : Niveau de détail des informations écrites dans les WAL, défaut à `replica`.

`wal_log_hints` : Force l'écriture complète d'une page de données lors de sa première modification après un *checkpoint*, même pour des modifications non critiques comme celles des *hint bits*. Cela permet l'utilisation de `pg_rewind`. Activé par défaut.

`track_commit_timestamp` : Permet de tracer l'horodatage des commits dans les journaux de transactions. Désactivé par défaut.

`max_wal_senders` : Nombre maximal de processus *wal sender*, par défaut à `5`.

`max_replication_slots` : Nombre maximal de slots de réplication, par défaut à `5`.

`wal_keep_segments` : Nombre maximal de WAL conservés pour les instances secondaires afin de les aider à récupérer leur retard, par défaut à `8`. Disponible jusqu'en PostgreSQL 12.

`wal_keep_size` : Quantité de WAL conservés pour les instances secondaires afin de les aider à récupérer leur retard, par défaut à `128MB`. Disponible à partir de PostgreSQL 13.

`listen_addresses` : La ou les interfaces sur lesquelles PostgreSQL écoute. Ce paramètre est défini via le paramètre `postgresql.listen` ou la variable d'environnement `PATRONI_POSTGRESQL_LISTEN`.

`port` : Le port sur lequel écoute l'instance, lui aussi défini par le paramètre `postgresql.listen` ou la variable d'environnement `PATRONI_POSTGRESQL_LISTEN`.

`cluster_name` : Permet de définir le nom de l'instance qui sera affiché dans la description des processus (eg. par la commande `ps`). Il est défini en fonction de la valeur du paramètre `scope` ou de la variable d'environnement `PATRONI_SCOPE`.

`hot_standby` : Permet d'ouvrir les instances secondaires en lecture seule. Activé par défaut.

Afin que ces paramètres ne puissent pas être modifiés via les fichiers de configuration, ils sont passés en paramètre de la commande de démarrage de PostgreSQL. Ce n'est le cas **que** pour la liste ci-dessus. Les autres paramètres présents dans la section `postgresql.parameters` sont appliqués de manière classique en suivant la hiérarchie décrite précédemment.

Les sous-sections **`postgresql.pg_hba`** et **`postgresql.pg_ident`** contiennent chacune une liste de règles à ajouter aux fichiers respectifs, dans leurs formats respectifs. Voir l'exemple ci-après.

Enfin, la sous-section **`postgresql.callbacks`** permet d'exécuter des scripts lors des différents changements d'état du cluster. Trois paramètres sont communiqués aux scripts : l'action, le rôle du nœud et le nom du cluster.

Les actions suivantes sont disponibles :

- **`on_reload`** : rechargement de la configuration ;
- **`on_restart`** : redémarrage de PostgreSQL ;
- **`on_role_change`** : promotion ou passage de primaire à standby ;
- **`on_start`** : démarrage de PostgreSQL ;
- **`on_stop`** : arrêt de PostgreSQL.

Ci-après un exemple de configuration de la section `bootstrap.dcs` incluant la configuration de PostgreSQL :

```
bootstrap:
   dcs:
      […]
      postgresql:
         use_pg_rewind: true
         use_slots: true
         parameters:
            wal_level: replica
            hot_standby: "on"
            wal_keep_segments: 8
            max_wal_senders: 5
            max_replication_slots: 5
            checkpoint_timeout: 30
         pg_hba:
          - local all all  peer
          - host all all 0.0.0.0/0 scram-sha-256
          - host replication replicator 10.0.30.12/32 scram-sha-256
          - host replication replicator 10.0.30.13/32 scram-sha-256
         pg_ident:
          - superusermapping root postgres
          - superusermapping dba postgres
```

1.5.6 Agrégat de secours

- Initialisé depuis la section `bootstrap.dcs.standby_cluster`
- Lie deux agrégats Patroni distincts :
  - un agrégat contenant une instance primaire
  - un agrégat ne contenant que des instances secondaires
- Réplication en cascade vers l'agrégat *standby*

Patroni permet de créer un agrégat de secours appelé *standby cluster* en utilisant la réplication en cascade de PostgreSQL. Pour ce faire, un second agrégat est créé et son leader, appelé *standby leader*, se connecte à un serveur de l'agrégat principal via le protocole de réplication. Les *followers* de ce second agrégat se connectent, eux, au *standby leader*, pour répliquer les modifications.

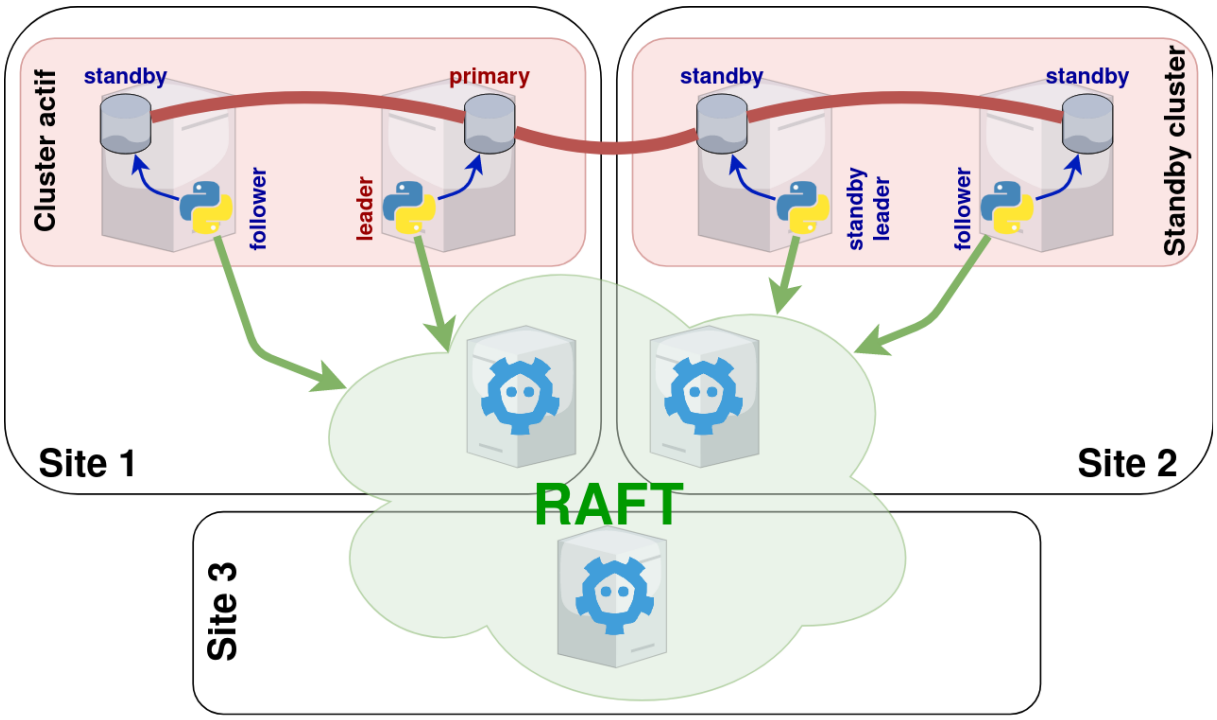


Figure 1/ .2: Standby cluster Patroni

Les clusters Patroni sur figure 1 utilisent le même agrégat etcd par simple commodité. Il n'y a aucune contrainte à ce propos, même si dans le cas de l'architecture présentée cela est tout indiqué.

La section `bootstrap.dcs.standby_cluster` est utilisée lorsque l'on crée un agrégat de secours[13]. Les éléments suivants sont définis pour cette section :

`host` : Adresse du serveur primaire.

`port` : Port du serveur primaire.

`primary_slot_name` : Indique le slot du serveur primaire à utiliser pour la réplication. Si ce paramètre n'est pas utilisé le nom sera dérivé du nom du serveur primaire. Pour cela, le nom est converti en Unicode, les espaces et tirets sont remplacés par des underscores et le nom est tronqué à 64 caractères. Le slot doit être créé manuellement et ne sera par défaut pas maintenu par Patroni.

`create_replica_methods` : Liste des méthodes disponibles pour initialiser le leader de l'agrégat de secours (voir l'option `bootstrap.method` décrite plus haut pour plus de détails).

`restore_command` : Commande utilisée pour récupérer les WAL via le *log shipping*.

`archive_cleanup_command` : Commande pour supprimer les journaux de transaction du répertoire d'archivage une fois qu'ils ne sont plus nécessaires.

[13] https://patroni.readthedocs.io/en/latest/replica_bootstrap.html#standby-cluster

`recovery_min_apply_delay` **(en millisecondes):** Délai d'application des modifications sur le leader de l'agrégat de secours, équivalent au paramètre de même nom de PostgreSQL.

Lorsque le service Patroni est démarré, il initialise le cluster en se connectant à l'instance spécifiée.

Afin de s'assurer que les WAL soient toujours disponibles pour l'agrégat de secours, il est possible d'utiliser un slot permanent avec la section `bootstrap.dcs.slots`, abordé plus loin.
Il est préférable de disposer d'une IP virtuelle (VIP) sur le serveur primaire de l'agrégat primaire. De cette façon, si on perd le serveur source de la réplication vers le *standby leader*, la réplication basculera vers un autre nœud. Il est possible qu'il y ait une divergence en cas de *failover* sur l'agrégat primaire.

Voici un exemple de configuration pour la création du leader d'un *standby cluster*. Le nom de cluster a été changé, une section `bootstap.dcs.standby_cluster` a été créé et le mot de passe de l'utilisateur de réplication a été configuré.

```
scope: acme-standby
name: p3
[…]
bootstrap:
  dcs:
    […]
    standby_cluster:
      host: 10.20.89.3
      port: 5432
[…]
postgresql:
  authentication:
    replication:
      username: replicator
      password: repass
[…]
```

On peut vérifier qu'un cluster a été créé dans le DCS pour les nœuds **p3** et **p4** (créé séparément) et que la configuration du *standby cluster* a été adaptée.

Les commandes suivantes sont lancées depuis un serveur etcd.

```
$ export ETCDCTL_API=3

$ etcdctl get --keys-only --prefix /service/acme-standby
/service/acme-standby/config
/service/acme-standby/initialize
/service/acme-standby/leader
/service/acme-standby/members/p3
/service/acme-standby/members/p4
/service/acme-standby/status

$ etcdctl get --print-value-only /service/acme-standby/config | python -m json.tool
```

```
{
  "ttl": 30,
  "loop_wait": 10,
  "retry_timeout": 10,
  "master_start_timeout": 300,
  "postgresql": {
    "use_pg_rewind": false,
    "use_slots": true,
    "parameters": {
      "archive_mode": "on",
      "archive_command": "/bin/true"
    }
  },
  "standby_cluster": {
    "host": "10.20.89.3",
    "port": 5432
  }
}
```

Le leader est marqué comme *standby leader* dans `patronictl` :

```
$ patronictl list
+--------+------------+----------------+---------+----+-----------+
| Member | Host       | Role           | State   | TL | Lag in MB |
+ Cluster: acme-standby (7148326914433478989) ---+----+-----------+
| p3     | 10.20.89.5 | Standby Leader | running |  1 |           |
| p4     | 10.20.89.6 | Replica        | running |  1 |         0 |
+--------+------------+----------------+---------+----+-----------+
```

1.5.7 Slots de réplication

Initialisés depuis les sections suivantes :

- `bootstrap.dcs.slots`
- `bootstrap.dcs.ignore_slots`

Patroni dispose de deux sections dédiées aux slots de réplication.

Lors d'une bascule, le comportement par défaut de Patroni lorsqu'il rencontre un slot de réplication qui n'est pas maintenu par lui est de le supprimer.

La section `bootstrap.dcs.slots`, puis `slots` dans la configuration dynamique, permet de définir des slots de réplication permanents qui seront préservés lors d'un *switchover* ou *failover*.

Les slots de réplication physique sont créés sur la nouvelle instance primaire au moment de la promotion ou de leur définition dans la configuration.

Les slots de réplication logique sont créés sur tous les nœuds et leur position est avancée sur les *followers* à chaque fois que `loop_wait` est écoulé. L'utilisation de cette fonctionnalité requiert la pré-

sence du paramètre `postgresql.use_slots`. De plus, le paramètre `hot_standby_feedback` est automatiquement activé sur les instances secondaires par Patroni.

Après la mise en place de la configuration, si les slots définis dans cette section n'existent pas, Patroni va tenter de les créer. Si c'est le résultat attendu, il faut vérifier que la création a bien fonctionné. Le paramétrage des slots (nom de base ou de plugin) n'est pas vérifié lors de la validation de la configuration, toute erreur fera donc échouer la création du slot.

Attention, la suppression des slots dans la configuration supprime aussi les slots sur les instances.

Ces slots doivent être décrits dans la section `slots` en créant une sous-section portant le nom du slot qui contient les paramètres suivants :

`type` : Le type du slot : `physical` ou `logical`.

`database` : Le nom de la base de données pour laquelle le slot de réplication logique est créé.

`plugin` : Le plugin de décodage utilisé par le slot de réplication logique.

Le nommage du slot de réplication persistant doit être fait en gardant à l'esprit qu'il faut éviter les collisions de nom avec les slots créés par Patroni pour les besoins de la réplication en flux utilisée par l'agrégat.

La section `ignore_slots` permet de donner à Patroni une liste de slots de réplication à ignorer. Chaque slot défini dans cette section est défini par :

`name` : Un nom de slot.
`type` : Un type de slot: `logical` ou `physical`.
`database` : Le nom de la base de données sur laquelle un slot de réplication Logique est défini.
`plugin` : Un plugin de décodage logique utilisé par le slot.

Voici un exemple de configuration des slots :

```
bootstrap:
   dcs:
      […]
      slots:
         replication_logique:
            type: logical
            database: magasin
            plugin: test_decoding
         standby_cluster:
            type: physical
      ignore_slots:
       - name: replication_logique_app
         type: logical
         database: magasin
         plugin: test_decoding
       - name: standby_hors_aggregat
         type: physical
```

1.5.8 Journaux applicatifs

- Par défaut dans le fichier de trace système
- Quelques paramètres de la section `log` :
  - `level` : niveau de log
  - `format` : format des lignes
  - `dir` : répertoire où placer les journaux
  - `file_num` : nombre de journaux à conserver
  - `file_size` : taille maximale d'un journal avant rotation

Patroni étant écrit en python, la configuration des traces devrait être familière aux utilisateurs de ce langage.

Les paramètres suivants peuvent être configurés pour contrôler le contenu et l'emplacement des traces :

`level` : Niveau de trace parmi `CRITICAL`, `ERROR`, `WARNING`, `INFO` et `DEBUG`. Par défaut `INFO`.

`traceback_level` : Niveau de trace à partir duquel les tracebacks sont visibles, par défaut `ERROR`.

`format` : Format des traces, défaut : `%(asctime)s %(levelname)s: %(message)s`. Voir : https://docs.python.org/3.10/library/logging.html#logrecord-attributes.

`dateformat` : Format de date. Voir : https://docs.python.org/3.10/library/logging.html#logging.Formatter.formatTime.

`max_queue_size` : Patroni génère une trace en deux temps. Il écrit les traces en mémoire et un thread séparé s'occupe de reporter ces traces vers un fichier ou la sortie standard. La quantité de traces gardée en mémoire est par défaut de 1000 enregistrements.

`dir` : Le répertoire où sont écrits les journaux. Patroni doit avoir les droits en écriture sur ce répertoire.

`file_num` : Nombre de journaux applicatifs à conserver, par défaut `4`.

`file_size` : Taille maximale d'un journal applicatif avant qu'un nouveau ne soit créé, par défaut `25MB`.

`loggers` : Cette section permet de définir un niveau de trace par module Python.

Patroni écrit lui même ses journaux applicatifs si et seulement si le paramètre `dir` est positionné. Sinon, les traces sont envoyées vers la sortie standard, habituellement capturée vers les journaux système par `journald` et/ou `syslog`.

Exemple où Patroni écrit ses journaux dans le répertoire `/var/log/patroni` :

```
[…]
log:
   level: INFO
   dir: /var/log/patroni
[…]
```

1.5.9 API REST de Patroni

Patroni expose une API REST :

- utile à son administration, par ex. via le CLI `patronictl`
- utilisée pour la communication inter-démons
- section `restapi` :
  - `connect_address`
  - `listen`
  - `authentication` (`username`, `password`)
  - SSL (`certfile`, `keyfile`, `keyfile_password`, `cafile`, `verify_client`)

L'accès à l'API REST peut être contrôlé grâce aux paramètres de la section `restapi` :

`listen` : Permet de définir les adresses et le port sur lesquelles Patroni expose son API REST. Elle est notamment utilisée par les autres membres de l'agrégat pour vérifier la santé du nœud lors d'une élection. Cette adresse peut également servir aux *health checks* d'outils comme HAProxy, la supervision et les connexions utilisateurs.

`connect_address` : Adresse IP et port fournis aux autres membres pour l'accès à l'API REST de Patroni. Information stockée dans le DCS.

`proxy_address` : Adresse et port pour joindre le pool de connexion ou proxy qui permet d'accéder à PostgreSQL. Une entrée `proxy_url` est crée dans le DCS afin de faciliter la découverte de service.

`authentication` : Permet de définir un `username` et `password` autorisant l'accès à l'API REST.

`certfile` : Certificat au format PEM. Active le SSL si présent.

`keyfile` : Clé secrète au format PEM.

`keyfile_password` : Mot de passe pour déchiffrer la clé.

`cafile` : Spécifie le certificat de l'autorité de certification.

`verify_client` : Définis quand la clé est requise : * `none` (défaut) : l'API REST ne vérifie pas les certificats ; * `requiered` : les certificats clients sont requis pour tous les accès a l'API REST ; * `optional` : les certificats ne sont requis que pour les accès marqués comme sensibles (appels `PUT`, `POST`, `PATCH` et `DELETE`).

`allowlist` : Liste d'hôtes autorisés à accéder aux API définies comme sensibles. Les noms d'hôtes, adresses IP ou sous-réseaux sont autorisés. Par défaut tout est autorisé.

