Formation HAPAT

Haute disponibilité avec Patroni

DALIBO
L'expertise PostgreSQL

25.03

Table des matières

Sur ce document

Formation	Formation HAPAT
Titre	Haute disponibilité avec Patroni
Révision	25.03
ISBN	978-2-38168-136-8
PDF	https://dali.bo/hapat_pdf
EPUB	https://dali.bo/hapat_epub
HTML	https://dali.bo/hapat_html
Slides	https://dali.bo/hapat_slides

Vous trouverez en ligne les différentes versions complètes de ce document. Les solutions de TP ne figurent pas forcément dans la version imprimée, mais sont dans les versions numériques (PDF ou HTML).

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 :

Alexandre Anriot, Jean-Paul Argudo, Carole Arnaud, Alexandre Baron, David Bidoc, Sharon Bonan, Franck Boudehen, Arnaud Bruniquel, Pierrick Chovelon, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Ronan Dunklau, Vik Fearing, Stefan Fercot, Dimitri Fontaine, Pierre Giraud, Nicolas Gollet, 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, Jehan-Guillaume de Rorthais, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Arnaud de Vathaire, 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

Cette formation est sous licence **CC-BY-NC-SA**[2]. Vous êtes libre de la redistribuer et/ou modifier aux conditions suivantes :

- Paternité
- Pas d'utilisation commerciale
- Partage des conditions initiales à l'identique

Vous n'avez pas le droit d'utiliser cette création à des fins commerciales.

Si vous modifiez, transformez ou adaptez cette création, vous n'avez le droit de distribuer la création qui en résulte que sous un contrat identique à celui-ci.

Vous devez citer le nom de l'auteur original de la manière indiquée par l'auteur de l'œuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'œuvre). À chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition. La meilleure manière de les indiquer est un lien vers cette page web. Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits sur cette œuvre. Rien dans ce contrat ne diminue ou ne restreint le droit moral de l'auteur ou des auteurs.

Le texte complet de la licence est disponible sur http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

[2] http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

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 13 à 17.

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/ Généralités sur la haute disponibilité

1.1 INTRODUCTION

- Définition de « Haute Disponibilité »
- Contraintes à définir
- Contraintes techniques
- Solutions existantes

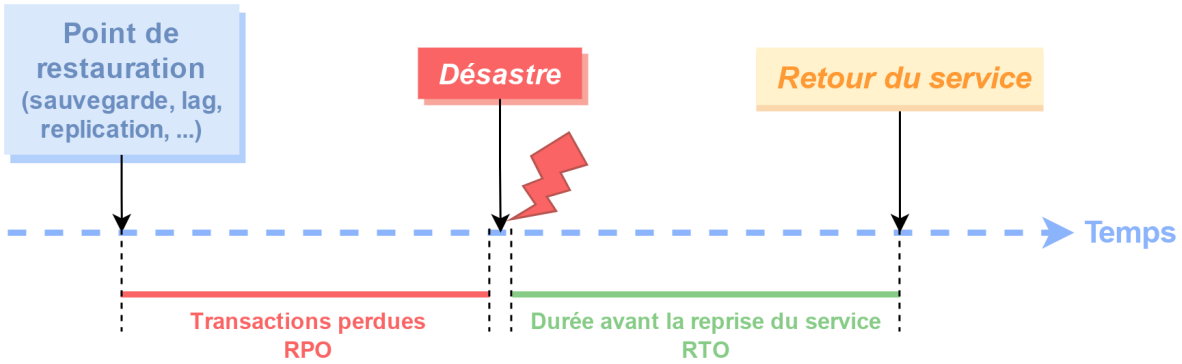
La haute disponibilité est un sujet complexe. Plusieurs outils libres coexistent au sein de l'écosystème PostgreSQL, chacun abordant le sujet d'une façon différente.

Ce document clarifie la définition de « haute disponibilité », des méthodes existantes et des contraintes à considérer. L'objectif est d'aider à la prise de décision et le choix de la solution.

1.2 DÉFINITIONS

- RTO / RPO
- Haute Disponibilité de données / de service

1.2.1 RTO / RPO



Deux critères essentiels permettent de contraindre le choix d'une solution : le RTO (*recovery time objectives*) et le RPO (*recovery point objective*).

Le RTO représente la durée maximale d'interruption de service admissible, depuis la coupure de service jusqu'à son rétablissement. Il inclut le délai de détection de l'incident, le délai de prise en charge et le temps de mise en œuvre des actions correctives. Un RTO peut tendre vers zéro mais ne l'atteint jamais parfaitement. Une coupure de service est le plus souvent **inévitable**, aussi courte soit-elle.

Le RPO représente la durée maximale d'activité de production déjà réalisée que l'on s'autorise à perdre en cas d'incident. Contrairement au RTO, le RPO peut atteindre l'objectif de zéro perte.

Les deux critères sont complémentaires. Ils ont une influence **importante** sur le choix d'une solution **et** sur son coût total. Plus les RTO et RPO sont courts, plus la solution est complexe. Cette complexité se répercute directement sur le coût de mise en œuvre, de formation et de maintenance.

Le coût d'une architecture est exponentiel par rapport à sa disponibilité.

1.2.2 Haute disponibilité de service

- Continuité d'activité malgré incident
- Redondance à tous les niveaux
  - réseaux, stockage, serveurs, administrateurs…
  - réplication des données
- Automatisation des bascules

La **Haute Disponibilité de service** définit les moyens techniques mis en œuvre pour garantir une continuité d'activité suite à un incident sur un service.

La haute disponibilité de service nécessite de redonder tous les éléments nécessaires à l'activité du service : l'alimentation électrique, ses accès réseaux, le réseau lui-même, les serveurs, le stockage, les administrateurs, etc.

En plus de cette redondance, une technique de réplication synchrone ou asynchrone est souvent mise en œuvre afin de maintenir à l'identique ou presque les serveurs redondés.

Pour que la disponibilité ne soit pas affectée par le temps de réaction des humains, on peut rechercher à automatiser les bascules vers un serveur sain en cas de problème.

1.2.3 Haute disponibilité des données

- Perte faible ou nulle de données après incident
  - redonder les données
  - garantir les écritures à plusieurs endroits
- Contradictoire avec la haute disponibilité de service
  - arbitrage possible avec une complexité et un budget plus importants

La **Haute disponibilité des données** définit les moyens techniques mis en œuvre pour garantir une perte faible voire nulle de données en cas d'incident. Ce niveau de disponibilité des données est assuré en redondant les données sur plusieurs systèmes physiques distincts et en assurant que chaque écriture est bien réalisée sur plusieurs d'entre eux.

Dans le cas d'une réplication synchrone entre les systèmes, les écritures sont suspendues tant qu'elles ne peuvent être validées de façon fiable sur au moins deux systèmes.

Autrement dit, la haute disponibilité des données et la haute disponibilité de service sont contradictoires, le premier nécessitant d'interrompre le service en écriture si l'ensemble ne repose que sur un seul système.

Par exemple, un RAID 1 fonctionnant sur un seul disque suite à un incident n'est PAS un environnement à haute disponibilité des données, mais à haute disponibilité de service.

La position du curseur entre la haute disponibilité de service et la haute disponibilité de données guide aussi le choix de la solution. S'il est possible d'atteindre le double objectif, l'impact sur les solutions possibles et le coût est une fois de plus important.

1.3 SAUVEGARDES

- Composant déjà présent
- Travail d'optimisation à effectuer
- RTO de quelques minutes possibles
- RPO de quelques minutes (secondes ?) facilement
- Et ne pas oublier de tester

Les différentes méthodes de sauvegardes de PostgreSQL (logique avec `pg_dump`, ou PITR avec pgBackRest ou d'autres outils) sont souvent sous-estimées. Elles sont pourtant un élément essentiel de toute architecture qui est souvent déjà présent. On n'oubliera d'ailleurs pas de tester régulièrement ses sauvegardes[1].

Investir dans l'optimisation des sauvegardes peut déjà assurer un certain niveau de disponibilité de votre service, à moindre coût.

Quoi qu'il en soit, la sauvegarde est un élément crucial de toute architecture. Ce sujet doit toujours faire partie de la réflexion autour de la disponibilité d'un service.

1.3.1 Sauvegarde PITR

- Sauvegarde incrémentale binaire
- Optimiser la sauvegarde complète
- Optimiser la restauration complète (RTO)
- Ajuster l'archivage au RPO désiré

La sauvegarde PITR (*Point In Time Recovery*) est une méthode permettant de restaurer une instance PostgreSQL à n'importe quel instant durant la fenêtre de rétention définie, par exemple les dernières 24 heures. Le temps de restauration (RTO) dépend de deux variables : le volume de l'instance et son volume d'écriture.

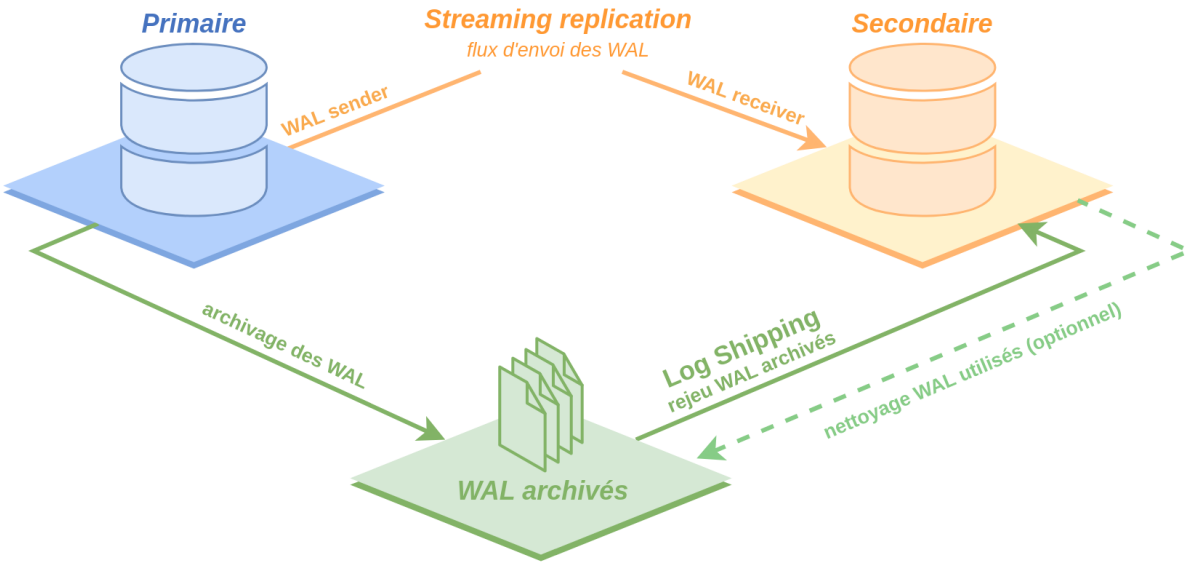
Avec le bon matériel, les bonnes pratiques et une politique de sauvegarde adaptée, il est possible d'atteindre un RTO de quelques minutes, dans la plupart des cas.

La maîtrise du RPO (perte de données) repose sur la fréquence d'archivage des journaux de transactions. Un RPO d'une minute est tout à fait envisageable. En-dessous, nous entrons dans le domaine de

[1] https://blog.dalibo.com/2024/03/29/world_backup_day_restauration.html

la réplication en *streaming*, soit vers une instance secondaire, soit avec l'outil `pg_receivewal` (pour la sauvegarde uniquement). Nous abordons ce sujet dans un futur chapitre.

1.3.2 PITR et redondance par réplication physique



Il est possible d'utiliser les journaux de transactions archivés dans le cadre de la réplication physique. Ces archives deviennent alors un second canal d'échange entre l'instance primaire et ses secondaires, apportant une redondance à la réplication elle-même.

1.3.3 Outils PITR

- Barman
- pgBackRest

Parmi les outils existants et éprouvés au sein de la communauté, nous pouvons citer les deux ci-dessus.

Un module de nos formations les traite en détail[2].

[2] https://dali.bo/i4_html

1.3.4 Bilan PITR

- Utiliser un outil libre issu de l'écosystème PostgreSQL
- Fiabilise l'architecture
- Facilite la mise en œuvre et l'administration
- Couvre déjà certains besoins de disponibilité
- Nécessite une intervention humaine
- Nécessite une supervision fiable

Le principal point faible de la sauvegarde PITR est le temps de prise en compte de l'incident et donc d'intervention d'un administrateur.

Enfin, la sauvegarde PITR doit être surveillée de très près par les équipes d'administration au travers d'une supervision adaptée.

1.4 RÉPLICATION PHYSIQUE

- Réplique les écritures via les journaux de transactions
- Entretient une ou plusieurs instances clones
- Intégrée à PostgreSQL
- Facilité de mise en œuvre
- Réduit RPO/RTO par rapport au PITR
- Plus de matériel
- Architecture et maintenance plus complexes
- Haute disponibilité des données

La réplication physique interne de PostgreSQL réplique le contenu des journaux de transactions. Les instances secondaires sont considérées comme des « clones » de l'instance primaire.

Avec peu de configuration préalable, il est possible de créer des instances secondaires directement à partir de l'instance primaire ou en restaurant une sauvegarde PITR.

La mécanique de réplication est très efficace, car elle ne réplique que les modifications binaires effectuées dans les tables et les index.

Cette étape assure déjà une haute disponibilité de données, ces dernières étant présentes sur plusieurs serveurs distincts.

La réplication permet d'atteindre un RPO et un RTO plus faibles que celui d'une simple sauvegarde PITR, au prix d'un investissement plus important (redondance du matériel), d'une complexification de l'architecture et de sa maintenance. Le RTO deviendra également plus lié au temps de réaction qu'à la bascule technique.

1.4.1 Réplication et RPO

- Réplication asynchrone ou synchrone
- Nécessite un réseau très fiable et performant
- Asynchrone : RPO dépendant du volume d'écriture
  - RPO < 1s hors maintenance et chargement en masse
- Synchrone : RPO = 0
  - 2 secondaires minimum
  - impact sur les performances !

PostgreSQL supporte la réplication asynchrone ou synchrone.

La **réplication asynchrone** autorise un retard entre l'instance primaire et ses secondaires, ce qui implique un RPO supérieur à zéro. Ce retard dépend directement du volume d'écriture envoyé par le primaire et de la capacité du réseau à diffuser ce volume, donc son débit. Une utilisation OLTP a un retard typique inférieur à la seconde. Ce retard peut cependant être plus important lors des périodes de maintenance (`VACUUM`, `REINDEX`) ou lors d'écritures en masse de données.

La **réplication synchrone** s'assure que chaque écriture soit présente sur au moins deux instances avant de valider une transaction. Ce mode permet d'atteindre un RPO de zéro, mais impose d'avoir au minimum trois nœuds dans le cluster, autorisant ainsi la perte complète d'un serveur sans bloquer les écritures. En effet, avec deux nœuds seulement, la disponibilité de service n'est plus assurée : la perte d'un nouveau serveur entraînerait le blocage des écritures.

De plus, le nombre de transactions par seconde dépend directement de la latence du réseau : chaque transaction doit attendre la propagation vers un secondaire et le retour de sa validation.

1.4.2 Réplication et RTO

- Bascule manuelle
- Promotion d'une instance en quelques secondes

La réplication seule n'assure pas de disponibilité de service en cas d'incident.

Comme pour les sauvegardes PITR, le RTO dépend principalement du temps de prise en charge et d'analyse de l'incident par un opérateur. Une fois la décision prise, la promotion d'un serveur secondaire en production ne nécessite qu'une commande et ne prend typiquement que quelques secondes.

Reste ensuite à faire converger les connexions applicatives vers la nouvelle instance primaire, ce qui est facilement automatisé.

1.4.3 Bilan sur la réplication

- 0 ≤ RPO < PITR
- RTO = prise en charge + 30s
- Simple à mettre en œuvre
- Investissement financier et humain plus important

La réplication nécessite donc au minimum deux serveurs, voire trois en cas de réplication synchrone. À ce coût s'ajoutent plusieurs autres plus ou moins cachés :

- le réseau se doit d'être redondé et fiable surtout en cas de réplication synchrone ;
- la formation des équipes d'administration ;
- la mise en œuvre des procédures de construction et de bascule ;
- une supervision plus fine et maîtrisée par les équipes.

1.5 BASCULE AUTOMATISÉE

- Détection d'anomalie et bascule automatique
- HA de service :
  - réduit le temps de prise en charge
- Plusieurs solutions en fonction du besoin
- Beaucoup de contraintes !

Une bascule automatique lors d'un incident permet de réduire le temps d'indisponibilité d'un service au plus bas, assurant ainsi une haute disponibilité de service.

Néanmoins, automatiser la détection d'incident et la prise de décision de basculer un service est un sujet très complexe, difficile à bien appréhender et maintenir, d'autant plus dans le domaine des SGBD.

1.5.1 Prise de décision

- La détection d'anomalie est naïve !
- L'architecture doit pouvoir éviter un *split-brain*
- Solutions éprouvées :
  - *fencing*
  - quorum
  - *watchdog*
  - SBD
- Solutions le plus souvent complémentaires.

Quelle que soit la solution choisie pour détecter les anomalies et déclencher une bascule, celle-ci est toujours très naïve. Contrairement à un opérateur humain, la solution n'a pas de capacité d'analyse et n'a pas accès aux mêmes informations. En cas de non-réponse d'un élément du cluster, il lui est impossible de déterminer dans quel état il se trouve précisément. Sous une charge importante ? Serveur arrêté brutalement ou non ? Réseau coupé ?

Il y a une forte probabilité de *split-brain* si le cluster se contente d'effectuer une bascule sans se préoccuper de l'ancien primaire. Dans cette situation, deux serveurs se partagent la même ressource (IP ou disque ou SGBD) sans le savoir. Corriger le problème et reconsolider les données est fastidieux et

entraîne une indisponibilité plus importante qu'une simple bascule manuelle avec analyse et prise de décision humaine.

Quatre mécaniques permettent de se prémunir plus ou moins d'un *split-brain* : le *fencing*, le quorum, le *watchdog* et le SBD (*Storage Based Death*). La plupart doivent être combinées pour fonctionner de façon optimale.

1.5.2 Mécanique de *fencing*

- Isole un serveur/ressource
  - électriquement
  - arrêt via IPMI, hyperviseur
  - coupe les réseaux
- Utile :
  - pour un serveur muet ou fantôme (*rogue node*)
  - lorsque l'arrêt d'une ressource est perturbé
- Déclenché depuis un des nœuds du cluster
- Nécessite une gestion fine des droits
- Supporté par Pacemaker, embryonnaire dans Patroni

Le *fencing* (clôture) isole un serveur ou une ressource de façon active. Suite à une anomalie, et **avant** la bascule vers le secours prévu, le composant fautif est isolé afin qu'il ne puisse plus interférer avec la production.

Il existe au moins deux anomalies où le *fencing* est incontournable. La première concerne le cas d'un serveur qui ne répond plus au cluster. Il est alors impossible de définir quelle est la situation sur le serveur. Est-il encore vivant ? Les ressources sont-elles encore actives ? Ont-elles encore un comportement normal ? Ont-elles encore accès à l'éventuel disque partagé ? Dans cette situation, la seule façon de répondre avec certitude à ces questions est d'éteindre le serveur. L'action définit avec certitude que les ressources y sont toutes inactives.

La seconde anomalie où le *fencing* est essentiel concerne l'arrêt des ressources. Si le serveur est disponible, communique, mais n'arrive pas à éteindre une ressource (problème technique ou *timeout*), le *fencing* permet « d'escalader » l'extinction de la ressource en extinction du serveur complet.

Il est aussi possible d'isoler un serveur d'une ressource. Le serveur n'est pas éteint, mais son accès à certaines ressources cruciales est coupé, l'empêchant ainsi de corrompre le cluster. L'isolation peut concerner l'accès au réseau Ethernet ou à un disque partagé par exemple.

Il existe donc plusieurs techniques pour un *fencing*, mais il doit toujours être rapide et efficace. Pas de

demi-mesures ! Les méthodes les plus connues soit coupent le courant, donc agissent sur l'UPS[3], ou le PDU[4] ; soit éteignent la machine au niveau matériel via l'IPMI[5] ; soit éteignent la machine virtuelle virtuelle brusquement via son hyperviseur ; soit coupent l'accès au réseau, SAN ou Ethernet.