`allowlist_include_members` : Autorise les membres de l'agrégat à accéder aux API sensibles. L'adresse IP est récupérée à partir du paramètre `api_url` stocké dans le DCS : attention à ce que ce soit bien l'IP utilisée pour accéder à l'API REST !

Il existe des paramètres supplémentaires permettant d'adapter les en-têtes HTTP ou HTTPS. Voir : https://patroni.readthedocs.io/en/latest/yaml_configuration.html#rest-api

Voici un exemple qui autorise les accès aux API sensibles uniquement depuis les membres de l'agrégat :

```
name: p1
scope: acme
[…]
restapi:
   listen: 0.0.0.0:8009
   connect_address: 10.20.89.54:8009
   allowlist_include_members: true
[…]
```

On peut voir que l'adresse définie dans `connect_address` est reportée dans le DCS sous le nom `api_url` :

```
$ export ETCDCTL_API=3
$ etcdctl get --print-value-only \
  '/service/acme/members/p1' | python -m json.tool

{
    "api_url": "http://10.20.89.54:8008/patroni",
    "conn_url": "postgres://10.20.89.54:5432/postgres",
    "pending_restart": true,
    "role": "master",
    "state": "running",
    "timeline": 1,
    "version": "2.1.4",
    "xlog_location": 50331968
}
```

Ci-après un exemple de commande lancée depuis le serveur **p2** et modifiant la configuration en utilisant l'API REST du serveur **p1**. La modification réussit :

```
# curl -s -XPATCH -d '{
>     "loop_wait": 10,
>     "master_start_timeout": 300,
>     "postgresql": {
>         "parameters": {
```

```
>             "archive_command": "/bin/true",
>             "archive_mode": "on",
>             "max_connections": 101
>         },
>         "use_pg_rewind": false,
>         "use_slot": true
>     },
>     "retry_timeout": 10,
>     "ttl": 30
> }' 10.20.89.54:8008/config | python -m json.tool

{
    "loop_wait": 10,
    "master_start_timeout": 300,
    "postgresql": {
        "parameters": {
            "archive_command": "/bin/true",
            "archive_mode": "on",
            "max_connections": 101
        },
        "use_pg_rewind": false,
        "use_slot": true
    },
    "retry_timeout": 10,
    "ttl": 30
}
```

La même commande lancée depuis un autre serveur se termine avec le message suivant :

```
Access is denied
```

1.5.10 API REST pour le CLI patronictl

Une configuration spécifique est réservée au CLI `patronictl` :

- section : `ctl`
  - `insecure`
  - `certfile`
  - `keyfile`
  - `keyfile_password`
  - `cacert`

Le [#CLI `patronictl`] livré avec Patroni permet d'effectuer différentes tâches d'administration. Cet outil pouvant être utilisé depuis le poste d'un administrateur, la section de configuration `ctl` est spécifiquement réservée à son mode d'authentification :

`insecure` : Autorise les connexions l'API REST sans vérification des certificats SSL.
`certfile` : Certificat au format PEM (active le SSL si présent).
`keyfile` : Clé secrète au format PEM.
`keyfile_password` : Mot de passe pour décrypter la clé.
`cacert` : Certificat d'autorité de certification.

1.5.11 Configuration du watchdog

- méthode de protection anti split-brain
- section `watchdog` :
  - `mode` : `off`, `automatic` ou `required`
  - `device`
  - `safety_margin`

Afin d'éviter une situation de *split-brain*, Patroni doit s'assurer que l'instance PostgreSQL d'un nœud n'accepte plus de transaction une fois que la *leader key* qui lui est associées expire. En temps normal, Patroni essaie d'obtenir cette garantie en arrêtant l'instance. Cependant cette opération peut échouer si :

- Patroni plante à cause d'un bug, un problème de mémoire ou le processus est tué par une source extérieure ;
- l'arrêt de PostgreSQL est trop lent ;
- Patroni ne fonctionne pas à cause d'une charge trop importante sur le serveur, ou un autre problème d'infrastructure ou d'hyperviseur.

Afin que le cluster réagisse correctement dans ces situations, Patroni supporte l'utilisation d'un *watchdog*. Un *watchdog* est un composant doté d'un compte à rebours qui, au moment où il expire, éteint ou redémarre physiquement le serveur *sur-le-champ*. En conséquence, un logiciel, ici Patroni, doit donc recharger continuellement ce *watchdog timeout* (*WDT*) avant qu'il n'expire. Le destin du serveur est donc directement lié à la bonne exécution de Patroni.

Activation et Désactivation

Patroni tente d'armer le *watchdog* sur le nœud qui devient leader avant la promotion de l'instance PostgreSQL. Si l'utilisation d'un *watchdog* est requise (voir `watchdog.mode`) et que le *watchdog* ne s'active pas, le nœud refuse de devenir leader.

Un test est également réalisé lorsqu'un nœud décide de participer à l'élection du primaire et que le *watchdog* est requis sur ce dernier. Dans ce cas Patroni vérifie que le *device* associé au *watchdog* existe

(voir `watchdog.device`) et est accessible. Il contrôle également que le timeout du *watchdog* est supérieur ou égal à la durée nominale d'une boucle (`loop_wait`).

Lorsqu'une instance perd le statut de leader ou que Patroni est mis en pause, le *watchdog* est désactivé.

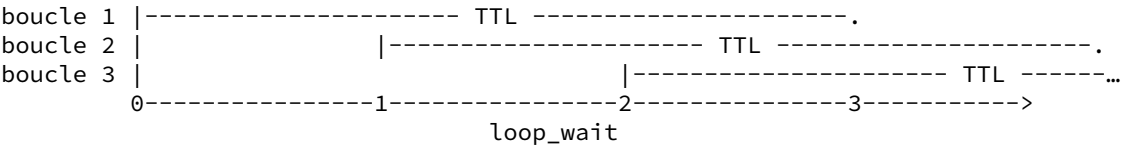
Mécanique et paramétrage

Par défaut Patroni configure le *watchdog* pour expirer 5 secondes avant que le `ttl` n'expire, c'est le paramètre `safety_margin`. Patroni calcule donc le *watchdog timeout* grâce à la formule suivante : `WDT = ttl - safety_margin`.

Ces 5 secondes laissent une marge de sécurité avant que le *leader key* n'expire. Elles permettent de garantir que l'ancien primaire est bien arrêté au moment où le `ttl` expire, ce qui déclenche alors une nouvelle élection. Nous évitons ainsi une situation de *split-brain* en cas d'incident ou de blocage.

Pour bien comprendre comment configurer `ttl`, `safety_margin`, `loop_wait` et `retry_timeout`, intéressons-nous au fonctionnement interne de Patroni.

La boucle de haute disponibilité est exécutée au moins toutes les 10 secondes par défaut, c'est le paramètre `loop_wait`. À la fin de chaque exécution, le processus calcule combien de temps il doit patienter avant sa prochaine exécution pour respecter au mieux cette période de `loop_wait` secondes. Dans les cas extrêmes (charge, lenteur, etc), il se ré-exécute sur-le-champs pour rattraper son retard. Le `ttl` étant de 30 secondes, Patroni a l'équivalent de trois exécutions de boucles pour le recharger.

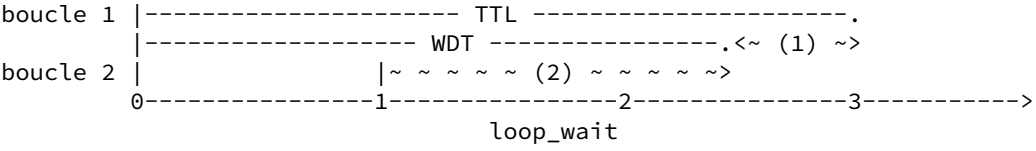


À chaque exécution de la boucle, après avoir déterminé que l'instance est primaire, Patroni doit :

1. recharger/valider le TTL du *leader key* ;
2. en cas de succès, recharger immédiatement le *watchdog timeout* ;
3. en cas d'échec, immédiatement déclasser l'instance en standby.

En temps normal, le temps écoulé entre les actions 1 et 2 est négligeable. Le *watchdog* expire donc environ `safety_margin` secondes *avant* le TTL de la *leader key*, ce qui est désiré.

Il faut aussi tenir compte qu'au début d'une boucle, avant que toute action ne commence, le WDT a dans le meilleur des cas été rechargé il y a déjà `loop_wait` secondes, lors de l'exécution de la précédente boucle. Par défaut, la boucle a donc 15 secondes pour effectuer ses toutes premières actions.




```
(1) =~ safety_margin
(2) =~ TTL - loop_wait - safety_margin = 30 - 10 - 5 = 15s
```

Ces différents paramètres et calculs en tête, intéressons-nous maintenant à deux scénarios d'incidents.

Le premier cas concerne une coupure de service avec le DCS. Dans cette situation, nous devons ici considérer le paramètre `retry_timeout` qui, dans le cas du leader, exprime le temps d'attente maximal d'une réponse du DCS avant de le considérer comme perdu. Par défaut, cette attente est de 10 secondes. Au moment où Patroni tente de mettre à jour le TTL de son *leader lock*, sans réponse du DCS après ces 10 secondes, Patroni déclenche une opération de *demote* pour déclasser l'instance en mode secondaire sur-le-champs. Notez qu'il ne recharge alors **pas** le *watchdog*, la situation du *leader lock* étant indéfinie. En conséquence, à ce moment précis, avec le paramétrage par défaut, Patroni n'a plus que 5 secondes pour effectuer l'opération de demote : `WDT - loop_wait - retry_timeout = 25 - 10 - 10 = 5s`

```
boucle 1 |---------------------- TTL ----------------------.
         |------------------- WDT ----------------.
boucle 2 |                |~ ~ ~ (1) ~ ~ ~ ><~(2)~>
         0----------------1----------------2---------------3----------->
                                loop_wait

(1) retry_timeout
(2) temps de demote = WDT - loop_wait - retry_timeout  = 25 - 10 - 10 = 5s
```

Si l'instance ne s'arrêtait pas proprement dans les temps, l'arrêt brutal par Patroni dans ce genre de situation lorsque la réplication synchrone est activée (voir paramètre `master_stop_timeout`).

Le second scénario implique une réponse du DCS anormalement longue (charge, coupure réseau, etc) ou un incident gelant la machine entre les deux recharges des TTL et WDT. Or, si le *watchdog* est rechargé plus que `safety_margin` secondes après le TTL, alors le WDT expire malheureusement **après** le prochain TTL.

```
boucle 1 |---------------------- TTL ----------------------.
         |<~ ~ (1) ~ ~>------------------- WDT ----------------.
         |                                                  <(2)>
         0----------------1----------------2---------------3----------->
                                loop_wait

(1) gel > safety_margin
(2) gel + WDT > TTL, le watchdog expire après le TTL
```

Ce cas est relativement peu probable, mais possible. De plus, il peut très bien survenir sans aucune conséquence pour le cluster. Rappelez-vous que la boucle est de 10 secondes par défaut, les TTL et WDT pourraient très bien être rechargés correctement à la prochaine boucle, longtemps avant leurs expirations respectives. Mais dans le pire des cas, la charge du DCS pourrait par exemple être continuelle reproduisant ainsi cet événement systématiquement, rendant le *watchdog* inefficace pour protéger votre cluster d'un *split-brain*.

Si l'on ne souhaite pas courir ce risque, il est possible de réduire le *watchdog timeout*. En contrepartie, il faut alors soit augmenter `ttl`, soit diminuer `loop_wait` et/ou `retry_timeout`. Pour illustrer cela, avec `safety_margin = -1 = ttl / 2` (valeur du *WDT* dans les anciennes versions de Patroni avant que `safety_margin` n'apparaisse), si nous ne modifions par ces autres paramètres, avec les calculs expliqués précédemment nous obtenons alors :

- `WDT = ttl / 2 = 30 / 2 = 15 secondes` ;
- le temps disponible au début d'une boucle pour réaliser l'ensemble de toutes ses actions, y compris recharger les TTL et WDT n'est plus que de 5 secondes : `WDT - loop_wait = 15 - 10 = 5 secondes` ;
- en cas de ralentissement du DCS, le *watchdog* peut se déclencher avant même d'avoir eu la validation du rechargement du TTL : `WDT - loop_wait - retry_timeout = 15 - 10 - 10 = -5 secondes`

Avec une telle configuration, le cluster est donc beaucoup plus sensible. De plus :

- augmenter `ttl` : retarde le déclenchement du *failover* ;
- diminuer `loop_wait` : augmente la consommation de ressource de Patroni ainsi que le nombre d'accès au DCS. Cela rend donc l'architecture plus sensible à tous types de ralentissements ;
- diminuer `retry_timeout` : rends le système plus sensible aux problèmes réseaux ou charge du DCS.

Quoi qu'il en soit, en cas de gel du serveur PostgreSQL, de charge importante sur les DCS ou les instances, ou encore d'incident réseau, le bon correctif reste de régler le problème à la racine.

Voici un résumé des paramètres de la section `watchdog` qui permettent de configurer le *watchdog* :

`mode` : Le *watchdog* peut être désactivé (`off`), être activé si c'est possible (`automatic`) ou être obligatoire (`required`). Dans ce dernier mode, si le *watchdog* ne peut pas s'activer, le nœud ne peut pas devenir leader. Par défaut à `automatic` .

`device` : Chemin vers le *watchdog*. Par défaut `/dev/watchdog` .

`safety_margin` : Marge de sécurité entre le déclenchement du *watchdog* et l'expiration de la *leader key*. Par défaut à `5` .

1.5.12 Marqueurs d'instance

Patroni supporte différents marqueurs dans la section `tags` :

- `nofailover`
- `clonefrom`
- `noloadbalance`
- `replicatefrom`
- `nosync`

Patroni permet de configurer des marqueurs pour adapter le fonctionnement des nœuds dans la section `tags` du fichier YAML :

`nofailover` : Interdit la promotion du nœud. Désactivé par défaut.

`clonefrom` : Définis le nœud comme source privilégiée pour l'initialisation des secondaires. Si plusieurs nœuds sont dans ce cas, la source est choisie au hasard. Désactivé par défaut.

`noloadbalance` : Si activé, le nœud renvoie le code HTTP 503 pour l'accès au *endpoint* `GET /repica` ce qui l'exclut du *load balancing*. Désactivé par défaut.

`replicatefrom` : L'adresse IP d'un autre réplica utilisé pour faire de la réplication en cascade.

`nosync` : Le nœud ne peut être sélectionné comme réplica synchrone.

Il est possible d'ajouter des marqueurs spécifiques. Voici un exemple de configuration :

```
[…]
tags:
  noloadbalance: true
  montag: "mon tag a moi"
```

Les marqueurs sont visibles depuis `patronictl list` :

```
$ patronictl list
+--------+-------------+---------+---------+-[…]-+----------------------+
| Member | Host        | Role    | State   | […] | Tags                 |
+ Cluster: acme (7147602572400925478)[…]---+-[…]-+----------------------+
| p1     | 10.20.89.54 | Leader  | running | […] | montag: mon tag a moi|
|        |             |         |         | […] | noloadbalance: true  |
+--------+-------------+---------+---------+-[…]-+----------------------+
| p2     | 10.20.89.55 | Replica | running | […] |                      |
+--------+-------------+---------+---------+-[…]-+----------------------+
```

1.5.13 Agrégat multi-nodes CITUS

- Permet de simplifier le déploiement d'un cluster Citus
- Paramètres :
  - `group`
  - `database`

Patroni permet de déployer un cluster multi-nodes CITUS[14]. Le CLI `patronictl` a également été adapté pour afficher les informations sur les groupes de serveurs Citus.

[14] https://docs.citusdata.com/en/stable/installation/multi_node.html

1.5.14 Variables d'environnement

- Tous les paramètres existent en tant que variables d'environnement
- Deux sont utiles au quotidien :
  - `PATRONICTL_CONFIG_FILE`
  - `PATRONI_SCOPE`

Il est possible d'utiliser des variables d'environnement pour configurer la plupart des éléments présentés précédemment. On choisit cependant généralement d'utiliser le fichier de configuration YAML pour cela.

La liste complète est disponible à cette adresse : https://patroni.readthedocs.io/en/latest/ENVIRONMENT.html#environment-configuration-settings

Certaines variables sont cependant utiles au quotidien et méritent d'être chargées au démarrage de la session de l'utilisateur destiné à manipuler `patronictl`. Notamment :

`PATRONICTL_CONFIG_FILE` : Permet de spécifier l'emplacement du fichier de configuration ce qui évite de spécifier l'option `-c` à chaque fois.

`PATRONI_SCOPE` : Permet de spécifier le nom de l'agrégat ce qui évite la plupart du temps d'avoir à saisir le nom de l'agrégat dans les commandes.

1.6 CLI PATRONICTL

- Interagit avec l'agrégat
- Depuis n'importe quelle machine

`patronictl` permet d'interagir avec l'agrégat pour modifier son comportement ou consulter son état. Avec la bonne configuration et arguments, il est possible de l'utiliser depuis n'importe quelle machine, et n'importe quel utilisateur, l'utilisateur système **postgres** y compris.

On peut indiquer l'emplacement du fichier de configuration dans la variable `PATRONICTL_CONFIG_FILE` et ainsi s'affranchir de l'option `-c (--config-file)` de la commande `patronictl`.

```
$ export PATRONICTL_CONFIG_FILE=/etc/patroni/config.yml
$ patronictl topology
+ Cluster: acme (6876375338380834518) ---+---------+-----+-----------+
| Member | Host           | Role         | State   |  TL | Lag in MB |
+--------+----------------+--------------+---------+-----+-----------+
| pg-1   | 10.0.3.85:5434 | Leader       | running | 122 |           |
| + pg-2 | 10.0.3.35:5434 | Sync Standby | running | 122 |         0 |
| + pg-3 | 10.0.3.70:5434 | Sync Standby | running | 122 |         0 |
+--------+----------------+--------------+---------+-----+-----------+
```

La commande utilise les informations contenues dans le DCS comme base de ses actions, mais doit aussi pouvoir atteindre les API REST des démons Patroni.

Le fichier de configuration doit au minimum contenir le paramètre `scope`. Si le DCS n'y est pas présent, il est possible de désigner l'un des nœuds via l'argument `--dcs-url` (`-d` ou `--dcs`). Toutes les commandes suivantes sont équivalentes :

```
$ cat ./acme.yml
scope: acme

$ patronictl -c ./acme -d etcd3://10.20.89.56 topology

$ cat ./acme.yml
scope: acme
etcd3:
  hosts:
   - 10.20.89.56:2379
   - 10.20.89.57:2379
   - 10.20.89.58:2379

$ patronictl -c ./acme.yml topology
```

```
$ export PATRONICTL_CONFIG_FILE=./acme.yml
$ patronictl topology
```

La liste complète des commandes est disponible par l'option `--help` :

```
$ patronictl --help
Usage: patronictl [OPTIONS] COMMAND [ARGS]...

Options:
  -c, --config-file TEXT  Configuration file
  -d, --dcs TEXT          Use this DCS
  -k, --insecure          Allow connections to SSL sites without certs
  --help                  Show this message and exit.

Commands:
  configure    Create configuration file
  dsn          Generate a dsn for the provided member, defaults to a dsn of...
  edit-config  Edit cluster configuration
  failover     Failover to a replica
  flush        Discard scheduled events
  history      Show the history of failovers/switchovers
  list         List the Patroni members for a given Patroni
  pause        Disable auto failover
  query        Query a Patroni PostgreSQL member
  reinit       Reinitialize cluster member
  reload       Reload cluster member configuration
  remove       Remove cluster from DCS
  restart      Restart cluster member
  resume       Resume auto failover
  scaffold     Create a structure for the cluster in DCS
  show-config  Show cluster configuration
  switchover   Switchover to a replica
  topology     Prints ASCII topology for given cluster
  version      Output version of patronictl command or a running Patroni...
```

1.6.1 Consultation d'état

- `list` : liste les membres d'un agrégat par ordre alphabétique
- `topology` : affiche la topologie d'un agrégat

```
$ patronictl list
+ Cluster: acme (6876375338380834518) ---+---------+-----+-----------+
| Member | Host           | Role         | State   |  TL | Lag in MB |
+--------+----------------+--------------+---------+-----+-----------+
| pg-1   | 10.0.3.85:5434 | Sync Standby | running | 123 |         0 |
| pg-2   | 10.0.3.35:5434 | Leader       | running | 123 |           |
| pg-3   | 10.0.3.70:5434 | Sync Standby | running | 123 |         0 |
+--------+----------------+--------------+---------+-----+-----------+
```

La commande `topology` affiche la liste des nœuds sous la forme d'un arbre débutant par le primaire courant, suivi des nœuds secondaires.

```
$ patronictl topology
+ Cluster: acme (6876375338380834518) ---+---------+-----+-----------+
| Member | Host           | Role         | State   |  TL | Lag in MB |
+--------+----------------+--------------+---------+-----+-----------+
| pg-2   | 10.0.3.35:5434 | Leader       | running | 123 |           |
| + pg-1 | 10.0.3.85:5434 | Sync Standby | running | 123 |         0 |
| + pg-3 | 10.0.3.70:5434 | Sync Standby | running | 123 |         0 |
+--------+----------------+--------------+---------+-----+-----------+
```

1.6.2 Consulter la configuration du cluster

- `show-config` : affiche la configuration dynamique

La configuration commune des instances peut être affichée avec l'argument `show-config` :

```
$ patronictl show-config
```

```
loop_wait: 10
maximum_lag_on_failover: 1048576
postgresql:
  parameters:
    archive_command: pgbackrest --stanza=main archive-push %p
    archive_mode: 'on'
    checkpoint_timeout: 15min
    hot_standby: 'on'
    log_min_duration_statement: -1
    max_relication_slots: 5
    max_wal_senders: 5
  use_pg_rewind: true
  use_slots: true
  wal_keep_segment: 2
  wal_level: replica
retry_timeout: 10
synchronous_mode: true
synchronous_node_count: 2
ttl: 30
```

1.6.3 Modifier la configuration du cluster

- `edit-config` : édite la configuration dynamique de l'agrégat
  - ni fichier de conf, ni `ALTER SYSTEM` !
- Si redémarrage : à demander explicitement

Cette commande lance l'éditeur par défaut de l'environnement afin d'effectuer une modification dans la configuration de l'agrégat. Elle nécessite la commande `less` pour la vérification finale.

Utiliser `ALTER SYSTEM` ou modifier manuellement les fichiers du `PGDATA` peut mener à des nœuds utilisant des configurations différentes !

Il ne faut pas modifier `postgresql.conf` directement, mais utiliser `edit-config` pour adapter la configuration YAML dynamique, par exemple :

```
$ patronictl -c /etc/patroni/config.yml edit-config
```

```
loop_wait: 10
maximum_lag_on_failover: 1048576
postgresql:
  parameters:
    hot_standby: true
    max_replication_slots: 5
    max_wal_senders: 10
    shared_buffers: 64MB
    wal_level: replica
    work_mem: 70MB
  use_pg_rewind: true
  use_slots: true
retry_timeout: 10
ttl: 30
```

Après avoir enregistré ses modifications et quitter l'éditeur, la configuration est écrite dans le DCS puis appliquée de manière asynchrone par Patroni, sur chacun des nœuds concernés, si celle-ci ne nécessite pas de redémarrage. Dans le cas contraire, le nœud est marqué comme *pending restart* et doit être redémarré manuellement avec la commande `patronictl restart`.

En pratique, la configuration de PostgreSQL est stockée par Patroni dans le DCS qui fait référence :

```
# etcdctl get /service/acme/config
```

puis dans un fichier `postgresql.conf` sur les nœuds.

Le `pg_hba.conf` se modifie avec la même commande. Bien penser à reprendre toute sa configuration la première fois.

```
$ patronictl -c /etc/patroni/config.yml edit-config
+++
@@ -10,5 +10,7 @@
     work_mem: 50MB
   use_pg_rewind: true
   use_slots: true
+  pg_hba:
+  - host user erp 192.168.99.0/24 md5
+  - host all all all scram-sha-256
+  - host replication replicator all scram-sha-256
 retry_timeout: 10
 ttl: 30

Apply these changes? [y/N]: y
Configuration changed
```

1.6.4 Commandes de bascule

- **`switchover`** : promotion d'un secondaire
- **`failover`** : bascule par défaillance du primaire

La commande `switchover` permet d'effectuer une bascule du rôle primaire vers l'une des instances secondaires. Cette commande nécessite de spécifier explicitement l'instance secondaire devant être promue. L'instance primaire est alors déchue en secondaire, puis relâche le *leader lock*. Le verrou ayant disparu, une élection est organisée et seule l'instance Patroni désignée est autorisée à prendre possession du *leader lock*, puis promouvoir son instance PostgreSQL locale en production.

Voici un exemple d'exécution de cette commande :

```
$ patronictl switchover
Primary [pg-1]:
Candidate ['pg-2', 'pg-3'] []: pg-2
When should the switchover take place (e.g. 2021-05-28T14:48 )  [now]:
Current cluster topology
+ Cluster: acme (6876375338380834518) ---+---------+-----+-----------+
| Member | Host           | Role         | State   |  TL | Lag in MB |
+--------+----------------+--------------+---------+-----+-----------+
| pg-1   | 10.0.3.85:5434 | Leader       | running | 124 |           |
| pg-2   | 10.0.3.35:5434 | Sync Standby | running | 124 |         0 |
| pg-3   | 10.0.3.70:5434 | Sync Standby | running | 124 |         0 |
+--------+----------------+--------------+---------+-----+-----------+
Are you sure you want to switchover cluster acme, demoting current leader pg-1?
[y/N]: y
```

```
2021-05-28 13:48:45.22331 Successfully switched over to "pg-2"
+ Cluster: acme (6876375338380834518) --------+-----+-----------+
| Member | Host           | Role    | State   |  TL | Lag in MB |
+--------+----------------+---------+---------+-----+-----------+
| pg-1   | 10.0.3.85:5434 | Replica | stopped |     |   unknown |
| pg-2   | 10.0.3.35:5434 | Leader  | running | 124 |           |
| pg-3   | 10.0.3.70:5434 | Replica | running | 124 |        16 |
+--------+----------------+---------+---------+-----+-----------+
```

Il est également possible de forcer une bascule de manière non interactive avec l'argument option `--force` et en spécifiant le primaire courant et le nœud secondaire cible :

```
$ patronictl switchover --leader pg-2 --candidate pg-1 --force
Current cluster topology
+ Cluster: acme (6876375338380834518) ---+---------+-----+-----------+
| Member | Host           | Role         | State   |  TL | Lag in MB |
+--------+----------------+--------------+---------+-----+-----------+
| pg-1   | 10.0.3.85:5434 | Sync Standby | running | 125 |         0 |
| pg-2   | 10.0.3.35:5434 | Leader       | running | 125 |           |
| pg-3   | 10.0.3.70:5434 | Sync Standby | running | 125 |         0 |
+--------+----------------+--------------+---------+-----+-----------+
2021-05-28 14:03:28.68678 Successfully switched over to "pg-1"
+ Cluster: acme (6876375338380834518) --------+-----+-----------+
| Member | Host           | Role    | State   |  TL | Lag in MB |
+--------+----------------+---------+---------+-----+-----------+
| pg-1   | 10.0.3.85:5434 | Leader  | running | 125 |           |
| pg-2   | 10.0.3.35:5434 | Replica | stopped |     |   unknown |
| pg-3   | 10.0.3.70:5434 | Replica | running | 125 |        16 |
+--------+----------------+---------+---------+-----+-----------+
```

La commande `failover` permet de déclencher une bascule en déclarant défaillant le primaire courant. C'est une bonne manière de valider qu'un secondaire est prêt à devenir primaire et que les secondaires sont capables de se raccrocher à lui une fois sa promotion effectuée.

Contrairement à la commande `switchover`, la commande `failover` ne nécessite pas de désigner l'instance à promouvoir. L'élection se déroule normalement et une des meilleures instances secondaires disponible est alors promue.

La commande `failover` permet aussi de promouvoir une instance lorsque toutes les instances disponibles sont au statut `replica`. Ce genre de cas peut se présenter lors de certaines opérations de restauration.

Voici un exemple d'utilisation :

```
$ patronictl failover
Candidate ['pg-1', 'pg-3'] []: pg-1
Current cluster topology
+ Cluster: acme (6876375338380834518) ---+---------+-----+-----------+
| Member | Host           | Role         | State   |  TL | Lag in MB |
+--------+----------------+--------------+---------+-----+-----------+
| pg-1   | 10.0.3.85:5434 | Sync Standby | running | 123 |         0 |
| pg-2   | 10.0.3.35:5434 | Leader       | running | 123 |           |
| pg-3   | 10.0.3.70:5434 | Sync Standby | running | 123 |         0 |
+--------+----------------+--------------+---------+-----+-----------+
Are you sure you want to failover cluster acme, demoting current leader pg-2?
```

```
[y/N]: y

2021-05-28 13:46:47.41163 Successfully failed over to "pg-1"
+ Cluster: acme (6876375338380834518) --------+-----+-----------+
| Member | Host           | Role    | State   |  TL | Lag in MB |
+--------+----------------+---------+---------+-----+-----------+
| pg-1   | 10.0.3.85:5434 | Leader  | running | 123 |           |
| pg-2   | 10.0.3.35:5434 | Replica | stopped |     |   unknown |
| pg-3   | 10.0.3.70:5434 | Replica | running | 123 |        16 |
+--------+----------------+---------+---------+-----+-----------+
```

Au passage, on remarque qu'après un *switchover* ou un *failover*, des secondaires configurés pour être synchrones, ne le sont plus pendant un court moment.

1.6.5 Contrôle de la bascule automatique

- `pause`, `resume`
- `history`
- `reinit`

Maintenance :

La commande `patronictl pause` place le cluster en mode maintenance. Cette commande « détache » le démon Patroni de l'instance qu'il manage. Cela a plusieurs effets sur le comportement du système :

- le mécanisme de promotion automatique est désactivé ;
- le redémarrage du service Patroni ne provoquera plus le redémarrage de l'instance associée ;
- il n'est pas possible d'effectuer une bascule sans préciser de cible.

On l'utilise généralement lorsqu'une anomalie conduit à une avalanche de bascules non désirées ou lorsque l'on doit exécuter des opérations de maintenance sur le nœud qui entreraient en conflit avec Patroni.

L'option `--wait` permet de s'assurer que la commande a été prise en compte par tous les nœuds. On peut observer son effet avec la commande `patronictl list` qui affiche que le mode maintenance est activé :

```
$ patronictl list
+--------+------------+----------------+---------+----+-----------+
| Member | Host       | Role           | State   | TL | Lag in MB |
[…]
```

```
+--------+------------+---------------+---------+----+-----------+
 Maintenance mode: on
```

La commande `patronictl resume` permet de désactiver le mode maintenance. L'option `--wait` permet de s'assurer que la commande a été prise en compte par tous les nœuds.

Historique :

L'historique des bascules est disponible via la commande `patronictl history` :

```
$ patronictl history
+-----+------------+------------------------------+----------------------------------+
|  TL |        LSN | Reason                       | Timestamp                        |
+-----+------------+------------------------------+----------------------------------+
|   1 |   25577936 | no recovery target specified |                                  |
|   2 |   83886528 | no recovery target specified |                                  |
|   3 |   83887160 | no recovery target specified |                                  |
[…]
| 122 | 4445962400 | no recovery target specified |2021-05-28T13:41:57.231514+00:00|
| 123 | 4462739616 | no recovery target specified |2021-05-28T13:46:47.366787+00:00|
| 124 | 4479516832 | no recovery target specified |2021-05-28T13:48:44.616172+00:00|
+-----+------------+------------------------------+----------------------------------+
```

Réinitialisation :

La commande `patronictl reinit` réinitialise un nœud entièrement.

Toutes les données du nœud sont détruites, écrasées par celles du primaire !

```
$ patronictl reinit acme pg-2
+ Cluster: acme (6876375338380834518) ----+----+-----------+
| Member | Host       | Role    | State   | TL | Lag in MB |
+--------+------------+---------+---------+----+-----------+
| pg-1   | 10.0.3.201 | Leader  | running |  8 |           |
| pg-2   | 10.0.3.202 | Replica | running |  8 |         0 |
+--------+------------+---------+---------+----+-----------+
Are you sure you want to reinitialize members pg-2? [y/N]: y
Success: reinitialize for member pg-2
```

Il est impossible de lancer une réinitialisation du nœud primaire puisqu'elle est faite à partir du primaire courant.

```
$ patronictl reinit acme pg-1
+ Cluster: acme (6876375338380834518) ----+----+-----------+
| Member | Host       | Role    | State   | TL | Lag in MB |
+--------+------------+---------+---------+----+-----------+
| pg-1   | 10.0.3.201 | Leader  | running |  8 |           |
| pg-2   | 10.0.3.202 | Replica | running |  8 |         0 |
+--------+------------+---------+---------+----+-----------+
Which member do you want to reinitialize [pg-1, pg-2]? []: pg-1
Are you sure you want to reinitialize members pg-1? [y/N]: y
Failed: reinitialize for member pg-1, status code=503, (I am the leader,
can not reinitialize)
```

1.6.6 Changements de configuration

- `reload`
  - attention aux *pending restart*
- `restart`

Changement de configuration :

La commande `patronictl reload` recharge la configuration de tous les nœuds de l'agrégat ou d'un nœud s'il est spécifié.

```
$ patronictl reload acme pg-1
+ Cluster: acme (6876375338380834518) ---+---------+-----+-----------+
| Member | Host           | Role         | State   |  TL | Lag in MB |
+--------+----------------+--------------+---------+-----+-----------+
| pg-1   | 10.0.3.85:5434 | Leader       | running | 122 |           |
| pg-2   | 10.0.3.35:5434 | Sync Standby | running | 122 |         0 |
| pg-3   | 10.0.3.70:5434 | Sync Standby | running | 122 |         0 |
+--------+----------------+--------------+---------+-----+-----------+
Are you sure you want to reload members pg-1? [y/N]: y
Reload request received for member pg-1 and will be processed within 10 seconds
```

Les traces montrent l'opération :

```
pg-2 patroni@acme[21335]: 2021-08-04 15:51:40,473
                                  INFO: Reloading PostgreSQL configuration.
pg-2 patroni@acme[21335]: envoi d'un signal au serveur
pg-2 patroni@acme[21335]: 2021-08-04 15:51:41,762
                                  INFO: Lock owner: pg-1; I am pg-2
```

Si les modifications nécessitent un redémarrage de PostgreSQL, les serveurs dont la configuration a été modifiée sont marqués avec la mention `pending restart` dans `patronictl list`. Il est nécessaire de les redémarrer pour que le changement de configuration soit bien pris en compte.

Redémarrage :

La commande `patronictl restart` redémarre le nœud spécifié ou l'agrégat entier en commençant par le primaire, suivi de ses secondaires.

Elle ne provoque pas de changement de rôle si l'opération se passe bien. Il est possible de redémarrer uniquement les serveurs en mode *pending restart* grâce à l'option `--pending`.

1.6.7 *endpoints* de l'API REST

L'API REST permet de :

- Contrôler du rôle du serveur
- S'informer sur l'état d'un nœud ou du cluster
- Manipuler le cluster

L'API REST de Patroni est principalement utilisée par `patronictl` mais peut tout aussi bien être consultée par n'importe quel autre outil, tel que `curl`, `wget` ou encore un *load balancer*.

Cette API permet par exemple de confirmer le rôle d'un serveur grâce à une simple requête HTTP sur l'un des *endpoints* suivants :

- `/primary`, `/leader` et `/standby-leader`
- `/replica`
- `/asynchronous`
- `/synchronous`
- `/read-only` et `/read-only-sync`

Il est possible d'enrichir la requête API en filtrant sur le lag de réplication. On peut aussi vérifier la présence ou la valeur d'un tag dans la configuration d'un nœud. Cette option n'est néanmoins pas disponibles pour les *endpoints* : `/primary`, `/leader` et `/standby-leader`.