Par conséquent, cette mécanique nécessite souvent de pouvoir gérer finement les droits d'accès à des opérations d'administration lourdes. C'est le cas par exemple au travers des communautés du protocole SNMP, ou la gestion de droits dans les ESX VMware, les accès au PDU, etc.

1.5.3 Mécanique d'un Quorum

- Chaque serveur possède un ou plusieurs votes
- Utile en cas de partition réseau
- La partition réseau qui a le plus de votes détient le quorum
- La partition qui détient le quorum peut héberger les ressources
- La partition sans quorum doit arrêter toute ressource
- Attention au retour d'une instance dans le cluster
- Supporté par Pacemaker et Patroni (via DCS)

La mécanique du quorum attribue à chaque nœud un (ou plusieurs) vote. Le cluster n'a le droit d'héberger des ressources que s'il possède la majorité absolue des voix. Par exemple, un cluster à 3 nœuds requiert 2 votes pour pouvoir démarrer les ressources, 3 pour un cluster à 5 nœuds, etc.

Lorsque qu'un ou plusieurs nœuds perdent le quorum, ceux-ci doivent arrêter les ressources qu'ils hébergent.

Il est conseillé de maintenir un nombre de nœuds impair au sein du cluster, mais plusieurs solutions existent en cas d'égalité (par exemple par ordre d'identifiant, par poids, serveurs « témoins » ou arbitre, etc).

Le quorum permet principalement de gérer les incidents liés au réseau, quand « tout va bien » sur les serveurs eux-mêmes et qu'ils peuvent éteindre leurs ressources sans problème, à la demande.

Dans le cadre de PostgreSQL, il faut porter une attention particulière au moment où des serveurs isolés rejoignent de nouveau le cluster. Si l'instance primaire a été arrêtée par manque de quorum, elle pourrait ne pas se raccrocher correctement au nouveau primaire, voire corrompre ses propres fichiers de données. En effet, il est impossible de déterminer quelles écritures ont eu lieu sur cet ancien primaire entre sa déconnexion du reste du cluster et son arrêt total.

Pacemaker intègre la gestion du quorum et peut aussi utiliser un serveur de gestion de vote appelé `corosync-qnetd`. Ce dernier est utile en tant que tiers pour gérer le quorum de plusieurs clusters Pacemaker à deux nœuds par exemple.

[3] https://fr.wikipedia.org/wiki/Alimentation_sans_interruption
[4] https://fr.wikipedia.org/wiki/Unit%C3%A9_de_distribution_d%27%C3%A9nergie
[5] https://fr.wikipedia.org/wiki/Intelligent_Platform_Management_Interface

Patroni repose sur un DCS[6] extérieur, par exemple etcd[7], pour stocker l'état du serveur et prendre ses décisions. La responsabilité de la gestion du quorum est donc déléguée au DCS, dont l'architecture robuste est conçue pour toujours présenter des données fiables et de référence à ses clients (ici Patroni).

1.5.4 Mécanique du watchdog

- Équipement matériel intégré partout
  - au pire : `softdog` (moins fiable)
- Compte à rebours avant redémarrage complet du serveur
- À ré-armer par un composant applicatif du serveur, **périodiquement**
- Permet de déclencher du *self-fencing* rapide et fiable
  - meilleure réactivité de l'agrégat
- « Fencing du pauvre », complémentaire du quorum
- Patroni et Pacemaker : oui

Tous les ordinateurs sont désormais équipés d'un *watchdog*. Par exemple, sur un ordinateur portable Dell Latitude, nous trouvons :

```
iTCO_wdt : Intel TCO WatchDog Timer Driver v1.11
```

Sur un Raspberry Pi modèle B :

```
bcm2835-wdt 20100000.watchdog : Broadcom BCM2835 watchdog timer
```

Au besoin, il est aussi possible d'ajouter plusieurs autres *watchdog* grâce à des cartes PCI par exemple, bien que ce ne soit pas nécessaire dans notre cas.

Concernant les machines virtuelles, une configuration supplémentaire est souvent nécessaire pour avoir accès à un *watchdog* virtualisé.

En dernier recours, il est possible de demander au noyau Linux lui-même de jouer le rôle de *watchdog* grâce au module `softdog`. Néanmoins, cette méthode est moins fiable qu'un *watchdog* matériel car il nécessite que le système d'exploitation fonctionne toujours correctement et qu'au moins un des CPU soit disponible. Cet article[8] entre plus en détails.

[6] *Distributed Control System*
[7] https://etcd.io/
[8] http://www.beekhof.net/blog/2019/savaged-by-softdog

Le principe du *watchdog* peut être résumé par : « nourris le chien de garde avant qu'il ait faim et te mange ». En pratique, un *watchdog* est un compte à rebours avant la réinitialisation brutale du serveur. Si ce compte à rebours n'est pas régulièrement ré-armé, le serveur est alors redémarré.

Un *watchdog* surveille donc passivement un processus et assure que ce dernier est toujours disponible et sain. Dans le cadre d'un cluster en haute disponibilité, le processus rendant compte de sa bonne forme au *watchdog* est le clusterware.

Notez qu'un *watchdog* permet aussi de déclencher un *self-fencing* rapide et fiable en cas de besoin. Il permet par exemple de résoudre rapidement le cas de l'arrêt forcé d'une ressource, déjà présenté dans le chapitre consacré au *fencing*.

Patroni et Pacemaker sont tous deux capables d'utiliser un *watchdog* sur chaque nœud. Pour Patroni, il n'est armé que sur l'instance primaire. Pour Pacemaker, il est armé sur tous les nœuds.

1.5.5 Storage Base Death

- L'une des méthodes historique de *fencing*
- Un ou plusieurs disques partagés
  - où les nœuds s'échangent des messages
- Un *watchdog* par nœud
- Message *poison pill* pour demander à un nœud distant de s'auto-fencer
- *Self-fencing* en cas de perte d'accès aux disques…
  - …si Pacemaker confirme lui aussi une anomalie
- Patroni : émulé

Le *Storage Base Death* est une méthode assez ancienne. Elle utilise un ou plusieurs disques partagés (pour la redondance), montés sur tous les nœuds du cluster à la fois. L'espace nécessaire sur chaque disque est très petit, de l'ordre de quelques mégaoctets pour plusieurs centaines de nœuds (le démon `sbd` utilise 1 à 4 Mo pour 255 nœuds). Cet espace disque est utilisé comme support de communication entre les nœuds qui y échangent des messages.

Le clusterware peut isoler un nœud en déposant un message *poison pill* à son attention. Le destinataire s'auto-fence grâce à son *watchdog* dès qu'il lit le message. De plus, un nœud s'auto-fence aussi s'il n'accède plus au stockage et que Pacemaker ou Corosync indiquent eux aussi une anomalie. Ce comportement défensif permet de s'assurer qu'aucun ordre de *self-fencing* ne peut se perdre.

Grâce au SBD, le cluster est assuré que le nœud distant peut effectuer son *self-fencing* soit par perte de son accès au disque partagé, soit par réception du *poison pill*, soit à cause d'une anomalie qui a empêché le clusterware d'assumer le ré-armement du *watchdog*.

Un exemple détaillé de mise en œuvre avec `sdb` est disponible dans la documentation de Suse[9].

Pacemaker supporte ce type d'architecture. Patroni ne supporte pas SBD mais a un comportement similaire vis-à-vis du DCS. D'une part les nœuds Patroni s'échangent des messages au travers du DCS. De plus, Patroni doit attendre l'expiration du verrou *leader* avant de pouvoir effectuer une bascule, ce qui est similaire au temps de réaction d'une architecture SBD. Mais surtout, l'instance PostgreSQL est déchue en cas de perte de communication avec le DCS, tout le serveur peut même être éteint si le *watchdog* est actif et que l'opération est trop longue.

1.5.6 Bilan des solutions anti-split-brain

À minima, une architecture fiable peut se composer au choix :

- *fencing* actif ou SBD
- 1 *watchdog* par serveur + quorum
- L'idéal : tous les configurer
- Désactiver les services au démarrage

Le *fencing* seul est suffisant pour mettre en œuvre un cluster fiable, même avec deux nœuds. Sans quorum, il est néanmoins nécessaire de désactiver le service au démarrage du cluster, afin d'éviter qu'un nœud isolé ne redémarre ses ressources locales sans l'aval du reste du cluster.

Notez que plusieurs algorithmes existent pour résoudre ce cas, hors quorum, (par exemple les paramètres `two_node`, `wait_for_all` et d'autres de Corosync).

Néanmoins, dans le cadre de PostgreSQL, il n'est jamais très prudent de laisser une ancienne instance primaire au sein d'un cluster sans validation préliminaire de son état. Nous conseillons donc toujours de désactiver le service au démarrage, quelle que soit la configuration du cluster.

L'utilisation d'un SBD est une alternative intéressante et fiable pour la création d'un cluster à deux nœuds sans *fencing* actif. Le stockage y joue un peu le rôle du tiers au sein du cluster pour départager quel nœud conserve les ressources en cas de partition réseau. Le seul défaut de SBD par rapport au *fencing* est le temps d'attente supplémentaire avant de pouvoir considérer que le nœud distant est bien hors service. Attention aussi au stockage partagé sur le même réseau que le cluster. En cas d'incident réseau généralisé, comme chaque machine perd son accès au disque ET aux autres machines via Pacemaker, toutes vont s'éteindre.

Une autre architecture possible est le cumul d'un quorum et du *watchdog*. Avec une telle configuration, en cas de partition réseau, la partition détenant le quorum attend alors la durée théorique du

[9] https://documentation.suse.com/sle-ha/15-SP4/html/SLE-HA-all/cha-ha-storage-protect.html

watchdog (plus une marge) avant de démarrer les ressources perdues. Théoriquement, les nœuds de la partition du cluster perdue sont alors soit redémarrés par leur *watchdog*, soit sains et ont pu arrêter les ressources normalement. Ce type d'architecture nécessite à minima trois nœuds dans le cluster, ou de mettre en place un nœud témoin, utilisé dans le cadre du quorum uniquement (par exemple `corosync QNetd`).

Le cluster idéal cumule les avantages du *fencing*, du quorum et des *watchdogs*.