Exemples d'utilisations :

```
$ curl -I -s http://10.20.61.103:8008/replica?tag_is_candidate=true
HTTP/1.0 200 OK

$ curl -I -s http://10.20.61.103:8008/replica?tag_is_here=true
HTTP/1.0 503 Service Unavailable

$ curl -I -s http://10.20.61.103:8008/replica?tag_doesnt_exist=false
HTTP/1.0 503 Service Unavailable
```

```
$ curl -I -s http://10.20.61.103:8008/replica?lag=16MB
HTTP/1.0 200 OK
```

L'API permet aussi d'accéder à d'autres informations :

- état d'un nœud : `/patroni`, `/` ;
- état du cluster : `/cluster` ;
- disponibilité du service ;
  - `/health` : code HTTP 200 si PostgreSQL fonctionne ;
  - `/liveness` : code HTTP 200 si Patroni fonctionne et que ça boucle de contrôle de la haute disponibilité fonctionne correctement ;
  - `/readiness` : code HTTP 200 si Patroni fonctionne en tant que leader ;
- historique des changements de timeline : `/history` ;
- configuration dynamique du cluster : `/config`.

Comme démontré précédemment, la configuration peut également être modifiée avec une requête `PATCH` ou remplacée avec une requête `PUT`.

Pour finir, il est possible d'interagir avec le cluster pour réaliser certaines actions de maintenances :

- `/switchover`, `/failover`
- `/restart`, `reload`, `/reinitialize`

Par exemple :

```
$ curl -s http://10.20.61.103:8008/switchover -XPOST \
>      -d'{"leader":"p2", "candidate":"p1"}'

Successfully switched over to "p1"
```

Les actions `switchover` et `restart` peuvent être planifiées :

```
$ curl -s http://10.20.61.103:8008/switchover -XPOST -d \
    '{"leader":"p1", "candidate":"p3", "scheduled_at":"2023-03-08T11:30+00"}'
Switchover scheduled

$ curl -s http://10.20.61.103:8009/restart -XPOST -d \
    '{"schedule":"2023-03-08T11:45+00"}'
Restart scheduled
```

Seule une opération de chaque type peut être planifiée. Elles peuvent être dé-planifiées avec une requête `DELETE`.

```
$ curl -s http://10.20.61.103:8008/switchover -XDELETE
scheduled switchover deleted

$ curl -s http://10.20.61.103:8009/restart -XDELETE
scheduled restart deleted
```

1.7 PROXY, VIP ET POOLERS DE CONNEXIONS

- Connexions aux réplicas
- Chaîne de connexion
- HAProxy
- Keepalived

1.7.1 Connexions aux réplicas

- Réplication asynchrone, synchrone et *remote apply*.
- API REST de Patroni

Les réplicas d'un cluster Patroni peuvent être utilisés pour faire de la répartition de charge en lecture (*query off-loading*). Il faut cependant être conscient des limitations associées à cette utilisation des instances secondaires.

Le mode de réplication est contrôlé par les paramètres `synchronous_commit` et `synchronous_standby_names`.

Si la réplication est asynchrone, l'instance secondaire n'est pas nécessairement à jour. Cependant, la mise en place d'une réplication synchrone ne règle pas totalement ce problème. En effet, la garantie de réplication synchrone porte sur l'écriture des données dans les WAL de l'instance secondaire et la demande de synchronisation sur disque, ce qui ne garanti pas l'application et la visibilité de ces données. Il est possible de configurer la réplication en *remote apply*, ce qui garanti que les données sont visibles sur la secondaire avant que la primaire ne rende la main au client. Dans cette situation cependant, les données sont potentiellement visibles sur la secondaire avant la primaire.

Un autre point à prendre en compte est la cohérence des données entre les instances secondaires. Sur ce plan, il n'y a aucune garantie.

L'API REST de Patroni permet de choisir à quel serveur on se connecte. Pour cela, il faut utiliser les *endpoints* `/replica`, `/read-only`, `/asynchronous` ou `/synchronous`. Une requête HTTP renverra le code 200, si le serveur correspond au filtre du *endpoint*.

On peut également éliminer des serveurs en fonction de la valeur d'un tag sur la base du retard de réplication pour les *endpoints* : `/replica`, `/read-only`, `/asynchronous`.

1.7.2 Chaîne de connexion

- Chaînes de connexion multi-hôtes
- 2 formats différents : URL ou `clé=valeur`
- Sélection du type de nœuds grâce à `target_session_attrs`

La chaîne de connexion utilisée pour se connecter à une instance PostgreSQL permet d'indiquer plusieurs nœuds et sur quel type de nœud se connecter, sans avoir besoin de proxy ou d'appel d'API.

Cette chaîne de connexion existe sous deux formats dont voici des exemples :

```
host=p1,p2,p3 user=dba dbname=postgres target_session_attrs=primary
postgresql://dba@p1,p2,p3/postgres?target_session_attrs=primary
```

Pour plus de détail à propos de ces chaînes de connexions, voir : https://www.postgresql.org/docs/current/libpq-connect.html#LIBPQ-CONNSTRING

Le paramètre `target_session_attrs` permet de définir quel type d'instance rechercher parmi les valeurs : `any` (par défaut), `read-write`, `read-only`, `primary`, `standby` et `prefer-standby`.

Les deux premiers existent depuis la version 10 de PostgreSQL, tous les autres ont été ajoutés à partir de la version 14. Pour plus de détails, voir : https://www.postgresql.org/docs/current/libpq-connect.html#LIBPQ-CONNECT-TARGET-SESSION-ATTRS

Pour se connecter à une instance en lecture seule avec `psql`, nous pourrons utiliser :

```
psql "postgresql://dba@p1,p2,p3:5432/postgres?target_session_attrs=prefer-standby"
```

1.7.3 HAProxy

- Répartiteur de charge (*round-robin*)
- Check HTTP sur l'API REST de Patroni
  - `\primary`
  - `\replica`
- Page Web pour les statistiques
- Attention à sa disponibilité !

HAProxy est un répartiteur de charge consommant généralement peu de ressources.

Il peut être installé sur un serveur indépendant, mais il est alors nécessaire de penser à sa mise en haute disponibilité pour éviter de créer un *SPOF* dans l'architecture.

Une autre solution consiste à placer HAProxy sur le serveur d'application ou de base de données et coupler la haute disponibilité du répartiteur de charge avec celle de l'application qu'on lui associe.

La documentation de Patroni propose d'utiliser la configuration suivante pour HAProxy :

```
global
    maxconn 100

defaults
    log     global
    mode    tcp
    retries 2
    timeout client 30m
    timeout connect 4s
    timeout server 30m
    timeout check 5s

listen stats
    mode http
    bind *:7000
    stats enable
    stats uri /

listen primary
    bind *:5000
    option httpchk HEAD /primary
    http-check expect status 200
    default-server inter 3s fall 3 rise 2 on-marked-down shutdown-sessions
    server node1 10.20.61.103:5432 maxconn 100 check port 8008
    server node2 10.20.61.104:5432 maxconn 100 check port 8008
    server node3 10.20.61.105:5432 maxconn 100 check port 8008

listen replicas
    bind *:5001
    option httpchk HEAD /replica
    http-check expect status 200
    default-server inter 3s fall 3 rise 2 on-marked-down shutdown-sessions
    server node1 10.20.61.103:5432 maxconn 100 check port 8008
    server node2 10.20.61.104:5432 maxconn 100 check port 8008
    server node3 10.20.61.105:5432 maxconn 100 check port 8008
```

Elle permet de mettre en place :

- un *endpoint* pour la consultation des statistiques sur le port 7000 ;
- un *endpoint* pour la connexion à l'instance primaire sur le port 5000 ;
- un *endpoint* pour la connexion aux instances secondaires sur le port 5001.

La section `global` défini le nombre de connexion maximal à 100.

La section `default` permet de définir où tracer (paramètre `log`), le type de connexion ici `tcp` (paramètre `mode`), ainsi que les timeouts pour les connexions à HAProxy et à PostgreSQL (paramètres `timeout*`).

La section `listen stats` permet de définir un *endpoint* pour la consultation des statistiques d'utilisation de HAProxy sur le port 7000. Elles sont consultables en se connectant avec un navigateur web (paramètre `mode`).

La section `listen primary` permet de renvoyer les connexions sur l'instance primaire lorsque l'on se connecte au port 5000 de HAProxy (paramètre `bind`). Une vérification est réalisée sur le *endpoint* `/primary` de l'API REST de Patroni pour établir quelle instance est la primaire (paramètres `http-check` et `option httpchl`). Pour ce faire, HAProxy teste pour chaque serveur déclaré (paramètres `server`) si l'API REST qui écoute sur le port 8008 (paramètre `check port` du serveur) répond par un code retour HTTP 200.

La section `listen replicas` permet de renvoyer les connexions sur les instances secondaires lorsque l'on se connecte au port 5001 de HAProxy. Une vérification est réalisée sur le *endpoint* `/replica` de l'API REST de Patroni pour vérifier quelles instances sont des secondaires. Pour ce faire, HAProxy teste pour chaque serveur déclaré, si l'API REST qui écoute sur le port 8008 répond par un code retour HTTP 200.

Pour chacune des sections permettant de se connecter à PostgreSQL, la connexion est vérifiée toutes les 3 secondes (paramètre `inter`). Au bout de trois échecs consécutifs, le serveur est considéré comme hors service (paramètre `fall`) et les sessions en cours sont stoppées (paramètre `on-marked-down shutdown-sessions`). Si un serveur est marqué comme indisponible, il faut 2 tests réussis consécutivement avant que le serveur soit considéré comme disponible (paramètre `rise`).

Il faut faire attention à l'utilisation du paramètre `lag=valeur` sur le *endpoint* de l'API REST de Patroni. Une valeur trop faible peut entraîner des changements de statut fréquents et donc des déconnexions fréquentes.

Par défaut, la répartition de charge se fait avec un algorithme de *round-robin*. On peut changer l'algorithme en ajoutant un paramètre `balance` dans la définition d'un *endpoint* et/ou spécifier des poids par serveur pour influencer l'algorithme (paramètre `weight` d'un serveur). Il y a beaucoup de possibilités d'algorithme, par exemple :

`static-rr` : Comme *round-robin* mais sans prendre en compte les poids.

`leastconn` : Serveur avec le moins de connexions.

`first` : Premier serveur (trié par identifiant) avec une connexion disponible. Un identifiant est automatiquement attribué à tout serveur n'en possédant pas.

Le nom fourni pour la déclaration des serveurs sera visible dans la page de statistiques, ici `node1`, `node2`, `node3`.

1.7.4 Keepalived

- Monte la VIP sur le serveur de l'instance primaire
- Utilisation de l'API REST de Patroni

Une VIP ou *Virtual IP address* est une adresse IP qui peut être partagée par plusieurs serveurs. Elle n'est active que sur un serveur à la fois. Cela permet d'avoir un point d'accès unique qui change en fonction de la disponibilité d'un service sur plusieurs serveurs.

Keepalived permet de gérer une VIP et de s'assurer qu'elle ne soit montée que sur un seul serveur. Il permet d'utiliser des scripts de vérification, de surveiller l'état d'un processus, la disponibilité d'un serveur ou la présence d'un fichier de déclenchement afin de conditionner le montage de la VIP.

Keepalived est un outil très versatile qui permet aussi de faire la répartition de charge.

Voici un exemple de configuration pour maintenir une VIP sur le serveur de l'instance primaire d'un cluster Patroni. Elle doit être mise en place sur tous les serveurs PostgreSQL du cluster.

```
global_defs {
    enable_script_security
    script_user root
}

vrrp_script keepalived_check_patroni {
    script "/usr/local/bin/keepalived_check_patroni.sh"
    interval 3          # interval between checks
    timeout 5           # how long to wait for the script return
    rise 1              # How many time the script must return ok, for the
                        # host to be considered healthy (avoid flapping)
    fall 1              # How many time the script must return Ko; for the
                        # host to be considered unhealthy (avoid flapping)
}

vrrp_instance VI_1 {
    state MASTER
    interface eth1
    virtual_router_id 51
    priority 244
    advert_int 1
    virtual_ipaddress {
        10.20.30.50/24
    }
    track_script {
        keepalived_check_patroni
    }
}
```

La section `vrrp_instance` permet de gérer la configuration de la VIP (spécifiée dans le paramètre `virtual_ipaddress`). Ici l'état initial de cette instance sera `MASTER` (paramètre `state`).

La VIP sera montée sur l'interface `eth1` (paramètre `interface`). Il est important de choisir un `virtual_router_id` inutilisé pour la configuration de la VIP. Il est possible de mettre un poids sur les serveurs (paramètre `priority`), par convention un serveur primaire devrait avoir la priorité 255. Le cas présent est un peu différent, puisqu'on utilise un script pour déterminer l'état de l'instance (paramètre `track_script`).

La section `keepalived_check_patroni` permet de gérer le script chargé de vérifier l'état de l'instance dans Patroni. Le script utilisé doit être capable d'interroger l'API REST de Patroni pour connaître l'état d'une instance en particulier (paramètre `script`). On peut définir la fréquence de passage du script (paramètre `interval`), ainsi qu'un timeout pour le script (paramètre `timeout`). Il est possible de configurer le nombre d'échecs successifs du script avant qu'une ressource soit considérée comme indisponible (paramètre `fall`). De même, on peut spécifier le nombre succès du script pour que la ressource soit considérée comme à nouveau disponible.

Voici un exemple de script de vérification :

```
#!/bin/bash

/usr/bin/curl \
  -X GET -I --fail \
  # --cacert ca.pem  --cert p1.pem  --key p1-key.pem \
  https://127.0.0.1:8008/primary &>>/var/log/patroni/keepalived_vip.log
```

Dans le cadre de ce script, pensez à prévoir une configuration *logrotate* sur le fichier de log obtenu.

1.8 QUESTIONS

– C'est le moment !

1.9 QUIZ

https://dali.bo/r58_quiz

1.10 TRAVAUX PRATIQUES

1.10.1 Patroni : installation

Sur les machines créées précédemment :

- installer PostgreSQL sur les **2 nœuds** depuis les dépôts PGDG
- installer Patroni sur les **2 nœuds** depuis les dépôts PGDG

Avec Debian, ne pas utiliser l'intégration de Patroni dans la structure de gestion multi-instance proposée par `postgresql-common` afin de se concentrer sur l'apprentissage.

- configurer et démarrer le cluster Patroni `acme` sur les **2 nœuds**
- observer les traces de chaque nœud Patroni
- observer la topologie de l'agrégat avec `patronictl`.

- Ajouter le 3ème nœud à l'agrégat.

- Déterminer le primaire via l'API Patroni en direct avec `curl`.

- Quels sont les slots de réplication sur le *leader* ?

- Forcer l'utilisation du watchdog dans la configuration de Patroni. Que se passe-t-il ?

- Donner les droits à l'utilisateur `postgres` sur le fichier `/dev/watchdog`. Après quelques secondes, que se passe-t-il ?

1.10.2 Patroni : utilisation

- Se connecter depuis l'extérieur à la base **postgres**, d'un des nœuds.

- Comment obtenir une connexion en lecture et écrire ?

- Créer une table :

```
CREATE TABLE insertions (
  id     int         GENERATED ALWAYS AS IDENTITY,
  d      timestamptz DEFAULT now(),
  source text        DEFAULT inet_server_addr()
);
```

- Insérer une ligne toutes les secondes, à chaque fois dans une nouvelle connexion au primaire.

- Dans une autre fenêtre, afficher les 20 dernières lignes de cette table.

- Stopper le nœud *leader* Patroni.
- Que se passe-t-il dans la topologie, et dans les requêtes ci-dessus ?

- Arrêter les processus du nouveau primaire. Il ne reste qu'un nœud actif. Que se passe-t-il ?

- Arrêter deux nœuds du cluster etcd. Que se passe-t-il ?

- Redémarrer les nœuds etcd.

- Relancer un des nœuds Patroni.

- Sur le troisième nœud (arrêté), détruire le `PGDATA`. Relancer Patroni.

- Forcer un *failover* vers le nœud `p1`.

- Modifier les paramètres `shared_buffers` et `work_mem`. Si besoin, redémarrer les nœuds.

1.11 TRAVAUX PRATIQUES DEBIAN 12 (SOLUTIONS)

1.11.1 Patroni : installation

> Sur les machines créées précédemment :
>
> – installer PostgreSQL sur les **2 nœuds** depuis les dépôts PGDG

Activation du dépôt PGDG, à exécuter sur nœuds `p1` et `p2` :

```
# apt update
[…]

# apt install postgresql-common
[…]

# /usr/share/postgresql-common/pgdg/apt.postgresql.org.sh -y
This script will enable the PostgreSQL APT repository on apt.postgresql.org on
your system. The distribution codename used will be bookworm-pgdg.

Using keyring /usr/share/postgresql-common/pgdg/apt.postgresql.org.gpg
Writing /etc/apt/sources.list.d/pgdg.sources ...

Running apt-get update […]
Reading package lists... Done

You can now start installing packages from apt.postgresql.org.

Have a look at https://wiki.postgresql.org/wiki/Apt for more information;
most notably the FAQ at https://wiki.postgresql.org/wiki/Apt/FAQ
```

Désactiver la création automatique d'une instance durant l'installation :

```
mkdir -p /etc/postgresql-common/createcluster.d
echo create_main_cluster = false >
 ↪  /etc/postgresql-common/createcluster.d/custom.conf
```

Installation de PostgreSQL 16 :

```
# apt install postgresql-16
[…]
Setting up postgresql-16 (16.2-1.pgdg120+2) ...
```

> – installer Patroni sur les **2 nœuds** depuis les dépôts PGDG

Les dépôts PGDG fournissent le paquet `patroni` :

```
# apt install patroni
```

> – configurer et démarrer le cluster Patroni `acme` sur les **2 nœuds**

Avec Debian, ne pas utiliser l'intégration de Patroni dans la structure de gestion multi-instance proposée par `postgresql-common` afin de se concentrer sur l'apprentissage.