Comme nous l'avons vu, Pacemaker dispose de toutes les solutions connues. Reste à trouver la bonne combinaison en fonction des contraintes de l'architecture. Patroni, quant à lui, a une architecture similaire au SBD, mais ne force pas à utiliser le *watchdog* sur les nœuds. Pour avoir une architecture aussi fiable que possible, il est recommandé de toujours activer le *watchdog* sur tous les nœuds, au strict minima via `softdog`.

1.6 IMPLICATION ET RISQUES DE LA BASCULE AUTOMATIQUE

- Un collègue peu loquace de plus : un automate
  - et tout doit passer à présent par lui
- Complexification de l'administration
  - Formation + Tests + Documentation + Communication
  - sinon : erreurs humaines plus fréquentes
  - au final : est-ce plus fiable ?
- Opérations post-bascule toujours à faire

L'ajout d'un mécanisme de bascule automatique implique quelques contraintes qu'il est important de prendre en compte lors de la prise de décision.

En premier lieu, l'automate chargé d'effectuer la bascule automatique a tout pouvoir sur vos instances PostgreSQL. Toute opération concernant vos instances de près ou de loin **doit** passer par lui. Il est vital que toutes les équipes soient informées de sa présence afin que toute intervention pouvant impacter le service en tienne compte (mise à jour SAN, coupure réseau, mise à jour applicative, etc).

Ensuite, il est essentiel de construire une architecture aussi simple que possible.

La complexification multiplie les chances de défaillance ou d'erreur humaine. Il est fréquent d'observer plus d'erreurs humaines sur un cluster complexe que sur une architecture sans bascule automatique !

Pour pallier ces erreurs humaines, la formation d'une équipe est vitale. La connaissance concernant le cluster doit être partagée par plusieurs personnes afin de toujours être en capacité d'agir en cas d'incident. Notez que même si la bascule automatique fonctionne convenablement, il est fréquent de devoir intervenir dessus dans un second temps afin de revenir à un état nominal (en reconstruisant un nœud, par exemple).

À ce propos, la documentation de l'ensemble des procédures est essentielle. En cas de maintenance planifiée ou d'incident, il faut être capable de réagir vite avec le moins d'improvisation possible. Quelle que soit la solution choisie, assurez-vous d'allouer suffisamment de temps au projet pour expérimenter, tester le cluster et le documenter.

1.7 RECONSTRUCTION AUTOMATIQUE DES INSTANCES

- mécanisme supporté par Patroni
- risque des:
  - perte d'informations
  - sur-incident
- préférer une reconstruction manuelle

Patroni dispose de plusieurs mécanismes pour reconstruire les instances de manière automatique :

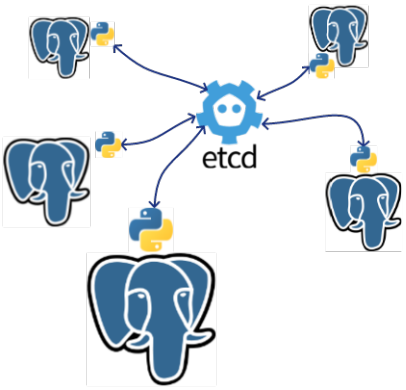
- `pg_rewind` : qui permet de ramener une instance ayant divergé à un point précédent la promotion.
- la suppression automatique du répertoire de données de l'instance quand le `pg_rewind` échoue ou qu'une instance reste bloquée sur une ancienne *timeline*. Cette suppression va provoquer une réinitialisation de l'instance. Elle empêche l'utilisation d'une restauration par delta, ce qui peut provoquer une longue indisponibilité sur des volumétries importantes.

La perspective d'un mécanisme entièrement automatisé qui bascule et reconstruit les instances automatiquement peut être séduisante. Cependant, nous ne préconisons pas l'automatisation de la remise en état du cluster après un *failover* ayant provoqué une divergence. En effet, ce genre de pratique risque de supprimer des éléments nécessaires à l'analyse du problème et peut empêcher de capitaliser sur les leçons de l'incident. De plus, en cas de problème, cela peut donner lieu à un sur-incident.

1.8 QUESTIONS

N’hésitez pas, c’est le moment !

2/ Patroni : Architecture



2.1 AU MENU

- Patroni
- Réplication PostgreSQL
- DCS

Ce module est une introduction à Patroni. Il permet de découvrir quels sont ses pré-requis, les différents éléments nécessaires, leur rôle et leurs interactions.

Les prochains chapitres abordent Patroni lui-même, son rapport avec la réplication PostgreSQL et sa dépendance à un DCS.

2.2 PATRONI

- Un démon `patroni` par instance PostgreSQL
- Les démons patroni coopèrent entre eux
- Chaque démon administre son instance PostgreSQL locale
  - et impose de passer par lui

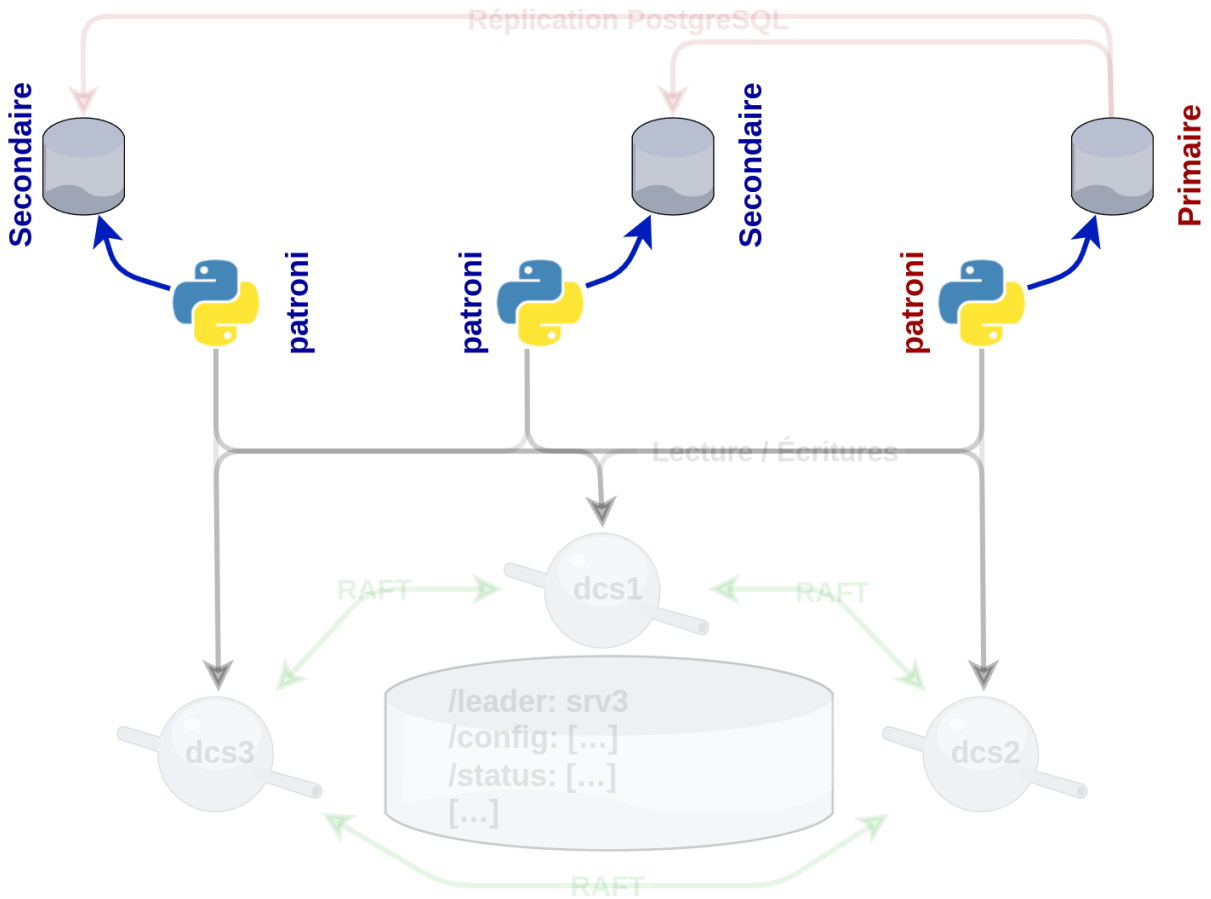


Figure 2/ .1: Étape 1 architecture Patroni

Patroni est un gestionnaire de cluster PostgreSQL, capable d'en assurer la haute disponibilité de service. Il maintient l'agrégat en condition opérationnelle, le supervise et provoque une bascule automatique en cas d'incident.

Patroni est en réalité lui-même un agrégat : chaque instance PostgreSQL du cluster est contrôlée par son propre démon `patroni`, écrit en python. Tous ces démons surveillent les évènements au sein de l'agrégat et coopèrent pour maintenir le service disponible.

Chaque démon `patroni` a la responsabilité de créer, configurer, démarrer, superviser, arrêter, promouvoir ou rétrograder son instance locale, en fonction des évènements détectés au sein du cluster.

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).