La configuration de Patroni se déroule dans le fichier `/etc/patroni/config.yml`. Dans le doute, ce fichier est indiqué par la commande `systemctl status patroni`.

Sur les nœuds `p1` et `p2`, commencer par générer un fichier de configuration comme base de travail :

```
PATH="$PATH:/usr/lib/postgresql/16/bin" patroni \
    --generate-sample-config /etc/patroni/config.yml
```

Dans ce fichier, éditer les variables suivantes :

- `scope: "acme"` : le scope, ou nom, du cluster. Ce nom est utilisé au sein du DCS comme préfixe de toutes les clés ;
- `name: "<hostname>"` : nom de la machine. Cette valeur doit être différente pour chaque nœud ;
- `log.level: INFO` : dans le cadre de ce TP, il est aussi possible de positionner le niveau de log à `DEBUG` ;
- `log.dir: '/var/log/patroni/acme'` : pour les besoins du TP, afin de bien séparer les journaux de Patroni et PostgeSQL, nous demandons à Patroni d'écrire lui-même ses journaux dans ce répertoire ;
- `restapi` : vérifier que l'adresse IP d'écoute est correcte pour chaque serveur ;
- `bootstrap.dcs.postgresql.use_pg_rewind: false` : il est préférable de désactiver par défaut l'utilisation de `pg_rewind`. Cette fonctionnalité ne doit être activée que sur certains environnements.
- `postgresql.datadir: "/var/lib/postgresql/16/main"` : emplacement du `PGDATA` ;
- `postgresql.pg_hba:` : vérifier la cohérence des règles pour les clients et la réplication. Éventuellement, facilitez-vous la vie pour le reste du TP ;
- `postgresql.bindir: "/usr/lib/postgresql/16/bin"` : chemin vers les binaires de PostgreSQL ;
- `postgresql.authentication.replication.password` : positionner le mot de passe à attribuer au rôle `replicator` ;
- `postgresql.authentication.superuser.password` : positionner le mot de passe à attribuer au rôle `postgres` ;
- `postgresql.listen` et `postgresql.connect_address` : vérifier que les adresses IP sont correctes ;

Une fois ce modèle complété, la section dédiée au DCS doit encore être ajoutée. Collecter les adresses IP des nœuds etcd et ajouter à la configuration la section `etcd3` sur le modèle suivant :

```
etcd3:
  hosts:
  - 10.0.0.11:2379
  - 10.0.0.12:2379
  - 10.0.0.13:2379
```

Cette configuration est minimale. Libre à vous de modifier la façon dont les instances sont créées (activation des *checksums*, collation par défaut, etc), ajouter des règles, activer l'authentification etcd, …

Pour ce TP, comme nous plaçons les journaux de Patroni dans des fichiers, nous recommandons de faire de même pour PostgreSQL en ajoutant ces paramètres :

```
postgresql:
  parameters:
    logging_collector: on
    log_destination: stderr
```

Ce paramétrage a pour seul but de faciliter ce TP. Ce n'est pas une recommandation pour un serveur en production. Aucune gestion de rotation, rétention ou externalisation n'est ici en place. Il est tout aussi possible de se reposer sur `journald`.

Voici un exemple de configuration obtenue sur le nœud **p1** :

```
scope: 'acme'
name: p1

etcd3:
  hosts:
  - 10.0.0.11:2379
  - 10.0.0.12:2379
  - 10.0.0.13:2379

log:
  format: '%(asctime)s %(levelname)s: %(message)s'
  level: INFO
  max_queue_size: 1000
  traceback_level: ERROR
  dir: '/var/log/patroni/acme'

restapi:
  connect_address: 10.0.0.21:8008
  listen: 10.0.0.21:8008

# The bootstrap configuration. Works only when the cluster is not yet initialized.
# If the cluster is already initialized, all changes in the `bootstrap` section are
 ↪  ignored!
bootstrap:
  # This section will be written into <dcs>:/<namespace>/<scope>/config after
 ↪  initializing
  # new cluster and all other cluster members will use it as a `global configuration`.
  # WARNING! If you want to change any of the parameters that were set up
  # via `bootstrap.dcs` section, please use `patronictl edit-config`!
  dcs:
    loop_wait: 10
    retry_timeout: 10
    ttl: 30
```

```
    postgresql:
      parameters:
        hot_standby: 'on'
        max_connections: 100
        max_locks_per_transaction: 64
        max_prepared_transactions: 0
        max_replication_slots: 10
        max_wal_senders: 10
        max_worker_processes: 8
        track_commit_timestamp: 'off'
        wal_keep_size: 128MB
        wal_level: replica
        wal_log_hints: 'on'
      use_pg_rewind: false
      use_slots: true

postgresql:
  authentication:
    replication:
      password: 'pass'
      username: replicator
    superuser:
      password: 'pass'
      username: postgres
  bin_dir: '/usr/lib/postgresql/16/bin'
  connect_address: 10.0.0.21:5432
  data_dir: '/var/lib/postgresql/16/main'
  listen: 10.0.0.21:5432
  parameters:
    password_encryption: scram-sha-256
    logging_collector: on
    log_destination: stderr
  pg_hba:
  - local all all trust
  - host all all all trust
  - host replication replicator all scram-sha-256

tags:
  clonefrom: true
  failover_priority: 1
  noloadbalance: false
  nosync: false
```

Assurez-vous que ce fichier de configuration est bien accessible à l'utilisateur `postgres` et créer le répertoire nécessaire aux journaux applicatifs :

```
chmod 0644 /etc/patroni/config.yml
install -o postgres -g postgres -d /var/log/patroni/acme
```

Nous pouvons désormais valider la configuration :

```
patroni --validate /etc/patroni/config.yml
```

Le code retour de la commande est `0` si tout est valide. Sinon, la commande affiche les avertissements appropriés.

Il est maintenant possible de démarrer le service Patroni. Nous commençons d'abord sur `p1` pour créer l'instance :

```
# systemctl start patroni
```

Puis nous démarrons le service sur `p2`, qui va donc créer le secondaire :

```
# systemctl start patroni
```

> – observer les traces de chaque nœud Patroni

Sur p1, nous trouvons les messages suivants dans le journal `/var/log/patroni/acme/patroni.log` :

```
INFO: Selected new etcd server http://10.20.0.11:2379
INFO: No PostgreSQL configuration items changed, nothing to reload.
INFO: Lock owner: None; I am p1
INFO: trying to bootstrap a new cluster
INFO: postmaster pid=5541
INFO: establishing a new patroni heartbeat connection to postgres
INFO: running post_bootstrap
WARNING: Could not activate Linux watchdog device: Can't open watchdog device:
 ↪  [Errno 2] No such file or directory: '/dev/watchdog'
INFO: initialized a new cluster
INFO: no action. I am (p1), the leader with the lock
```

Le démon Patroni démarre, choisi un serveur etcd, se saisit du *leader lock* et initialise l'instance PostgreSQL locale. Les sorties standard et d'erreur des commandes exécutées par Patroni n'est pas capturée vers le journal de ce dernier. Ces commandes sont donc capturées par journald et associées au service `patroni`. Nous y retrouvons par exemple la sortie de `initdb` :

```
# journalctl -u patroni
[…]
patroni: The files belonging to this database system will be owned by user
 ↪  "postgres".
patroni: This user must also own the server process.
patroni: The database cluster will be initialized with locale "C.UTF-8".
[…]
patroni: Success. You can now start the database server using:
patroni:     /usr/lib/postgresql/16/bin/pg_ctl -D /var/lib/postgresql/16/main -l
 ↪  logfile start
```

Sur `p2`, nous trouvons dans le journal correspondant :

```
INFO: Selected new etcd server http://10.0.0.13:2379
INFO: No PostgreSQL configuration items changed, nothing to reload.
INFO: Lock owner: p1; I am p2
INFO: trying to bootstrap from leader 'p1'
INFO: replica has been created using basebackup
INFO: bootstrapped from leader 'p1'
INFO: postmaster pid=4551
INFO: Lock owner: p1; I am p2
INFO: establishing a new patroni heartbeat connection to postgres
INFO: no action. I am (p2), a secondary, and following a leader (p1)
```

Comme sur `p1`, le démon Patroni démarre et choisi un serveur etcd, mais il découvre le *leader lock* appartient déjà à `p1`. L'instance PostgreSQL locale n'existant pas, Patroni décide de la créer depuis celle de `p1`.

Dans les deux cas, la configuration par défaut des journaux applicatifs de PostgreSQL les places dans le répertoire `PGDATA/log`, donc ici `/var/lib/postgresql/16/main/log`, équivalent à `~postgres/16/main/log`.

- observer la topologie de l'agrégat avec `patronictl`.

L'utilisation de `patronictl` nécessite un fichier de configuration permettant de déterminer le nom du cluster et idéalement les nœuds du DCS. Sur les machines `p1` et `p2`, nous pouvons utiliser directement le fichier de configuration de Patroni `/etc/patroni/config.yml`. La commande devient :

```
$ patronictl -c /etc/patroni/config.yml topology
+ Cluster: acme (7349612307776631369) -----+----+-----------+-------+
| Member | Host       | Role    | State     | TL | Lag in MB | Tags  |
+--------+------------+---------+-----------+----+-----------+-------+
| p1     | 10.0.0.21 | Leader  | running   |  1 |           | […]   |
| + p2   | 10.0.0.22 | Replica | streaming |  1 |         0 | […]   |
+--------+------------+---------+-----------+----+-----------+-------+
```

Nous constatons que `p1` héberge bien l'instance primaire et `p2` la secondaire.

Pour simplifier cette commande, il est possible de positionner dans votre environnement le variable `PATRONICTL_CONFIG_FILE`. Par exemple :

```
$ export PATRONICTL_CONFIG_FILE=/etc/patroni/config.yml
$ patronictl topology
[…]
```

Pour la positionner automatiquement, vous pouvez par exemple créer le fichier `/etc/profile.d/99-patroni.sh` avec le contenu suivant :

```
cat <<EOF > /etc/profile.d/99-patroni.sh
PATRONICTL_CONFIG_FILE=/etc/patroni/config.yml
export PATRONICTL_CONFIG_FILE
EOF
chmod +x /etc/profile.d/99-patroni.sh
```

- Ajouter le 3ème nœud à l'agrégat.

Répéter les étapes précédentes sur le serveur `p3` :

- installation des dépôts PGDG ;
- installation de PostgreSQL et Patroni ;
- création du fichier de configuration de Patroni ;
- démarrage de l'instance sur `p3`.

Après l'ajout du troisième nœud, la topologie est la suivante :

```
$ patronictl topology
+ Cluster: acme (7349612307776631369) -----+----+-----------+------+
| Member | Host       | Role    | State     | TL | Lag in MB | Tags |
+--------+------------+---------+-----------+----+-----------+------+
| p1     | 10.0.0.21 | Leader  | running   |  1 |           | […]  |
| + p2   | 10.0.0.22 | Replica | streaming |  1 |         0 | […]  |
| + p3   | 10.0.0.23 | Replica | streaming |  1 |         0 | […]  |
+--------+------------+---------+-----------+----+-----------+------+
```

Il est conseillé lors des tests de garder une fenêtre répétant l'ordre régulièrement :

```
$ watch -n1 patronictl topology
```

- Déterminer le primaire via l'API Patroni en direct avec `curl`.

La configuration est visible de l'extérieur :

```
$ curl -s http://p1:8008/patroni | jq
```

```
{
  "state": "running",
  "postmaster_start_time": "[…]",
  "role": "master",
  "server_version": 160002,
  "xlog": {
    "location": 50534136
  },
  "timeline": 1,
  "replication": [
    {
      "usename": "replicator",
      "application_name": "p2",
      "client_addr": "10.0.0.22",
      "state": "streaming",
      "sync_state": "async",
      "sync_priority": 0
    },
    {
      "usename": "replicator",
      "application_name": "p3",
      "client_addr": "10.0.0.23",
      "state": "streaming",
      "sync_state": "async",
      "sync_priority": 0
    }
  ],
  "dcs_last_seen": 1711217105,
  "tags": {
    "clonefrom": true,
    "failover_priority": 1
  },
  "database_system_identifier": "7349612307776631369",
  "patroni": {
    "version": "3.2.2",
    "scope": "acme",
```

```
    "name": "p1"
  }
}
```

De manière plus précise :

```
$ curl -s http://p1:8008/cluster |\
jq '.members[] | select ((.role == "leader" ) and ( .state == "running")) | { name }'

{
  "name": "p1"
}
```

> – Quels sont les slots de réplication sur le *leader* ?

Par défaut, les slots de réplications portent le nom des nœuds réplicas.

```
$ curl -s http://p1:8008/cluster | jq '.members[] | select (.role == "replica") | {
↪  name }'

{
  "name": "p2"
}
{
  "name": "p3"
}
```

On peut le vérifier en se connectant à PostgreSQL sur le serveur primaire (ici `p1`) :

```
$ sudo -iu postgres
$ psql -c "SELECT slot_name FROM pg_replication_slots"
 slot_name
-----------
 p2
 p3
```

> – Forcer l'utilisation du watchdog dans la configuration de Patroni. Que se passe-t-il ?

Ajoutez la section watchdog dans le fichier de configuration de Patroni `/etc/patroni/config.yml` en positionnant `mode: required` :

```
watchdog:
  mode: required
  # device: /dev/watchdog
  safety_margin: 5
```

Il nous faut recharger la configuration de Patroni :

```
$ patronictl reload acme
+ Cluster: acme (7349612307776631369) --------+----+-----------+
| Member | Host       | Role    | State        | TL | Lag in MB |
+--------+------------+---------+--------------+----+-----------+
| p1     | 10.0.0.21 | Replica | start failed |    |   unknown |
```

```
| p2     | 10.0.0.22 | Replica | start failed |    |   unknown |
| p3     | 10.0.0.23 | Leader  | running      |  6 |           |
+--------+-----------+---------+--------------+----+-----------+
Are you sure you want to reload members p1, p2, p3? [y/N]: y
Reload request received for member p1 and will be processed within 10 seconds
Reload request received for member p2 and will be processed within 10 seconds
Reload request received for member p3 and will be processed within 10 seconds
```

Suite à cette modification, le cluster Patroni se retrouve sans primaire. Dans les journaux applicatif de `p3`, ici précédemment marqué *Leader*, nous trouvons :

```
INFO: Reloading PostgreSQL configuration.
INFO: Lock owner: p3; I am p3
ERROR: Configuration requires watchdog, but watchdog could not be configured.
INFO: Demoting self (immediate)
INFO: Leader key released
INFO: Demoting self because watchdog could not be activated
INFO: Lock owner: None; I am p3
INFO: not healthy enough for leader race
INFO: starting after demotion in progress
INFO: closed patroni connections to postgres
INFO: postmaster pid=561
INFO: establishing a new patroni heartbeat connection to postgres
WARNING: Watchdog device is not usable
INFO: Dropped unknown replication slot 'p1'
INFO: Dropped unknown replication slot 'p2'
INFO: following a different leader because i am not the healthiest node
```

> – Donner les droits à l'utilisateur `postgres` sur le fichier `/dev/watchdog`. Après quelques secondes, que se passe-t-il ?

Un simple `chown` sur le device `/dev/watchdog` de chaque VM peut suffire, au moins le temps de ce TP. Néanmoins, il ne survivrait pas au e de la machine.

Le fichier de service de Patroni propose de systématiquement modifier les droits sur ce device décommentant la ligne suivante :

```
#ExecStartPre=-/usr/bin/sudo /bin/chown postgres /dev/watchdog
```

Une autre solution est de configurer `udev` afin qu'il modifie les droits sur ce fichier automatiquement à chaque démarrage :

```
cat <<'EOF' > /etc/udev/rules.d/99-watchdog.rules
# give writes on watchdog device to postgres
SUBSYSTEM=="misc", KERNEL=="watchdog", ACTION=="add", RUN+="/bin/chown postgres
 ↪  /dev/watchdog"

# Or a better solution using ACL:
#SUBSYSTEM=="misc", KERNEL=="watchdog", ACTION=="add", RUN+="/bin/setfacl -m
 ↪  u:postgres:rw- /dev/watchdog"
EOF
```

Une fois les droits positionnés, l'un des nœuds Patroni devrait finalement réussir à activer le watchdog et ainsi devenir *leader* et promouvoir son instance PostgreSQL :

```
INFO: i6300ESB timer activated with 25 second timeout, timing slack 15 seconds
INFO: promoted self to leader by acquiring session lock
INFO: Lock owner: p2; I am p2
INFO: updated leader lock during promote
INFO: Lock owner: p2; I am p2
INFO: no action. I am (p2), the leader with the lock
```

1.11.2 Patroni : utilisation

- Se connecter depuis l'extérieur à la base **postgres**, d'un des nœuds.

La connexion extérieure peut se faire depuis la machine hôte des VM. Il est nécessaire d'y installer un client PostgreSQL, par exemple `postgresql-client`. Pour se connecter à l'instance `p1`, utiliser l'une des commandes suivantes :

```
$ psql -h p1 -p 5432 -U postgres -d postgres
$ psql -h 10.0.0.21 postgres postgres
```

En cas de problème, il faut regarder :

- les règles de *firewall* ;
- les traces du nœud PostgreSQL concerné, plus instructives que le simple message d'erreur du client ;
- le fichier de configuration `pg_hba.conf`.

- Comment obtenir une connexion en lecture et écrire ?

Il est nécessaire de se connecter à l'instance primaire pour réaliser des écritures en base. La chaîne de connexion permettant d'indiquer plusieurs nœuds, nous pouvons y préciser tous les nœuds du cluster. Afin de sélectionner le nœud primaire, il suffit d'ajouter le paramètre `target_session_attrs=primary` ou `target_session_attrs=read-write`.