2.3 RÉPLICATION POSTGRESQL

Patroni configure la réplication physique entre les instances pour :

- assurer la réplication des données
  - en mode synchrone et/ou asynchrone
  - aussi en cascade
- maintenir la réplication après bascule

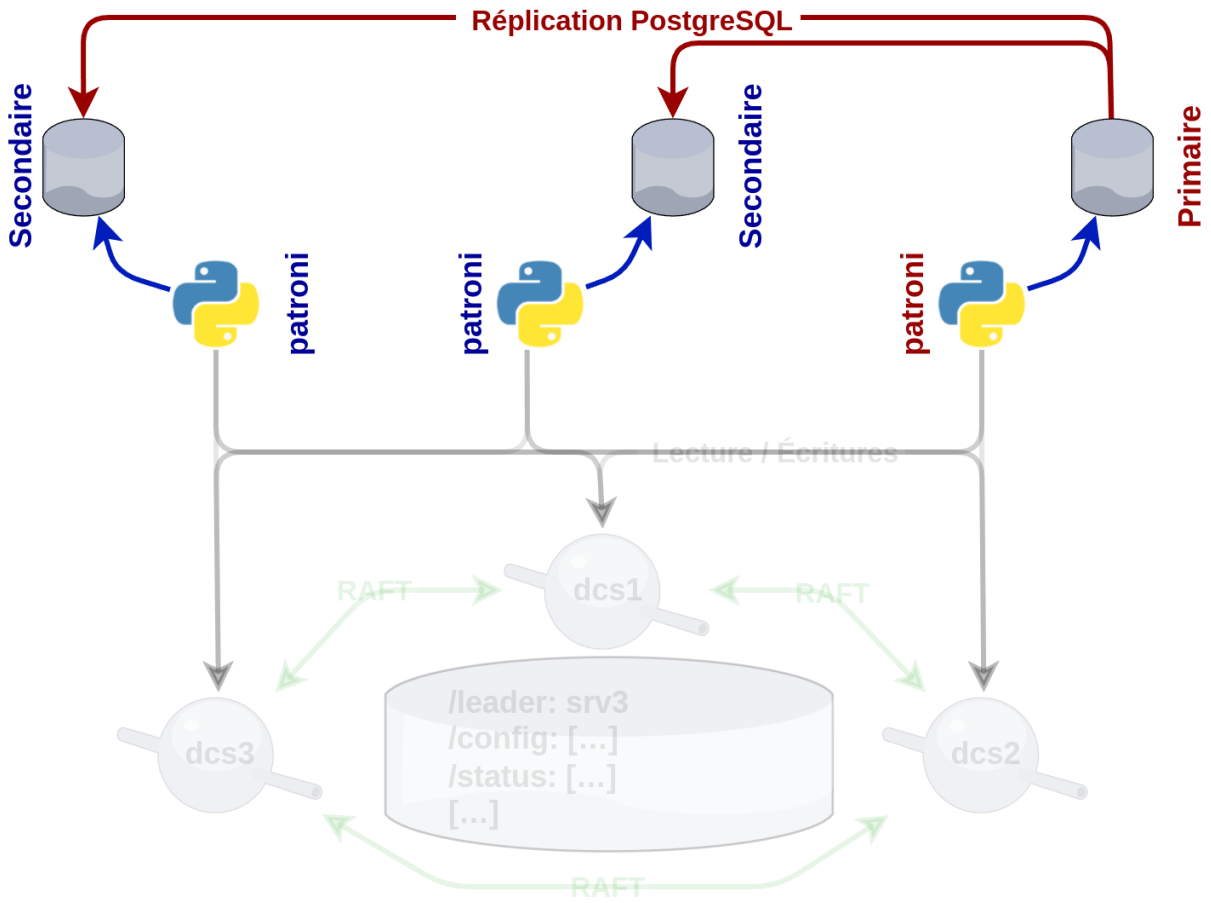


Figure 2/ .2: Étape 2 architecture Patroni

Patroni se base sur la réplication physique native de PostgreSQL pour assurer la haute disponibilité des données. Maîtrisant la création et configuration des instances, il est capable d'assurer automatiquement la mise en œuvre de cette réplication.

Patroni configure et maintient cette réplication physique en mode synchrone ou asynchrone, avec ou sans cascade de réplication, en fonction de la configuration demandée.

Suite à une bascule, il reconfigure automatiquement les secondaires afin que ceux-ci se reconnectent au nouveau primaire.

Optionnellement, il est aussi capable de resynchroniser un ancien primaire en tant que secondaire suite à un incident.

Pour effectuer ces différentes opérations, il s'appuie sur des outils fournis avec PostgreSQL comme `pg_basebackup`[1] et `pg_rewind`[2] pour construire ou reconstruire une instance secondaire. Mais cette capacité est encore étendue grâce à la possibilité d'utiliser des outils de la communauté comme barman[3], pgbackrest[4] ou wal-g[5].

[1] https://docs.postgresql.fr/current/app-pgbasebackup.html
[2] https://docs.postgresql.fr/current/app-pgrewind.html
[3] https://www.pgbarman.org/
[4] https://pgbackrest.org/
[5] https://github.com/wal-g/wal-g

2.4 DCS

- Stockage distribué par consensus
- écritures distribuées et atomiques
- source de vérité de l'agrégat
- hautement disponible

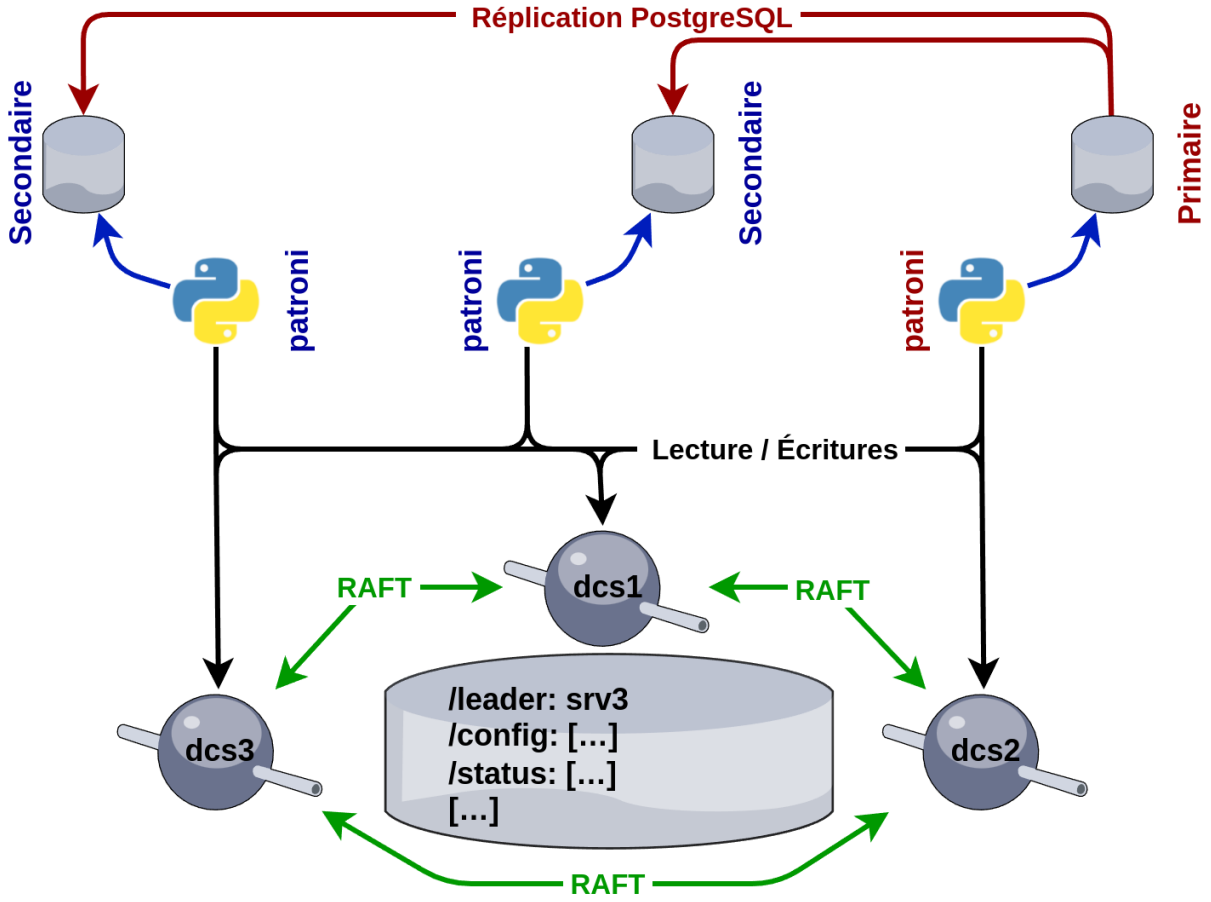


Figure 2/ .3: Étape 3 architecture Patroni

Les démons patroni s'appuient sur un stockage de données distribué (DCS, *Distributed Consensus Store*[6]) pour partager l'état des nœuds et leur configuration au sein du cluster.

[6] ou parfois *Distributed Configuration Store*

Par exemple les processus Patroni y écrivent lequel d'entre eux est le *leader*, pouvant héberger l'instance primaire PostgreSQL. Ils reposent sur les fonctionnalités du DCS qui assurent des écritures atomiques et homogènes au sein du cluster, empêchant toute *race condition* et donc que les démons patroni ne voient des valeurs différentes.

Le DCS est la « source de vérité » du cluster, le notaire arbitrant de façon fiable et officielle le *leader* du cluster. Les processus patroni lui font entièrement confiance et respectent scrupuleusement les informations qu'ils y trouvent.

À tel point que par défaut, si le leader Patroni ne peut plus joindre le DCS, il rétrograde immédiatement son instance locale en secondaire. Si cette opération échoue ou est trop longue, il est même capable de déclencher un reset du serveur.

Le but étant la haute disponibilité, le DCS 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.

Patroni supporte plusieurs DCS différents, s'adaptant ainsi au mieux à l'environnement dans lequel il est déployé. En voici une liste à titre indicatif : etcd (v2 ou v3), Consul, Zookeeper, Exhibitor et Kubernetes.