Par exemple :

```
psql "host=p1,p2,p3 user=postgres target_session_attrs=primary"
psql "postgresql://postgres@p1,p2,p3/postgres?target_session_attrs=read-write"
```

Pour une connexion en lecture seule, nous utilisons par exemple :

```
psql "postgresql://postgres@p1,p2,p3:5432/postgres?target_session_attrs=prefer-
↪  standby"
```

- Créer une table :

```
CREATE TABLE insertions (
  id     int         GENERATED ALWAYS AS IDENTITY,
  d      timestamptz DEFAULT now(),
  source text        DEFAULT inet_server_addr()
);
```

- Insérer une ligne toutes les secondes, à chaque fois dans une nouvelle connexion au primaire.

> – Dans une autre fenêtre, afficher les 20 dernières lignes de cette table.

Dans la colonne `source` de la table créée, la valeur par défaut fait appel à la fonction `inet_server_addr()` qui retourne l'adresse IP du serveur PostgreSQL sur lequel nous sommes connectés.

Lancer une insertion toutes les secondes :

```
watch -n1 'psql -X -d "host=p1,p2,p3 user=postgres target_session_attrs=primary" \
               -c "INSERT INTO insertions SELECT;"'
```

Lire les vingt dernières lignes de la table :

```
watch -n1 'psql -X -d "host=p1,p2,p3 port=5432 user=postgres" \
               -c "SELECT * FROM insertions ORDER BY d DESC LIMIT 20"'
```

Bien entendu, nous obtenons toujours le même nœud dans la colonne `source`, ici `p1` :

```
 id  |               d                |    source
-----+--------------------------------+--------------
 …
 313 | 2024-03-05 15:40:03.118307+00 | 10.0.0.21/32
 312 | 2024-03-05 15:40:02.105116+00 | 10.0.0.21/32
 311 | 2024-03-05 15:40:01.090775+00 | 10.0.0.21/32
 310 | 2024-03-05 15:40:00.075847+00 | 10.0.0.21/32
 309 | 2024-03-05 15:39:59.061759+00 | 10.0.0.21/32
 308 | 2024-03-05 15:39:58.048074+00 | 10.0.0.21/32

(20 lignes)
```

> – Stopper le nœud *leader* Patroni.
> – Que se passe-t-il dans la topologie, et dans les requêtes ci-dessus ?

Sur **p1** :

```
# systemctl stop patroni
```

Après l'arrêt de `p1`, `p3` prend le rôle de *leader* :

```
$ patronictl -c /etc/patroni/config.yml topology
+ Cluster: acme (7349612307776631369) ---+----+-----------+------+
| Member | Host      | Role    | State   | TL | Lag in MB | Tags |
+--------+-----------+---------+---------+----+-----------+------+
| p3     | 10.0.0.23 | Leader  | running |  2 |           | […]  |
| + p1   | 10.0.0.21 | Replica | stopped |    |   unknown | […]  |
| + p2   | 10.0.0.22 | Replica | running |  1 |         0 | […]  |
+--------+-----------+---------+---------+----+-----------+------+
```

Les insertions échouent le temps de la bascule, ici pendant environ 2 secondes, puis continuent depuis l'autre nœud :

```
 id  |               d                |    source
-----+--------------------------------+--------------
 …
 445 | 2024-03-24 15:50:17.840394+00 | 10.0.0.23/32
```

```
 444 | 2024-03-24 15:50:16.823004+00 | 10.0.0.23/32
 431 | 2024-03-24 15:50:14.755045+00 | 10.0.0.21/32
 430 | 2024-03-24 15:50:13.740541+00 | 10.0.0.21/32
```

> – Arrêter les processus du nouveau primaire. Il ne reste qu'un nœud actif. Que se passe-t-il ?

Le *leader* Patroni a changé à nouveau :

```
$ patronictl -c /etc/patroni/config.yml topology
+ Cluster: acme (7349612307776631369) --+----+-----------+------+
| Member | Host      | Role   | State   | TL | Lag in MB | Tags |
+--------+-----------+--------+---------+----+-----------+------+
| p2     | 10.0.0.22 | Leader | running |  6 |           | […]  |
+--------+-----------+--------+---------+----+-----------+------+
```

> – Arrêter deux nœuds du cluster etcd. Que se passe-t-il ?

Sur `e1` et `e2` :

```
# systemctl stop etcd
```

Il n'y a plus de quorum etcd garantissant une référence. Le cluster Patroni se met en lecture seule et les insertions tombent en échec puisqu'elles exigent une connexion ouverte en écriture :

```
psql: error: connection to server at "p1" (10.0.0.21), port 5432 failed: Connection
 ↪  refused
        Is the server running on that host and accepting TCP/IP connections?
connection to server at "p2" (10.0.0.22), port 5432 failed: server is in hot standby
 ↪  mode
connection to server at "p3" (10.0.0.23), port 5432 failed: Connection refused
        Is the server running on that host and accepting TCP/IP connections?
```

> – Redémarrer les nœuds etcd.

Les écritures reprennent.

> – Relancer un des nœuds Patroni.

Dans le cadre de cette correction `p2` est l'actuel *leader*, nous redémarrons donc Patroni sur `p1` :

```
# systemctl restart patroni
```

L'instance sur `p1` se raccroche en secondaire :

```
$ patronictl -c /etc/patroni/config.yml topology
+ Cluster: acme (7349612307776631369) -----+----+-----------+------+
| Member | Host      | Role    | State     | TL | Lag in MB | Tags |
+--------+-----------+---------+-----------+----+-----------+------+
| p2     | 10.0.0.22 | Leader  | running   |  7 |           | […]  |
| + p1   | 10.0.0.21 | Replica | streaming |  7 |         0 | […]  |
+--------+-----------+---------+-----------+----+-----------+------+
```

– Sur le troisième nœud (arrêté), détruire le `PGDATA`. Relancer Patroni.

Le PGDATA de nos instances se trouve dans `/var/lib/postgresql/16/main`. Supprimons ce `PGDATA` sur `p3`, le nœud restant :

```
rm -rf /var/lib/postgresql/16/main
```

Relançons Patroni sur ce nœud :

```
# systemctl start patroni
```

Nous observons dans les journaux de Patroni et PostgreSQL que l'instance est recrée et se raccroche à `p2` :

```
LOG:  starting PostgreSQL 16.2 (Debian 16.2-1.pgdg120+2) […]
LOG:  listening on IPv4 address "10.0.0.23", port 5432
LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
LOG:  database system was interrupted while in recovery at log time […]
HINT:  If this has occurred more than once some data might be corrupted and you might
 ↪  need to choose an earlier recovery target.
LOG:  entering standby mode
LOG:  starting backup recovery with redo LSN 0/45405620, checkpoint LSN 0/4FD4D788,
 ↪  on timeline ID 7
LOG:  redo starts at 0/45405620
LOG:  completed backup recovery with redo LSN 0/45405620 and end LSN 0/509EA568
LOG:  consistent recovery state reached at 0/509EA568
LOG:  database system is ready to accept read-only connections
LOG:  started streaming WAL from primary at 0/50000000 on timeline 7
```

Le temps de la reconstruction de zéro, il est possible de voir l'évolution de l'état de `p3` de `creating replica` à `streaming` :

```
$ export PATRONICTL_CONFIG_FILE=/etc/patroni/config.yml
$ patronictl topology
+ Cluster: acme (7349612307776631369) ------------+----+-----------+------+
| Member | Host       | Role    | State            | TL | Lag in MB | Tags |
+--------+------------+---------+------------------+----+-----------+------+
| p2     | 10.0.0.22 | Leader  | running          |  7 |           | […]  |
| + p1   | 10.0.0.21 | Replica | streaming        |  7 |         0 | […]  |
| + p3   | 10.0.0.23 | Replica | creating replica |    |   unknown | […]  |
+--------+------------+---------+------------------+----+-----------+------+

/* Après un certain temps */

$ patronictl topology
+ Cluster: acme (7349612307776631369) -----+----+-----------+------+
| Member | Host       | Role    | State     | TL | Lag in MB | Tags |
+--------+------------+---------+-----------+----+-----------+------+
| p2     | 10.0.0.22 | Leader  | running   |  7 |           | […]  |
| + p1   | 10.0.0.21 | Replica | streaming |  7 |         0 | […]  |
| + p3   | 10.0.0.23 | Replica | streaming |  7 |         0 | […]  |
+--------+------------+---------+-----------+----+-----------+------+
```

- Forcer un *failover* vers le nœud `p1`.

```
$ patronictl failover
Current cluster topology
+ Cluster: acme (7349612307776631369) -----+----+-----------+------+
| Member | Host       | Role    | State     | TL | Lag in MB | Tags |
+--------+------------+---------+-----------+----+-----------+------+
| p1     | 10.0.0.21 | Replica | streaming |  7 |         0 | […]  |
+--------+------------+---------+-----------+----+-----------+------+
| p2     | 10.0.0.22 | Leader  | running   |  7 |           | […]  |
+--------+------------+---------+-----------+----+-----------+------+
| p3     | 10.0.0.23 | Replica | streaming |  7 |         0 | […]  |
+--------+------------+---------+-----------+----+-----------+------+
Candidate ['p1', 'p3'] []: p1
Are you sure you want to failover cluster acme, demoting current leader p2? [y/N]: y
[…] Successfully failed over to "p1"
+ Cluster: acme (7349612307776631369) ---+----+-----------+------+
| Member | Host       | Role    | State   | TL | Lag in MB | Tags |
+--------+------------+---------+---------+----+-----------+------+
| p1     | 10.0.0.21 | Leader  | running |  7 |           | […]  |
+--------+------------+---------+---------+----+-----------+------+
| p2     | 10.0.0.22 | Replica | stopped |    |   unknown | […]  |
+--------+------------+---------+---------+----+-----------+------+
| p3     | 10.0.0.23 | Replica | running |  7 |         0 | […]  |
+--------+------------+---------+---------+----+-----------+------+

$ patronictl topology
+ Cluster: acme (7349612307776631369) -----+----+-----------+------+
| Member | Host       | Role    | State     | TL | Lag in MB | Tags |
+--------+------------+---------+-----------+----+-----------+------+
| p1     | 10.0.0.21 | Leader  | running   |  8 |           | […]  |
| + p2   | 10.0.0.22 | Replica | streaming |  8 |         0 | […]  |
| + p3   | 10.0.0.23 | Replica | streaming |  8 |         0 | […]  |
+--------+------------+---------+-----------+----+-----------+------+
```

- Modifier les paramètres `shared_buffers` et `work_mem`. Si besoin, redémarrer les nœuds.

Pour modifier la configuration, nous devons utiliser `patronictl`, plutôt qu'éditer directement les fichiers de configuration. Une alternative est de modifier la configuration statique, dans le fichier YAML "/etc/patroni/config.yml". Cette méthode facilite leur maintenance, mais impose que le contenu soit identique sur tous les nœuds, ce qui est généralement le cas dans un déploiement industrialisé.

La commande `patronictl edit-config` appelle l'éditeur par défaut, souvent `vi`, `vim` ou `nano`. Vous pouvez modifier la variable d'environnement `EDITOR` pour pointer sur votre éditeur favori.

Éditons la configuration dynamique et ajoutons les deux paramètres :

```
$ export PATRONICTL_CONFIG_FILE=/etc/patroni/config.yml
$ patronictl edit-config
[…]

---
+++
```

```
@@ -12,6 +12,8 @@
     wal_keep_size: 128MB
     wal_level: replica
     wal_log_hints: 'on'
+    shared_buffers: 300MB
+    work_mem: 50MB
   use_pg_rewind: false
   use_slots: true
 retry_timeout: 10


Apply these changes? [y/N]: y
Configuration changed
```

Les nœuds sont à redémarrer à cause de la modification de `shared_buffers` :

```
$ patronictl topology
+ Cluster: acme (7349612307776631369) -----+----+------+-----------------+------+
| Member | Host      | Role    | State     | TL | Lag… | Pending restart | Tags |
+--------+-----------+---------+-----------+----+------+-----------------+------+
| p1     | 10.0.0.21 | Leader  | running   |  8 |      | *               | […]  |
| + p2   | 10.0.0.22 | Replica | streaming |  8 |    0 | *               | […]  |
| + p3   | 10.0.0.23 | Replica | streaming |  8 |    0 | *               | […]  |
+--------+-----------+---------+-----------+----+------+-----------------+------+
```

Commandons le redémarrage de PostgreSQL sur les trois nœuds :

```
$ patronictl restart acme
+ Cluster: acme (7349612307776631369) -----+----+------+-----------------+------+
| Member | Host      | Role    | State     | TL | Lag… | Pending restart | Tags |
+--------+-----------+---------+-----------+----+------+-----------------+------+
| p1     | 10.0.0.21 | Leader  | running   |  8 |      | *               | […]  |
+--------+-----------+---------+-----------+----+------+-----------------+------+
| p2     | 10.0.0.22 | Replica | streaming |  8 |    0 | *               | […]  |
+--------+-----------+---------+-----------+----+------+-----------------+------+
| p3     | 10.0.0.23 | Replica | streaming |  8 |    0 | *               | […]  |
+--------+-----------+---------+-----------+----+------+-----------------+------+
When should the restart take place (e.g. […])  [now]:
Are you sure you want to restart members p1, p2, p3? [y/N]: y
Restart if the PostgreSQL version is less than provided (e.g. 9.5.2)  []:
Success: restart on member p1
Success: restart on member p2
Success: restart on member p3
```

Vérifions que le paramétrage a bien été modifié :

```
$ for h in p1 p2 p3; do echo -ne $h:; psql -Xtd "host=$h user=postgres" -c "show
 ↪  shared_buffers"; done
p1: 300MB

p2: 300MB

p3: 300MB
```

Noter que le contenu des modifications est tracé dans un fichier `patroni.dynamic.json` dans le `PGDATA` :

```
$ jq . patroni.dynamic.json

{
  "loop_wait": 10,
  "postgresql": {
    "parameters": {
      "hot_standby": "on",
      "max_connections": 100,
      "max_locks_per_transaction": 64,
      "max_prepared_transactions": 0,
      "max_replication_slots": 10,
      "max_wal_senders": 10,
      "max_worker_processes": 8,
      "track_commit_timestamp": "off",
      "wal_keep_size": "128MB",
      "wal_level": "replica",
      "wal_log_hints": "on",
      "shared_buffers": "300MB",
      "work_mem": "50MB"
    },
    "use_pg_rewind": false,
    "use_slots": true
  },
  "retry_timeout": 10,
  "ttl": 30
}
```

1.12 TRAVAUX PRATIQUES ROCKY 9 (SOLUTIONS)

1.12.1 Patroni : installation

> Sur les machines créées précédemment :
>
> - installer PostgreSQL sur les **2 nœuds** depuis les dépôts PGDG

Activation du dépôt PGDG, à exécuter sur nœuds `p1` et `p2` :

```
# dnf install https://download.postgresql.org/pub/repos/yum/reporpms/EL-9-x86_64/\
pgdg-redhat-repo-latest.noarch.rpm

# dnf install -y postgresql16-server
[…]
Installed:
    postgresql16-16.2-1PGDG.rhel9.x86_64
    postgresql16-libs-16.2-1PGDG.rhel9.x86_64
    postgresql16-server-16.2-1PGDG.rhel9.x86_64

Complete!
```

> - installer Patroni sur les **2 nœuds** depuis les dépôts PGDG

Les dépôts PGDG fournissent le paquet `patroni` :

```
dnf install -y epel-release
dnf install -y patroni-etcd
```

> - configurer et démarrer le cluster Patroni `acme` sur les **2 nœuds**

La configuration de Patroni se déroule dans le fichier `/etc/patroni/patroni.yml`.

Sur les nœuds `p1` et `p2`, commencer par générer un fichier de configuration comme base de travail :

```
PATH="$PATH:/usr/pgsql-16/bin" patroni \
    --generate-sample-config /etc/patroni/patroni.yml
```

Dans ce fichier, éditer les variables suivantes :

- `scope: "acme"` : le scope, ou nom, du cluster. Ce nom est utilisé au sein du DCS comme préfixe de toutes les clés ;
- `name: "<hostname>"` : nom de la machine. Cette valeur doit être différente pour chaque nœud ;
- `log.level: INFO` : dans le cadre de ce TP, il est aussi possible de positionner le niveau de log à `DEBUG` ;

- `log.dir: '/var/log/patroni/acme'` : pour les besoins du TP, afin de bien séparer les journaux de Patroni et PostgeSQL, nous demandons à Patroni d'écrire lui-même ses journaux dans ce répertoire ;
- `restapi` : vérifier que l'adresse IP d'écoute est correcte pour chaque serveur ;
- `bootstrap.dcs.postgresql.use_pg_rewind: false` : il est préférable de désactiver par défaut l'utilisation de `pg_rewind`. Cette fonctionnalité ne doit être activée que sur certains environnements.
- `postgresql.datadir: "/var/lib/postgresql/16/main"` : emplacement du `PGDATA` ;
- `postgresql.pg_hba:` : vérifier la cohérence des règles pour les clients et la réplication. Éventuellement, facilitez-vous la vie pour le reste du TP ;
- `postgresql.bindir: "/usr/lib/postgresql/16/bin"` : chemin vers les binaires de PostgreSQL ;
- `postgresql.authentication.replication.password` : positionner le mot de passe à attribuer au rôle `replicator` ;
- `postgresql.authentication.superuser.password` : positionner le mot de passe à attribuer au rôle `postgres` ;
- `postgresql.listen` et `postgresql.connect_address` : vérifier que les adresses IP sont correctes ;

Une fois ce modèle complété, la section dédiée au DCS doit encore être ajoutée. Collecter les adresses IP des nœuds etcd et ajouter à la configuration la section `etcd3` sur le modèle suivant :

```
etcd3:
  hosts:
  - 10.0.0.11:2379
  - 10.0.0.12:2379
  - 10.0.0.13:2379
```

Cette configuration est minimale. Libre à vous de modifier la façon dont les instances sont créées (activation des *checksums*, collation par défaut, etc), ajouter des règles, activer l'authentification etcd, …

Pour ce TP, comme nous plaçons les journaux de Patroni dans des fichiers, nous recommandons de faire de même pour PostgreSQL en ajoutant ces paramètres :

```
postgresql:
  parameters:
    logging_collector: on
    log_destination: stderr
```

Ce paramétrage a pour seul but de faciliter ce TP. Ce n'est pas une recommandation pour un serveur en production. Aucune gestion de rotation, rétention ou externalisation n'est ici en place. Il est tout aussi possible de se reposer sur `journald`.