2.5 DEUX GRAPPES DE SERVEURS

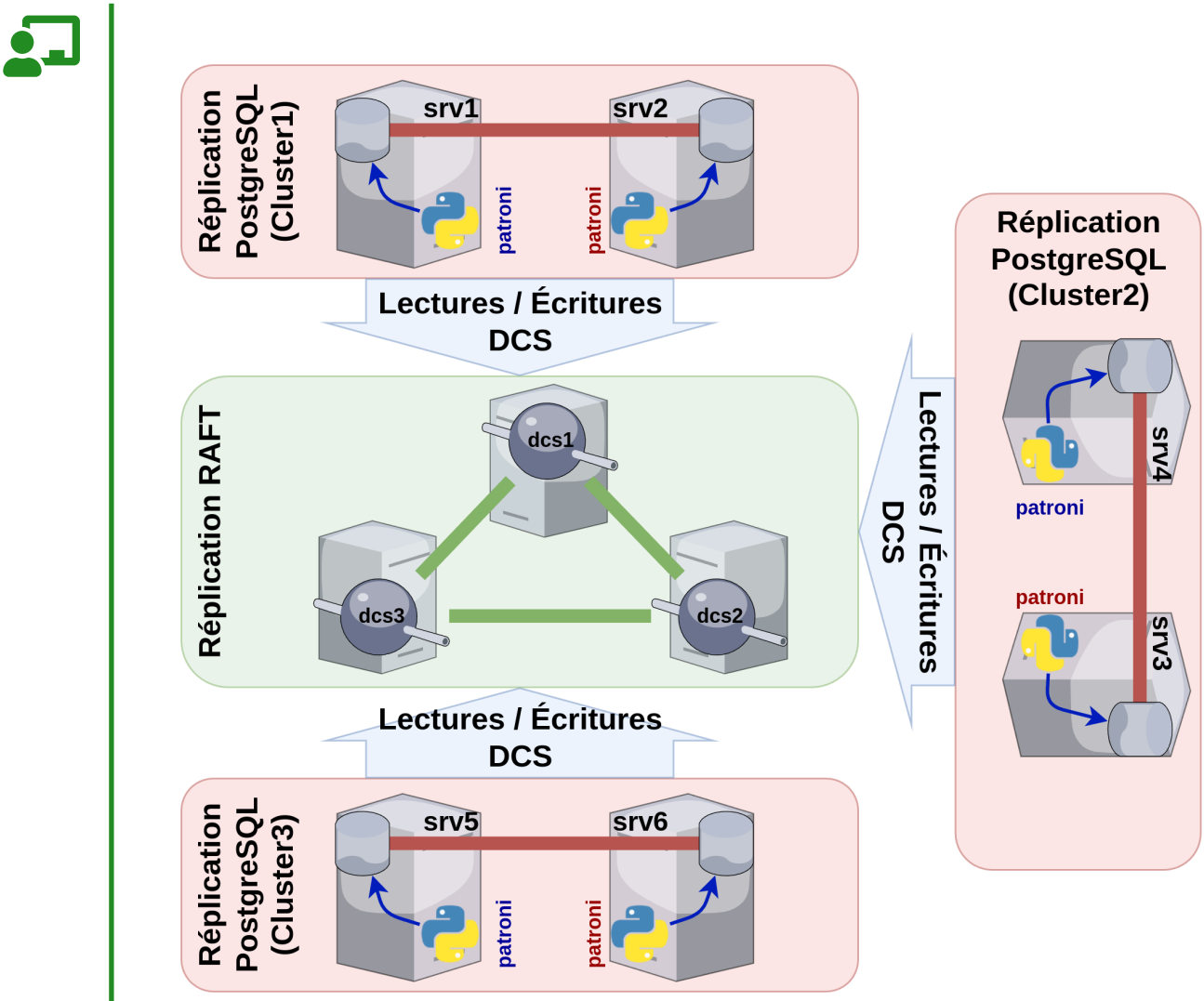


Figure 2/ .4: Agrégats de serveurs pour Patroni

Ce schéma présente les deux types d'agrégats :

- un unique agrégat de trois serveurs minimum héberge les démons DCS ;
- plusieurs agrégats PostgreSQL indépendants hébergent les démons patroni et leur instance PostgreSQL respective (2 instances minimum).

Un cluster Patroni/PostgreSQL est client du cluster DCS. Un même cluster DCS peut gérer plusieurs clusters Patroni/PostgreSQL différents.

Ces clusters ont des outils, procédures et maintenances différentes qu'il faut comprendre en détail, documenter et superviser. Différents modules de formation abordent chacun des services de cette architecture :

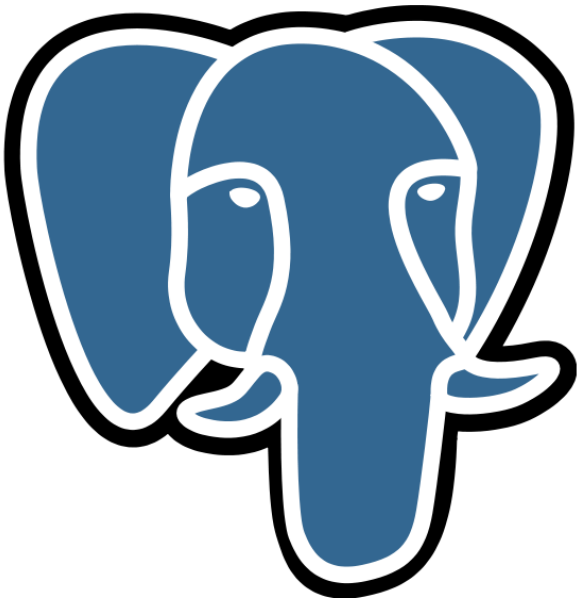
- un module de rappel concernant la réplication de PostgreSQL[7] ;
- un module dédié au DCS etcd[8] ;
- un module dédié à Patroni[9] lui-même.

[7] https://dali.bo/r56_html
[8] https://dali.bo/r57_html
[9] https://dali.bo/r58_html

2.6 QUESTIONS

- C'est le moment !

3/ Réplication physique : rappels



3.1 INTRODUCTION

- Patroni repose sur la réplication physique de PostgreSQL
- la haute disponibilité implique les équipes techniques
- les équipes techniques doivent comprendre la mécanique

Patroni configure la réplication physique native de PostgreSQL pour assurer la redondance des données au sein de l'agrégat. Il est important de bien maîtriser comment cette réplication fonctionne.

Effectivement, Patroni ayant pour but d'optimiser la disponibilité, l'administrateur doit être lui-même réactif et identifier rapidement les causes d'un incident ou d'un problème de réplication, savoir réintégrer un nœud en réplication, etc. Et ce, sans créer de sur-incident bien entendu.

3.1.1 Au menu

- Mise en place de la réplication physique
- Promotion
- Retour à l'état stable

3.2 RÉPLICATION PAR STREAMING

3.2.1 Mise en place de la réplication par streaming

- Réplication en flux
- Un processus du serveur primaire discute avec un processus du serveur secondaire
  - d'où un *lag* moins important
- Asynchrone ou synchrone
- En cascade

Le serveur PostgreSQL secondaire lance un processus appelé `walreceiver`, dont le but est de se connecter au serveur primaire et d'attendre les modifications de la réplication.

Le `walreceiver` a donc besoin de se connecter sur le serveur PostgreSQL primaire. Ce dernier doit être configuré pour accepter cette connexion. Quand elle est acceptée par le serveur primaire, le serveur PostgreSQL du serveur primaire lance un nouveau processus, appelé `walsender`. Ce dernier a pour but d'envoyer les données de réplication au serveur secondaire. Les données de réplication sont envoyées suivant l'activité et certains paramètres de configuration.

Cette méthode permet une réplication plus proche du serveur primaire que le *log shipping*. On peut même configurer un mode synchrone : un client du serveur primaire ne récupère pas la main tant que ses modifications ne sont pas enregistrées sur le serveur primaire **et** sur le serveur secondaire synchrone. Cela s'effectue à la validation de la transaction, implicite ou lors d'un `COMMIT`.

Enfin, la réplication en cascade permet à un secondaire de fournir les informations de réplication à un autre secondaire, déchargeant ainsi le serveur primaire d'un certain travail et diminuant aussi la bande passante réseau utilisée par le serveur primaire.

3.2.2 Serveur primaire (1/2) - Configuration

Dans `postgresql.conf` :

- `wal_level = replica` (ou `logical`)
- `max_wal_senders = X`
  - 1 par client par *streaming*
  - défaut : 10
- `wal_sender_timeout = 60s`

Il faut tout d'abord s'assurer que PostgreSQL enregistre suffisamment d'informations pour que le serveur secondaire puisse rejouer toutes les modifications survenant sur le serveur primaire. Dans certains cas, PostgreSQL peut économiser l'écriture de journaux quand cela ne pose pas de problème pour l'intégrité des données en cas de crash. Par exemple, sur une instance sans archivage ni réplication, il est inutile de tracer la totalité d'une transaction qui commence par créer une table, puis qui la remplit. En cas de crash pendant l'opération, l'opération complète est annulée, la table n'existera plus : PostgreSQL peut donc écrire directement son contenu sur le disque sans journaliser.

Cependant, pour restaurer cette table ou la répliquer, il est nécessaire d'avoir les étapes intermédiaires (le contenu de la table) et il faut donc écrire ces informations supplémentaires dans les journaux.

Le paramètre `wal_level` fixe le comportement à adopter. Comme son nom l'indique, il permet de préciser le niveau d'informations que l'on souhaite avoir dans les journaux. Il connaît trois valeurs :

- Le niveau `replica` est adapté à l'archivage ou la réplication, en plus de la sécurisation contre les arrêts brutaux. C'est le niveau par défaut. L'optimisation évoquée plus haut n'est pas possible.
- Le niveau `minimal` n'offre que la protection contre les arrêts brutaux, mais ne permet ni réplication ni sauvegarde PITR. Ce niveau ne sert plus guère qu'aux environnements ni archivés, ni répliqués, pour réduire la quantité de journaux générés, comme dans l'optimisation ci-dessus.
- Le niveau `logical` est le plus complet et doit être activé pour l'utilisation du décodage logique, notamment pour utiliser la réplication logique. Il n'est pas nécessaire pour la sauvegarde PITR ou la réplication physique, ni incompatible.

Le serveur primaire accepte un nombre maximum de connexions de réplication : il s'agit du paramètre `max_wal_senders`. Il faut compter au moins une connexion pour chaque serveur secondaire susceptible de se connecter, ou les outils utilisant le *streaming* comme `pg_basebackup` ou `pg_receivewal`. Il est conseillé de prévoir « large » d'entrée : l'impact mémoire est négligeable, et cela évite d'avoir à redémarrer l'instance primaire à chaque modification. La valeur par défaut de 10 devrait suffire dans la plupart des cas.

Le paramètre `wal_sender_timeout` permet de couper toute connexion inactive après le délai indi-

qué par ce paramètre. Par défaut, le délai est d'une minute. Cela permet au serveur primaire de ne pas conserver une connexion coupée ou dont le client a disparu pour une raison ou une autre. Le secondaire retentera par la suite une connexion complète.

3.2.3 Serveur primaire (2/2) - Authentification

- Le serveur secondaire doit pouvoir se connecter au serveur primaire
- Pseudo-base `replication`
- Utilisateur dédié conseillé avec attributs `LOGIN` et `REPLICATION`
- Configurer `pg_hba.conf` :

```
host replication user_repli 10.2.3.4/32   scram-sha-256
```

- Recharger la configuration

Il est nécessaire après cela de configurer le fichier `pg_hba.conf`. Dans ce fichier, une ligne (par secondaire) doit indiquer les connexions de réplication. L'idée est d'éviter que tout le monde puisse se connecter pour répliquer l'intégralité des données.

Pour distinguer une ligne de connexion standard et une ligne de connexion de réplication, la colonne indiquant la base de données doit contenir le mot « replication ». Par exemple :

```
host    replication    user_repli    10.0.0.2/32    scram-sha-256
```

Dans ce cas, l'utilisateur `user_repli` pourra entamer une connexion de réplication vers le serveur primaire à condition que la demande de connexion provienne de l'adresse IP `10.0.0.2` et que cette demande de connexion précise le bon mot de passe au format `scram-sha-256`.

Un utilisateur dédié à la réplication est conseillé pour des raisons de sécurité. On le créera avec les droits suivants :

```
CREATE ROLE user_repli LOGIN REPLICATION ;
```

et bien sûr un mot de passe complexe.

Les connexions locales de réplication sont autorisées par défaut sans mot de passe.

Après modification du fichier `postgresql.conf` et du fichier `pg_hba.conf`, il est temps de demander à PostgreSQL de recharger sa configuration. L'action `reload` suffit dans tous les cas, sauf celui où `max_wal_senders` est modifié (auquel cas il faudra redémarrer PostgreSQL).

3.2.4 Serveur secondaire (1/4) - Copie des données

Copie des données du serveur primaire (à chaud !) :

- Copie généralement à chaud donc incohérente !
- Le plus simple : `pg_basebackup`
  - simple mais a des limites
- Idéal : outil PITR
- Possible : `rsync`, `cp`…
  - ne pas oublier `pg_backup_start()` / `pg_backup_stop()` !
  - exclure certains répertoires et fichiers
  - garantir la disponibilité des journaux de transaction

La première action à réaliser ressemble beaucoup à ce que propose la sauvegarde en ligne des fichiers. Il s'agit de copier le répertoire des données de PostgreSQL ainsi que les tablespaces associés.

Rappelons que généralement cette copie aura lieu à chaud, donc une simple copie directe sera incohérente.

pg_basebackup :

L'outil le plus simple est `pg_basebackup`. Ses avantages sont sa disponibilité et sa facilité d'utilisation. Il sait ce qu'il n'y a pas besoin de copier et peut inclure les journaux nécessaires pour ne pas avoir à paramétrer l'archivage.

Il peut utiliser la connexion de réplication déjà prévue pour le secondaire, poser des slots temporaires ou le slot définitif.

Pour faciliter la mise en place d'un secondaire, il peut générer les fichiers de configuration à partir des paramètres qui lui ont été fournis (option `--write-recovery-conf`).

Malgré beaucoup d'améliorations dans les dernières versions, la limite principale de `pg_basebackup` reste d'exiger un répertoire cible vide : on doit toujours recopier l'intégralité de la base copiée. Cela peut être pénible lors de tests répétés avec une grosse base, ou avec une liaison instable. Toutefois, à partir de PostgreSQL 17, il permet une sauvegarde incrémentale.

Outils PITR :

L'idéal est un outil de restauration PITR permettant la restauration en mode delta, par exemple pgBackRest avec l'option `--delta`. Ne sont restaurés que les fichiers ayant changé, et le primaire n'est pas chargé par la copie.

rsync :

Un script de copie reste une option possible. Il est possible de le faire manuellement, tout comme pour une sauvegarde PITR.

Une copie manuelle implique que les journaux sont archivés par ailleurs.

Rappelons les trois étapes essentielles :

- le `pg_backup_start()` ;
- la copie des fichiers : généralement avec `rsync --whole-file`, ou tout moyen permettant une copie fiable et rapide ;
- le `pg_backup_stop()`.

On exclura les fichiers inutiles lors de la copie qui pourraient gêner un redémarrage, notamment les fichiers `postmaster.pid`, `postmaster.opts`, `pg_internal.init`, les répertoires `pg_wal`, `pg_replslot`, `pg_dynshmem`, `pg_notify`, `pg_serial`, `pg_snapshots`, `pg_stat_tmp`, `pg_subtrans`, `pgslq_tmp*`. La liste complète figure dans la documentation officielle[1].

3.2.5 Serveur secondaire (2/4) - Fichiers de configuration

- `postgresql.conf` & `postgresql.auto.conf`
  - paramètres
- `standby.signal` (dans PGDATA)
  - vide

Au choix, les paramètres sont à ajouter dans `postgresql.conf`, dans un fichier appelé par ce dernier avec une clause d'inclusion, ou dans `postgresql.auto.conf` (forcément dans le répertoire de données pour ce dernier, et qui surcharge les fichiers précédents). Cela dépend des habitudes, de la méthode d'industrialisation…

S'il y a des paramètres propres au primaire dans la configuration d'un secondaire, ils seront ignorés, et vice-versa. Dans les cas simples, le `postgresql.conf` peut donc être le même.

Puis il faut créer un fichier vide nommé `standby.signal` dans le répertoire `PGDATA`, qui indique à PostgreSQL que le serveur doit rester en *recovery* permanent.

[1] https://docs.postgresql.fr/current/continuous-archiving.html#BACKUP-LOWLEVEL-BASE-BACKUP

Au cas où vous rencontreriez un vieux serveur en version antérieure à la 12 : jusqu'en version 11, on activait le mode *standby* non dans la configuration, mais en créant un fichier texte `recovery.conf` dans le `PGDATA` de l'instance, et en y plaçant le paramètre `standby_mode` à `on`. Les autres paramètres sont les mêmes. Toute modification impliquait un redémarrage.

3.2.6 Serveur secondaire (3/4) - Paramètres

- `primary_conninfo` (*streaming*) :

```
primary_conninfo = 'user=user_repli host=prod port=5434
 application_name=standby '
```

- Optionnel :
  - `primary_slot_name`
  - `restore_command`
  - `wal_receiver_timeout`

PostgreSQL doit aussi savoir comment se connecter au serveur primaire. C'est le paramètre `primary_conninfo` qui le lui dit. Il s'agit d'un DSN standard où il est possible de spécifier l'adresse IP de l'hôte ou son alias, le numéro de port, le nom de l'utilisateur, etc. Il est aussi possible de spécifier le mot de passe, mais c'est risqué en terme de sécurité. En effet, PostgreSQL ne vérifie pas si ce fichier est lisible par quelqu'un d'autre que lui. Il est donc préférable de placer le mot de passe dans le fichier `.pgpass`, généralement dans `~postgres/` sur le secondaire, fichier qui n'est utilisé que s'il n'est lisible que par son propriétaire. Par exemple :

```
primary_conninfo = 'user=postgres host=prod passfile=/var/lib/postgresql/.pgpass'
```

Toutes les options de la libpq sont accessibles. Par exemple, cette chaîne de connexion a été générée pour un nouveau secondaire par `pg_basebackup -R` :

```
primary_conninfo = 'host=prod user=postgres passfile=''/var/lib/postgresql/.pgpass''
 ↪  channel_binding=prefer port=5436 sslmode=prefer sslcompression=0
 ↪  sslcertmode=allow sslsni=1 ssl_min_protocol_version=TLSv1.2 gssencmode=prefer
 ↪  krbsrvname=postgres gssdelegation=0 target_session_attrs=any
 ↪  load_balance_hosts=disable
```

S'y trouvent beaucoup de paramétrage par défaut dépendant de méthodes d'authentification, ou pour le SSL.

Parmi les autres paramètres optionnels de `primary_conninfo`, il est conseillé d'ajouter `application_name`, par exemple avec le nom du serveur. Cela facilite la supervision. C'est même nécessaire pour paramétrer une réplication synchrone.

```
primary_conninfo = 'user=postgres host=prod passfile=/var/lib/postgresql/.pgpass
↪  application_name=secondaire2 '
```

Si `application_name` n'est pas fourni, le `cluster_name` du secondaire sera utilisé, mais il est rarement correctement configuré (par défaut, il vaut `16/main` sur Debian/Ubuntu, et n'est pas configuré sur Red Hat/Rocky Linux).

De manière optionnelle, nous verrons que l'on peut définir aussi deux paramètres :

- `primary_slot_name`, pour sécuriser la réplication avec un slot de réplication ;
- `restore_command`, pour sécuriser la réplication avec un accès à la sauvegarde PITR.

Le paramètre `wal_receiver_timeout` sur le secondaire est le symétrique de `wal_sender_timeout` sur le primaire. Il indique au bout de combien de temps couper une connexion inactive. Le secondaire retentera la connexion plus tard.

3.2.7 Serveur secondaire (4/4) - Démarrage

- Démarrer PostgreSQL
- Suivre dans les traces que tout va bien

Il ne reste plus qu'à démarrer le serveur secondaire.

En cas de problème, le premier endroit où aller chercher est bien entendu le fichier de trace `postgresql.log`.

3.2.8 Processus

Sur le primaire :

- `walsender ... streaming 0/3BD48728`

Sur le secondaire :

- `walreceiver   streaming 0/3BD48728`

Sur le primaire, un processus `walsender` apparaît pour chaque secondaire connecté. Son nom de processus est mis à jour en permanence avec l'emplacement dans le flux de journaux de transactions :

```
postgres: 16/secondaire1: walsender postgres [local] streaming 15/6A6EF408
postgres: 16/secondaire2: walsender postgres [local] streaming 15/6A6EF408
```

Symétriquement, sur chaque secondaire, un process `walreceiver` apparaît.

```
postgres: 16/secondaire2: walreceiver streaming 0/DD73C218
```

3.3 PROMOTION

3.3.1 Au menu

- Attention au *split-brain* !
- Vérification avant promotion
- Promotion : méthode et déroulement
- Retour à l'état stable

3.3.2 Attention au split-brain !

- Si un serveur secondaire devient le nouveau primaire
  - s'assurer que l'ancien primaire ne reçoit plus d'écriture
- Éviter que les deux instances soient ouvertes aux écritures
  - confusion et perte de données !