Voici un exemple de configuration obtenue sur le nœud **p1** :

```
scope: 'acme'
name: p1.hapat.vm

etcd3:
  hosts:
  - 10.0.0.11:2379
  - 10.0.0.12:2379
  - 10.0.0.13:2379

log:
  format: '%(asctime)s %(levelname)s: %(message)s'
  level: INFO
  max_queue_size: 1000
  traceback_level: ERROR
  dir: '/var/log/patroni/acme'

restapi:
  connect_address: 10.0.0.21:8008
  listen: 10.0.0.21:8008

# The bootstrap configuration. Works only when the cluster is not yet initialized.
# If the cluster is already initialized, all changes in the `bootstrap` section are
 ↪  ignored!
bootstrap:
  # This section will be written into <dcs>:/<namespace>/<scope>/config after
 ↪  initializing
  # new cluster and all other cluster members will use it as a `global configuration`.
  # WARNING! If you want to change any of the parameters that were set up
  # via `bootstrap.dcs` section, please use `patronictl edit-config`!
  dcs:
    loop_wait: 10
    retry_timeout: 10
    ttl: 30
    postgresql:
      parameters:
        hot_standby: 'on'
        max_connections: 100
        max_locks_per_transaction: 64
        max_prepared_transactions: 0
        max_replication_slots: 10
        max_wal_senders: 10
        max_worker_processes: 8
        track_commit_timestamp: 'off'
        wal_keep_size: 128MB
        wal_level: replica
        wal_log_hints: 'on'
      use_pg_rewind: false
      use_slots: true

postgresql:
  authentication:
    replication:
      password: 'pass'
      username: replicator
    superuser:
      password: 'pass'
```

```
      username: postgres
  bin_dir: '/usr/pgsql-16/bin'
  connect_address: 10.0.0.21:5432
  data_dir: '/var/lib/pgsql/16/data'
  listen: 10.0.0.21:5432
  parameters:
    password_encryption: scram-sha-256
    logging_collector: on
    log_destination: stderr
  pg_hba:
  - local all all trust
  - host all all all trust
  - host replication replicator all scram-sha-256

tags:
  clonefrom: true
  failover_priority: 1
  noloadbalance: false
  nosync: false
```

Assurez-vous que ce fichier de configuration est bien accessible à l'utilisateur `postgres` et créer le répertoire nécessaire aux journaux applicatifs :

```
chmod 0644 /etc/patroni/patroni.yml
install -o postgres -g postgres -d /var/log/patroni/acme
```

Nous pouvons désormais valider la configuration :

```
patroni --validate /etc/patroni/patroni.yml
```

Le code retour de la commande est `0` si tout est valide. Sinon, la commande affiche les avertissements appropriés.

Il est maintenant possible de démarrer le service Patroni. Nous commençons d'abord sur `p1` pour créer l'instance :

```
# systemctl start patroni
```

Puis nous démarrons le service sur `p2`, qui va donc créer le secondaire :

```
# systemctl start patroni
```

- observer les traces de chaque nœud Patroni

Sur p1, nous trouvons les messages suivants dans le journal `/var/log/patroni/acme/patroni.log` :

```
INFO: Selected new etcd server http://10.0.0.11:2379
INFO: No PostgreSQL configuration items changed, nothing to reload.
INFO: Lock owner: None; I am p1.hapat.vm
INFO: trying to bootstrap a new cluster
INFO: postmaster pid=18415
INFO: establishing a new patroni heartbeat connection to postgres
INFO: running post_bootstrap
WARNING: Could not activate Linux watchdog device: Can't open watchdog device:
 ↪  [Errno 13] Permission denied: '/dev/watchdog'
```

```
INFO: initialized a new cluster
INFO: no action. I am (p1.hapat.vm), the leader with the lock
```

Le démon Patroni démarre, choisi un serveur etcd, se saisit du *leader lock* et initialise l'instance PostgreSQL locale. Les sorties standard et d'erreur des commandes exécutées par Patroni n'est pas capturée vers le journal de ce dernier. Ces commandes sont donc capturées par journald et associées au service `patroni`. Nous y retrouvons par exemple la sortie de `initdb` :

```
# journalctl -u patroni
[…]
patroni: The files belonging to this database system will be owned by user
 ↪  "postgres".
patroni: This user must also own the server process.
patroni: The database cluster will be initialized with locale "C.UTF-8".
[…]
patroni: Success. You can now start the database server using:
patroni:     /usr/lib/postgresql/16/bin/pg_ctl -D /var/lib/postgresql/16/main -l
 ↪  logfile start
```

Sur `p2`, nous trouvons dans le journal correspondant :

```
INFO: Selected new etcd server http://10.0.0.13:2379
INFO: No PostgreSQL configuration items changed, nothing to reload.
INFO: Lock owner: p1.hapat.vm; I am p2.hapat.vm
INFO: trying to bootstrap from leader 'p1.hapat.vm'
INFO: replica has been created using basebackup
INFO: bootstrapped from leader 'p1.hapat.vm'
INFO: postmaster pid=18195
INFO: Lock owner: p1.hapat.vm; I am p2.hapat.vm
INFO: establishing a new patroni heartbeat connection to postgres
INFO: no action. I am (p2.hapat.vm), a secondary, and following a leader
 ↪  (p1.hapat.vm)
```

Comme sur `p1`, le démon Patroni démarre et choisi un serveur etcd, mais il découvre le *leader lock* appartient déjà à `p1`. L'instance PostgreSQL locale n'existant pas, Patroni décide de la créer depuis celle de `p1`.

Dans les deux cas, la configuration par défaut des journaux applicatifs de PostgreSQL les places dans le répertoire `PGDATA/log`, donc ici `/var/lib/pgsql/16/data/log/`, équivalent à `~postgres/16/data/log`.

- observer la topologie de l'agrégat avec `patronictl`.

L'utilisation de `patronictl` nécessite un fichier de configuration permettant de déterminer le nom du cluster et idéalement les nœuds du DCS. Sur les machines `p1` et `p2`, nous pouvons utiliser directement le fichier de configuration de Patroni `/etc/patroni/patroni.yml`. La commande devient :

```
$ patronictl -c /etc/patroni/patroni.yml topology
+ Cluster: acme (7350009258581743592) +-----------+----+-----------+------+
| Member        | Host      | Role    | State     | TL | Lag in MB | Tags |
+---------------+-----------+---------+-----------+----+-----------+------+
```

```
| p1.hapat.vm   | 10.0.0.21 | Leader  | running   |  1 |           | […]  |
| + p2.hapat.vm | 10.0.0.22 | Replica | streaming |  1 |         0 | […]  |
+---------------+-----------+---------+-----------+----+-----------+------+
```

Nous constatons que `p1` héberge bien l'instance primaire et `p2` la secondaire.

Pour simplifier cette commande, il est possible de positionner dans votre environnement le variable `PATRONICTL_CONFIG_FILE`. Par exemple :

```
$ export PATRONICTL_CONFIG_FILE=/etc/patroni/patroni.yml
$ patronictl topology
[…]
```

Pour la positionner automatiquement, vous pouvez par exemple créer le fichier `/etc/profile.d/99-patroni.sh` avec le contenu suivant :

```
cat <<EOF > /etc/profile.d/99-patroni.sh
PATRONICTL_CONFIG_FILE=/etc/patroni/patroni.yml
export PATRONICTL_CONFIG_FILE
EOF
chmod +x /etc/profile.d/99-patroni.sh
```

> – Ajouter le 3ème nœud à l'agrégat.

Répéter les étapes précédentes sur le serveur `p3` :

- installation des dépôts PGDG ;
- installation de PostgreSQL et Patroni ;
- création du fichier de configuration de Patroni ;
- démarrage de l'instance sur `p3`.

Après l'ajout du troisième nœud, la topologie est la suivante :

```
$ patronictl topology
+ Cluster: acme (7350009258581743592) +-----------+----+-----------+------+
| Member        | Host      | Role    | State     | TL | Lag in MB | Tags |
+---------------+-----------+---------+-----------+----+-----------+------+
| p1.hapat.vm   | 10.0.0.21 | Leader  | running   |  1 |           | […]  |
| + p2.hapat.vm | 10.0.0.22 | Replica | streaming |  1 |         0 | […]  |
| + p3.hapat.vm | 10.0.0.23 | Replica | streaming |  1 |         0 | […]  |
+---------------+-----------+---------+-----------+----+-----------+------+
```

Il est conseillé lors des tests de garder une fenêtre répétant l'ordre régulièrement :

```
$ watch -n1 patronictl topology
```

> – Déterminer le primaire via l'API Patroni en direct avec `curl`.

La configuration est visible de l'extérieur :

```
$ curl -s http://p1:8008/patroni | jq
```

```
{
  "state": "running",
  "postmaster_start_time": "[…]",
  "role": "master",
  "server_version": 160002,
  "xlog": {
    "location": 50651960
  },
  "timeline": 1,
  "replication": [
    {
      "usename": "replicator",
      "application_name": "p2.hapat.vm",
      "client_addr": "10.0.0.22",
      "state": "streaming",
      "sync_state": "async",
      "sync_priority": 0
    },
    {
      "usename": "replicator",
      "application_name": "p3.hapat.vm",
      "client_addr": "10.0.0.23",
      "state": "streaming",
      "sync_state": "async",
      "sync_priority": 0
    }
  ],
  "dcs_last_seen": 1711308128,
  "tags": {
    "clonefrom": true,
    "failover_priority": 1
  },
  "database_system_identifier": "7350009258581743592",
  "patroni": {
    "version": "3.2.2",
    "scope": "acme",
    "name": "p1.hapat.vm"
  }
}
```


De manière plus précise :

```
$ curl -s http://p1:8008/cluster |\
jq '.members[] | select ((.role == "leader" ) and ( .state == "running")) | { name }'
```


```
{
  "name": "p1.hapat.vm"
}
```


> – Quels sont les slots de réplication sur le *leader* ?

Par défaut, les slots de réplications portent le nom des nœuds réplicas.

```
$ curl -s http://p1:8008/cluster | jq '.members[] | select (.role == "replica") | {
↪  name }'
```

```
{
  "name": "p2.hapat.vm"
}
{
  "name": "p3.hapat.vm"
}
```

On peut le vérifier en se connectant à PostgreSQL sur le serveur primaire (ici `p1`) :

```
$ sudo -iu postgres
$ psql -c "SELECT slot_name FROM pg_replication_slots"
 slot_name
-----------
 p2_hapat_vm
 p3_hapat_vm
```

> – Forcer l'utilisation du watchdog dans la configuration de Patroni. Que se passe-t-il ?

Ajoutez la section watchdog dans le fichier de configuration de Patroni `/etc/patroni/patroni.yml` en positionnant `mode: required` :

```
watchdog:
  mode: required
  # device: /dev/watchdog
  safety_margin: 5
```

Il nous faut recharger la configuration de Patroni :

```
$ patronictl reload acme
+ Cluster: acme (7350009258581743592) -------------+----+-----------+
| Member      | Host      | Role    | State         | TL | Lag in MB |
+-------------+-----------+---------+---------------+----+-----------+
| p1.hapat.vm | 10.0.0.21 | Replica | start failed |    |   unknown |
| p2.hapat.vm | 10.0.0.22 | Replica | start failed |    |   unknown |
| p3.hapat.vm | 10.0.0.23 | Leader  | running      |  6 |           |
+-------------+-----------+---------+---------------+----+-----------+
Are you sure you want to reload members p1.hapat.vm, p2.hapat.vm, p3.hapat.vm?
 ↪  [y/N]: y
Reload request received for member p1.hapat.vm and will be processed within 10
 ↪  seconds
Reload request received for member p2.hapat.vm and will be processed within 10
 ↪  seconds
Reload request received for member p3.hapat.vm and will be processed within 10
 ↪  seconds
```

Suite à cette modification, le cluster Patroni se retrouve sans primaire. Dans les journaux applicatif de `p3`, ici précédemment marqué *Leader*, nous trouvons :

```
INFO: Reloading PostgreSQL configuration.
INFO: Lock owner: p3.hapat.vm; I am p3.hapat.vm
ERROR: Configuration requires watchdog, but watchdog could not be configured.
INFO: Demoting self (immediate)
INFO: Leader key released
INFO: Demoting self because watchdog could not be activated
```

```
INFO: Lock owner: None; I am p3.hapat.vm
INFO: not healthy enough for leader race
INFO: starting after demotion in progress
INFO: closed patroni connections to postgres
INFO: postmaster pid=561
INFO: establishing a new patroni heartbeat connection to postgres
WARNING: Watchdog device is not usable
INFO: Dropped unknown replication slot 'p1_hapat_vm'
INFO: Dropped unknown replication slot 'p2_hapat_vm'
INFO: following a different leader because i am not the healthiest node
```

- Donner les droits à l'utilisateur `postgres` sur le fichier `/dev/watchdog`. Après quelques secondes, que se passe-t-il ?

Un simple `chown` sur le device `/dev/watchdog` de chaque VM peut suffire, au moins le temps de ce TP. Néanmoins, il ne survivrait pas au redémarrage de la machine.

Le fichier de service de Patroni propose de systématiquement modifier les droits sur ce device décommentant la ligne suivante :

```
#ExecStartPre=-/usr/bin/sudo /bin/chown postgres /dev/watchdog
```

Une autre solution est de configurer `udev` afin qu'il modifie les droits sur ce fichier automatiquement à chaque démarrage :

```
cat <<'EOF' > /etc/udev/rules.d/99-watchdog.rules
# give writes on watchdog device to postgres
SUBSYSTEM=="misc", KERNEL=="watchdog", ACTION=="add", RUN+="/bin/chown postgres
 ↪  /dev/watchdog"

# Or a better solution using ACL:
#SUBSYSTEM=="misc", KERNEL=="watchdog", ACTION=="add", RUN+="/bin/setfacl -m
 ↪  u:postgres:rw- /dev/watchdog"
EOF
```

Une fois les droits positionnés, l'un des nœuds Patroni devrait finalement réussir à activer le watchdog et ainsi devenir *leader* et promouvoir son instance PostgreSQL :

```
INFO: i6300ESB timer activated with 25 second timeout, timing slack 15 seconds
INFO: promoted self to leader by acquiring session lock
INFO: Lock owner: p2.hapat.vm; I am p2.hapat.vm
INFO: updated leader lock during promote
INFO: Lock owner: p2.hapat.vm; I am p2.hapat.vm
INFO: no action. I am (p2.hapat.vm), the leader with the lock
```

1.12.2 Patroni : utilisation

- Se connecter depuis l'extérieur à la base **postgres**, d'un des nœuds.

La connexion extérieure peut se faire depuis la machine hôte des VM. Il est nécessaire d'y installer un client PostgreSQL, par exemple `postgresql-client`. Pour se connecter à l'instance `p1`, utiliser l'une des commandes suivantes :

```
$ psql -h p1 -p 5432 -U postgres -d postgres
$ psql -h 10.0.0.21 postgres postgres
```

En cas de problème, il faut regarder :

- les règles de *firewall* ;
- les traces du nœud PostgreSQL concerné, plus instructives que le simple message d'erreur du client ;
- le fichier de configuration `pg_hba.conf`.

- Comment obtenir une connexion en lecture et écrire ?

Il est nécessaire de se connecter à l'instance primaire pour réaliser des écritures en base. La chaîne de connexion permettant d'indiquer plusieurs nœuds, nous pouvons y préciser tous les nœuds du cluster. Afin de sélectionner le nœud primaire, il suffit d'ajouter le paramètre `target_session_attrs=primary` ou `target_session_attrs=read-write`.

Par exemple :

```
psql "host=p1,p2,p3 user=postgres target_session_attrs=primary"
psql "postgresql://postgres@p1,p2,p3/postgres?target_session_attrs=read-write"
```

Pour une connexion en lecture seule, nous utilisons par exemple :

```
psql "postgresql://postgres@p1,p2,p3:5432/postgres?target_session_attrs=prefer-
↪  standby"
```

- Créer une table :

```
CREATE TABLE insertions (
  id     int         GENERATED ALWAYS AS IDENTITY,
  d      timestamptz DEFAULT now(),
  source text        DEFAULT inet_server_addr()
);
```

- Insérer une ligne toutes les secondes, à chaque fois dans une nouvelle connexion au primaire.
- Dans une autre fenêtre, afficher les 20 dernières lignes de cette table.

Dans la colonne `source` de la table créée, la valeur par défaut fait appel à la fonction `inet_server_addr()` qui retourne l'adresse IP du serveur PostgreSQL sur lequel nous sommes connectés.

Lancer une insertion toutes les secondes :

```
watch -n1 'psql -X -d "host=p1,p2,p3 user=postgres target_session_attrs=primary" \
               -c "INSERT INTO insertions SELECT;"'
```

Lire les vingt dernières lignes de la table :

```
watch -n1 'psql -X -d "host=p1,p2,p3 port=5432 user=postgres" \
               -c "SELECT * FROM insertions ORDER BY d DESC LIMIT 20"'
```

Bien entendu, nous obtenons toujours le même nœud dans la colonne `source`, ici `p1` :

```
 id  |               d                |    source
-----+--------------------------------+---------------
 …
 313 | 2024-03-05 15:40:03.118307+00 | 10.0.0.21/32
 312 | 2024-03-05 15:40:02.105116+00 | 10.0.0.21/32
 311 | 2024-03-05 15:40:01.090775+00 | 10.0.0.21/32
 310 | 2024-03-05 15:40:00.075847+00 | 10.0.0.21/32
 309 | 2024-03-05 15:39:59.061759+00 | 10.0.0.21/32
 308 | 2024-03-05 15:39:58.048074+00 | 10.0.0.21/32

(20 lignes)
```

- Stopper le nœud *leader* Patroni.
- Que se passe-t-il dans la topologie, et dans les requêtes ci-dessus ?

Sur **p1** :

```
# systemctl stop patroni
```

Après l'arrêt de `p1`, `p3` prend le rôle de *leader* :

```
$ patronictl -c /etc/patroni/patroni.yml topology
+ Cluster: acme (7350009258581743592) +---------+----+-----------+------+
| Member        | Host      | Role    | State   | TL | Lag in MB | Tags |
+---------------+-----------+---------+---------+----+-----------+------+
| p3.hapat.vm   | 10.0.0.23 | Leader  | running |  2 |           | […]  |
| + p1.hapat.vm | 10.0.0.21 | Replica | stopped |    |   unknown | […]  |
| + p2.hapat.vm | 10.0.0.22 | Replica | running |  1 |         0 | […]  |
+---------------+-----------+---------+---------+----+-----------+------+
```

Les insertions échouent le temps de la bascule, ici pendant environ 2 secondes, puis continuent depuis l'autre nœud :

```
 id  |               d                |    source
-----+--------------------------------+---------------
 …
 445 | 2024-03-24 15:50:17.840394+00 | 10.0.0.23/32
 444 | 2024-03-24 15:50:16.823004+00 | 10.0.0.23/32
 431 | 2024-03-24 15:50:14.755045+00 | 10.0.0.21/32
 430 | 2024-03-24 15:50:13.740541+00 | 10.0.0.21/32
```

- Arrêter les processus du nouveau primaire. Il ne reste qu'un nœud actif. Que se passe-t-il ?

Le *leader* Patroni a changé à nouveau :

```
$ patronictl -c /etc/patroni/patroni.yml topology
+ Cluster: acme (7350009258581743592) -------+----+-----------+------+
| Member      | Host      | Role   | State   | TL | Lag in MB | Tags |
+-------------+-----------+--------+---------+----+-----------+------+
| p2.hapat.vm | 10.0.0.22 | Leader | running |  6 |           | […]  |
+-------------+-----------+--------+---------+----+-----------+------+
```

> – Arrêter deux nœuds du cluster etcd. Que se passe-t-il ?

Sur `e1` et `e2` :

```
# systemctl stop etcd
```

Il n'y a plus de quorum etcd garantissant une référence. Le cluster Patroni se met en lecture seule et les insertions tombent en échec puisqu'elles exigent une connexion ouverte en écriture :

```
psql: error: connection to server at "p1" (10.0.0.21), port 5432 failed: Connection
 ↪  refused
        Is the server running on that host and accepting TCP/IP connections?
connection to server at "p2" (10.0.0.22), port 5432 failed: server is in hot standby
 ↪  mode
connection to server at "p3" (10.0.0.23), port 5432 failed: Connection refused
        Is the server running on that host and accepting TCP/IP connections?
```

> – Redémarrer les nœuds etcd.

Les écritures reprennent.

> – Relancer un des nœuds Patroni.

Dans le cadre de cette correction `p2` est l'actuel *leader*, nous redémarrons donc Patroni sur `p1` :

```
# systemctl restart patroni
```

L'instance sur `p1` se raccroche en secondaire :

```
$ patronictl -c /etc/patroni/patroni.yml topology
+ Cluster: acme (7350009258581743592) +-----------+----+-----------+------+
| Member        | Host      | Role    | State     | TL | Lag in MB | Tags |
+---------------+-----------+---------+-----------+----+-----------+------+
| p2.hapat.vm   | 10.0.0.22 | Leader  | running   |  7 |           | […]  |
| + p1.hapat.vm | 10.0.0.21 | Replica | streaming |  7 |         0 | […]  |
+---------------+-----------+---------+-----------+----+-----------+------+
```

> – Sur le troisième nœud (arrêté), détruire le `PGDATA`. Relancer Patroni.

Le PGDATA de nos instances se trouve dans `/var/lib/pgsql/16/data`. Supprimons ce `PGDATA` sur `p3`, le nœud restant :

```
rm -rf /var/lib/pgsql/16/data
```

Relançons Patroni sur ce nœud :

```
# systemctl start patroni
```

Nous observons dans les journaux de Patroni et PostgreSQL que l'instance est recrée et se raccroche à `p2` :

```
LOG:  starting PostgreSQL 16.2 on x86_64-pc-linux-gnu, […]
LOG:  listening on IPv4 address "10.0.0.23", port 5432
LOG:  listening on Unix socket "/run/postgresql/.s.PGSQL.5432"
LOG:  listening on Unix socket "/tmp/.s.PGSQL.5432"
LOG:  database system was interrupted while in recovery at log time […]
HINT:  If this has occurred more than once some data might be corrupted and you might
 ↪  need to choose an earlier recovery target.
LOG:  entering standby mode
LOG:  starting backup recovery with redo LSN 0/45405620, checkpoint LSN 0/4FD4D788,
 ↪  on timeline ID 7
LOG:  redo starts at 0/45405620
LOG:  completed backup recovery with redo LSN 0/45405620 and end LSN 0/509EA568
LOG:  consistent recovery state reached at 0/509EA568
LOG:  database system is ready to accept read-only connections
LOG:  started streaming WAL from primary at 0/50000000 on timeline 7
```

Le temps de la reconstruction de zéro, il est possible de voir l'évolution de l'état de `p3` de `creating replica` à `streaming` :

```
$ export PATRONICTL_CONFIG_FILE=/etc/patroni/patroni.yml
$ patronictl topology
+ Cluster: acme (7350009258581743592) +------------------+----+-----------+------+
| Member        | Host      | Role    | State            | TL | Lag in MB | Tags |
+---------------+-----------+---------+------------------+----+-----------+------+
| p2.hapat.vm   | 10.0.0.22 | Leader  | running          |  7 |           | […]  |
| + p1.hapat.vm | 10.0.0.21 | Replica | streaming        |  7 |         0 | […]  |
| + p3.hapat.vm | 10.0.0.23 | Replica | creating replica |    |   unknown | […]  |
+---------------+-----------+---------+------------------+----+-----------+------+

/* Après un certain temps */

$ patronictl topology
+ Cluster: acme (7350009258581743592) +-----------+----+-----------+------+
| Member        | Host      | Role    | State     | TL | Lag in MB | Tags |
+---------------+-----------+---------+-----------+----+-----------+------+
| p2.hapat.vm   | 10.0.0.22 | Leader  | running   |  7 |           | […]  |
| + p1.hapat.vm | 10.0.0.21 | Replica | streaming |  7 |         0 | […]  |
| + p3.hapat.vm | 10.0.0.23 | Replica | streaming |  7 |         0 | […]  |
+---------------+-----------+---------+-----------+----+-----------+------+
```

- Forcer un *failover* vers le nœud `p1`.

```
$ patronictl failover
Current cluster topology
+ Cluster: acme (7350009258581743592) ----------+----+-----------+------+
| Member      | Host      | Role    | State     | TL | Lag in MB | Tags |
+-------------+-----------+---------+-----------+----+-----------+------+
| p1.hapat.vm | 10.0.0.21 | Replica | streaming |  7 |         0 | […]  |
+-------------+-----------+---------+-----------+----+-----------+------+
| p2.hapat.vm | 10.0.0.22 | Leader  | running   |  7 |           | […]  |
+-------------+-----------+---------+-----------+----+-----------+------+
| p3.hapat.vm | 10.0.0.23 | Replica | streaming |  7 |         0 | […]  |
+-------------+-----------+---------+-----------+----+-----------+------+
Candidate ['p1', 'p3'] []: p1
```

```
Are you sure you want to failover cluster acme, demoting current leader p2? [y/N]: y
[…] Successfully failed over to "p1"
+ Cluster: acme (7350009258581743592) --------+----+-----------+------+
| Member      | Host      | Role    | State   | TL | Lag in MB | Tags |
+-------------+-----------+---------+---------+----+-----------+------+
| p1.hapat.vm | 10.0.0.21 | Leader  | running |  7 |           | […]  |
+-------------+-----------+---------+---------+----+-----------+------+
| p2.hapat.vm | 10.0.0.22 | Replica | stopped |    |   unknown | […]  |
+-------------+-----------+---------+---------+----+-----------+------+
| p3.hapat.vm | 10.0.0.23 | Replica | running |  7 |         0 | […]  |
+-------------+-----------+---------+---------+----+-----------+------+

$ patronictl topology
+ Cluster: acme (7350009258581743592) +-----------+----+-----------+------+
| Member        | Host      | Role    | State     | TL | Lag in MB | Tags |
+---------------+-----------+---------+-----------+----+-----------+------+
| p1.hapat.vm   | 10.0.0.21 | Leader  | running   |  8 |           | […]  |
| + p2.hapat.vm | 10.0.0.22 | Replica | streaming |  8 |         0 | […]  |
| + p3.hapat.vm | 10.0.0.23 | Replica | streaming |  8 |         0 | […]  |
+---------------+-----------+---------+-----------+----+-----------+------+
```

- Modifier les paramètres `shared_buffers` et `work_mem`. Si besoin, redémarrer les nœuds.

Pour modifier la configuration, nous devons utiliser `patronictl`, plutôt qu'éditer directement les fichiers de configuration. Une alternative est de modifier la configuration statique, dans le fichier YAML `/etc/patroni/patroni.yml`. Cette méthode facilite leur maintenance, mais impose que le contenu soit identique sur tous les nœuds, ce qui est généralement le cas dans un déploiement industrialisé.

La commande `patronictl edit-config` appelle l'éditeur par défaut, souvent `vi`, `vim` ou `nano`. Vous pouvez modifier la variable d'environnement `EDITOR` pour pointer sur votre éditeur favori.

Éditons la configuration dynamique et ajoutons les deux paramètres :

```
$ export PATRONICTL_CONFIG_FILE=/etc/patroni/patroni.yml
$ patronictl edit-config
[…]

---
+++
@@ -12,6 +12,8 @@
     wal_keep_size: 128MB
     wal_level: replica
     wal_log_hints: 'on'
+    shared_buffers: 300MB
+    work_mem: 50MB
   use_pg_rewind: false
   use_slots: true
 retry_timeout: 10


Apply these changes? [y/N]: y
Configuration changed
```

Les nœuds sont à redémarrer à cause de la modification de `shared_buffers` :

```
$ patronictl topology
+ Cluster: acme (7350009258581743592) +-----------+----+------+-----------------+-
| Member        | Host      | Role    | State     | TL | Lag… | Pending restart |
+---------------+-----------+---------+-----------+----+------+-----------------+-
| p1.hapat.vm   | 10.0.0.21 | Leader  | running   |  1 |      | *               |
| + p2.hapat.vm | 10.0.0.22 | Replica | streaming |  1 |    0 | *               |
| + p3.hapat.vm | 10.0.0.23 | Replica | streaming |  1 |    0 | *               |
+---------------+-----------+---------+-----------+----+------+-----------------+-
```

Commandons le redémarrage de PostgreSQL sur les trois nœuds :

```
$ patronictl restart acme
+ Cluster: acme (7350009258581743592) ----------+----+------+-----------------+-
| Member      | Host      | Role    | State     | TL | Lag… | Pending restart |
+-------------+-----------+---------+-----------+----+------+-----------------+-
| p1.hapat.vm | 10.0.0.21 | Leader  | running   |  1 |      | *               |
+-------------+-----------+---------+-----------+----+------+-----------------+-
| p2.hapat.vm | 10.0.0.22 | Replica | streaming |  1 |    0 | *               |
+-------------+-----------+---------+-----------+----+------+-----------------+-
| p3.hapat.vm | 10.0.0.23 | Replica | streaming |  1 |    0 | *               |
+-------------+-----------+---------+-----------+----+------+-----------------+-
When should the restart take place (e.g. […])  [now]:
Are you sure you want to restart members p1.hapat.vm, p2.hapat.vm, p3.hapat.vm?
 ↪  [y/N]: y
Restart if the PostgreSQL version is less than provided (e.g. 9.5.2)  []:
Success: restart on member p1.hapat.vm
Success: restart on member p2.hapat.vm
Success: restart on member p3.hapat.vm
```

Vérifions que le paramétrage a bien été modifié :

```
$ for h in p1 p2 p3; do echo -ne $h:; psql -Xtd "host=$h user=postgres" -c "show
 ↪  shared_buffers"; done
p1: 300MB

p2: 300MB

p3: 300MB
```

Noter que le contenu des modifications est tracé dans un fichier `patroni.dynamic.json` dans le `PGDATA` :

```
$ jq . patroni.dynamic.json

{
  "loop_wait": 10,
  "postgresql": {
    "parameters": {
      "hot_standby": "on",
      "max_connections": 100,
      "max_locks_per_transaction": 64,
      "max_prepared_transactions": 0,
      "max_replication_slots": 10,
      "max_wal_senders": 10,
      "max_worker_processes": 8,
      "track_commit_timestamp": "off",
```

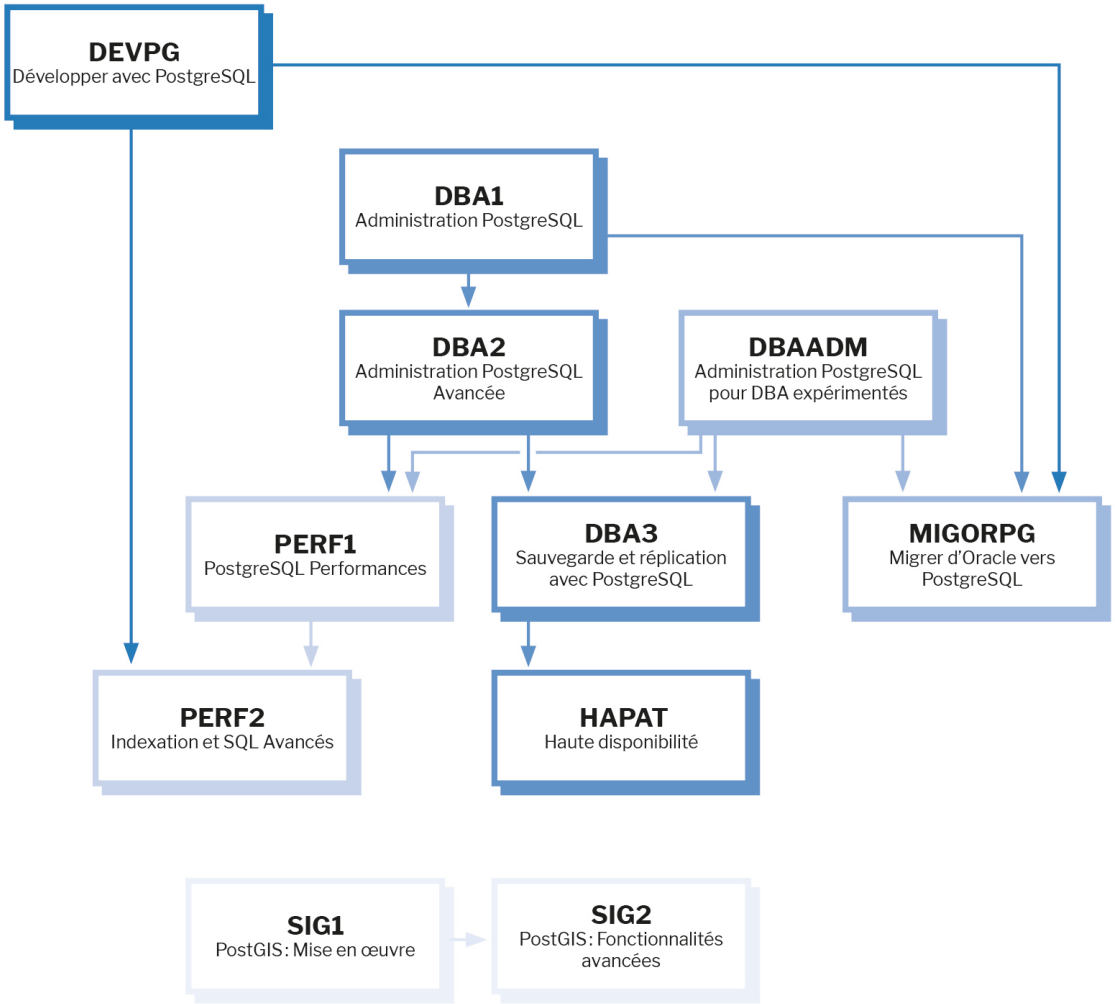
```
      "wal_keep_size": "128MB",
      "wal_level": "replica",
      "wal_log_hints": "on",
      "shared_buffers": "300MB",
      "work_mem": "50MB"
    },
    "use_pg_rewind": false,
    "use_slots": true
  },
  "retry_timeout": 10,
  "ttl": 30
}
```

Les formations Dalibo

Retrouvez nos formations et le calendrier sur https://dali.bo/formation

Pour toute information ou question, n'hésitez pas à nous écrire sur contact@dalibo.com.

Cursus des formations



Retrouvez nos formations dans leur dernière version :

- DBA1 : Administration PostgreSQL
  https://dali.bo/dba1
- DBA2 : Administration PostgreSQL avancé
  https://dali.bo/dba2
- DBA3 : Sauvegarde et réplication avec PostgreSQL
  https://dali.bo/dba3
- DEVPG : Développer avec PostgreSQL
  https://dali.bo/devpg
- PERF1 : PostgreSQL Performances
  https://dali.bo/perf1
- PERF2 : Indexation et SQL avancés
  https://dali.bo/perf2
- MIGORPG : Migrer d'Oracle à PostgreSQL
  https://dali.bo/migorpg
- HAPAT : Haute disponibilité avec PostgreSQL
  https://dali.bo/hapat

Les livres blancs

- Migrer d'Oracle à PostgreSQL
  https://dali.bo/dlb01
- Industrialiser PostgreSQL
  https://dali.bo/dlb02
- Bonnes pratiques de modélisation avec PostgreSQL
  https://dali.bo/dlb04
- Bonnes pratiques de développement avec PostgreSQL
  https://dali.bo/dlb05

Téléchargement gratuit

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open source ou sous licence Creative Commons.