La pire chose qui puisse arriver lors d'une bascule est d'avoir les deux serveurs, ancien primaire et nouveau primaire promu, ouverts tous les deux en écriture. Les applications risquent alors d'écrire dans l'un ou l'autre…

Quelques histoires « d'horreur » à ce sujet :

- de nombreux exemples sur diverses technologies de réplication[2] ;
- *post mortem* d'un gros problème chez Github en 2018[3].

[2] https://github.blog/2018-10-30-oct21-post-incident-analysis
[3] https://aphyr.com/posts/288-the-network-is-reliable

3.3.3 Vérification avant promotion

- Primaire :

```
# systemctl stop postgresql-17

$ pg_controldata -D /var/lib/pgsql/17/data/ \
| grep -E '(Database cluster state)|(REDO location)'
Database cluster state:               shut down
Latest checkpoint's REDO location:    0/3BD487D0
```

- Secondaire :

```
$ psql -c 'CHECKPOINT;'
$ pg_controldata -D /var/lib/pgsql/17/data/ \
| grep -E '(Database cluster state)|(REDO location)'
Database cluster state:               in archive recovery
Latest checkpoint's REDO location:    0/3BD487D0
```

Avant une bascule, il est capital de vérifier que toutes les modifications envoyées par le primaire sont arrivées sur le secondaire. Si le primaire a été arrêté proprement, ce sera le cas. Après un `CHECKPOINT` sur le secondaire, on y retrouvera le même emplacement dans les journaux de transaction.

Ce contrôle doit être systématique avant une bascule. Même si toutes les écritures applicatives sont stoppées sur le primaire, quelques opérations de maintenance peuvent en effet écrire dans les journaux et provoquer un écart entre les deux serveurs (divergence). Il n'y aura alors pas de perte de données mais cela pourrait gêner la transformation de l'ancien primaire en secondaire, par exemple. En revanche, **même avec un arrêt propre du primaire**, il peut y avoir perte de données s'il y a un *lag* important entre primaire et secondaire : même si le rejeu va toujours jusqu'au bout avant le changement de *timeline*, les WAL qui n'ont pas pu être récupérés avant la déconnexion, ou après la récupération des archives (si le *log shipping* est en place) sont perdus pour le nouveau primaire.

Noter que `pg_controldata` n'est pas dans les chemins par défaut des distributions. La fonction SQL `pg_control_checkpoint()` affiche les même informations, mais n'est bien sûr pas accessible sur un primaire arrêté.

3.3.4 Promotion du standby : méthode

- Shell :
  - `pg_ctl promote`
- SQL :
  - fonction `pg_promote()`

Il existe plusieurs méthodes pour promouvoir un serveur PostgreSQL en mode *standby*. Les méthodes les plus appropriées sont :

- l'action `promote` de l'outil `pg_ctl`, ou de son équivalent dans les scripts des paquets d'installation, comme `pg_ctlcluster` sous Debian ;
- la fonction SQL `pg_promote`.

Ces deux méthodes sont à préférer à la méthode historique du fichier de déclenchement (*trigger file*), qui existe encore jusque PostgreSQL 15 inclus. Le paramètre `promote_trigger_file` sur une instance secondaire définit un fichier dont l'apparition provoque la promotion de l'instance. Par sécurité, utiliser un emplacement accessible uniquement aux administrateurs.

3.3.5 Promotion du standby : déroulement

Une promotion déclenche :

- déconnexion de la *streaming replication* (bascule programmée)
- rejeu des dernières transactions en attente d'application
- récupération et rejeu de toutes les archives disponibles
- choix d'une nouvelle *timeline* du journal de transaction
- suppression du fichier `standby.signal`
- nouvelle *timeline* et fichier `.history`
- ouverture aux écritures

La promotion se déroule en bonne partie comme un *recovery* après restauration PITR.

Une fois l'instance promue, elle finit de rejouer les derniers journaux de transaction en provenance du serveur principal en sa possession, puis se déconnecte de celui-ci (si l'on est encore connecté en

streaming). Après la déconnexion, si une `restore_command` est configurée, toutes les archives disponibles sont récupérées et rejouées (en général, il n'y a pas d'archive contenant des WAL plus récents que le dernier récupéré en *streaming* ; mais des écritures lourdes et/ou un réseau trop lent peuvent entraîner un retard du *streaming*).

Le dernier journal reçu de l'ancien primaire est souvent incomplet. Il est renommé avec le suffixe `.partial` et archivé. Cela évite un conflit de nom éventuel avec le même fichier issu de l'ancien serveur, qui a pu aussi être archivé, à un point éventuellement postérieur à la divergence.

Ensuite, l'instance choisit une nouvelle *timeline* pour son journal de transactions. Rappelons que la *timeline* est le premier numéro dans le nom du segment (fichier WAL) ; par exemple une timeline 5 pour un fichier nommé `000000050000003200000031`. Le nouveau primaire choisit généralement le numéro suivant celui du primaire (à moins que les archives ne contiennent d'autres *timelines* de numéro supérieur, s'il y a eu plusieurs restaurations et retours en arrière, et il choisit alors le numéro suivant la dernière).

Le choix d'une nouvelle *timeline* permet à PostgreSQL de rendre les journaux de transactions de ce nouveau serveur en écriture incompatibles avec son ancien serveur principal. De plus, des journaux de nom différent permet l'archivage depuis ce primaire sans perturber l'ancien s'il existe encore. Il n'y a plus de fichier en commun même si l'espace d'archivage est partagé.

Les *timelines* ne changent pas que lors des promotions, mais aussi lors des restaurations PITR. En général, on désire que les secondaires (parfois en cascade) suivent. Heureusement, ceci est le paramétrage par défaut depuis la version 12 :

```
recovery_target_timeline = latest
```

Un secondaire suit donc par défaut les évolutions de *timeline* de son primaire, tant que celui-ci n'effectue pas de retour en arrière.

L'instance crée un fichier d'historique dans `pg_wal/`, par exemple `00000006.history` pour la nouvelle *timeline* 6. C'est un petit fichier texte qui contient les différentes *timelines* ayant mené à la nouvelle. Ce fichier est immédiatement archivé s'il y a archivage.

Enfin, l'instance autorise les connexions en lecture et en écriture.

3.3.6 Opérations après promotion du standby

- `VACUUM ANALYZE` conseillé
  - calcul d'informations nécessaires pour autovacuum

Il n'y a aucune opération obligatoire après une promotion. Cependant, il est conseillé d'exécuter un `VACUUM` ou un `ANALYZE` pour que PostgreSQL mette à jour les estimations de nombre de lignes vivantes et mortes. Ces estimations sont utilisées par l'autovacuum pour lutter contre la fragmentation des tables et mettre à jour les statistiques sur les données. Or ces estimations faisant partie des statistiques d'activité, elles ne sont pas répliquées vers les secondaires. Il est donc intéressant de les mettre à jour après une promotion.

3.3.7 Retour à l'état stable

Si un *standby* a été momentanément indisponible :

- Rattrapage possible si **tous** les journaux sont disponibles :
  - *streaming* depuis le primaire (slot, `wal_keep_size`)
  - *log shipping* depuis les archives (`restore_command`)
  - ou les deux (si configurés)
- Sinon :
  - « décrochage… »
  - reconstruction nécessaire

Si un serveur secondaire est momentanément indisponible mais revient en ligne sans perte de données (réseau coupé, problème OS…), alors il a de bonnes chances de se « raccrocher » à son serveur primaire. Il faut bien sûr que l'ensemble des journaux de transactions depuis son arrêt soit accessible à ce serveur, **sans exception**.

Si le secondaire ne peut rattraper le flux des journaux du primaire, il peut rester ouvert en lecture, mais il ne rattrapera jamais le primaire. Il est donc irrécupérable et doit être reconstruit par l'une des méthodes précédentes, ce qui peut être long et lourd.

Le secondaire cherche ces journaux par *log shipping* et par *streaming* auprès de son primaire, s'ils sont configurés et fonctionnels. Si une méthode échoue, le secondaire tentera l'autre, alternativement, et indéfiniment jusqu'à obtenir les journaux nécessaires. Si une seule méthode est configurée, le secondaire ne tente bien sûr que celle-ci. Tant qu'une méthode fonctionne, le secondaire l'utilise jusqu'à ce qu'elle tombe en échec.

En cas de réplication par *streaming* :

Le secondaire utilise la `primary_conninfo` pour demander au primaire les journaux à partir de sa propre position dans le flux de journaux, successivement et dans l'ordre. Le primaire ne doit pas avoir

recyclé ou supprimé ces journaux devenus inutiles après leur archivage et un *checkpoint*, ce qui peut ne prendre que quelques minutes.

Si le primaire n'a plus le journal nécessaire, les deux instances afficheront cette erreur :

```
ERROR:  requested WAL segment 000000010000009900000045 has already been removed
```

On peut forcer le primaire à conserver des journaux pour un serveur secondaire :

- La première méthode est simple et consiste à définir un paramètre nommé `wal_keep_size`, pour que le primaire conserve toujours une certaine quantité de journaux, inutiles pour lui, au cas où un secondaire le demande.
- La méthode la plus propre consiste à créer un « slot de réplication » sur le primaire, et dédié à ce secondaire (paramètre `primary_slot` à définir sur celui-ci). Le primaire sait alors en permanence où en est son secondaire, et lui conserve toujours les journaux nécessaires. C'est très sûr, mais peut avoir des inconvénients.

En cas de réplication par *log shipping* :

Il faut que la `restore_command` fonctionne, que le stock des journaux remonte assez loin dans le temps (jusqu'au moment où le secondaire a perdu contact), et qu'aucun journal ne manque ni ne soit corrompu. Souvent, le secondaire utilise le *log shipping* depuis les archives PITR pour rattraper son retard, jusqu'à ce que l'archivage tombe en erreur après épuisement de tous les journaux disponibles ; puis il bascule sur le *streaming* si c'est possible.

3.4 CONCLUSION

- La réplication physique native est très robuste
- Patroni peut automatiser :
  - la mise en réplication
  - la promotion
  - le raccrochage d'un ancien primaire

La robustesse de la réplication physique est éprouvée depuis longtemps. C'est à cette base solide que Patroni amène l'automatisation de :

- l'ajout de nœuds supplémentaires ;
- la promotion automatique en fonction du nœud ;
- le raccrochage d'un ancien primaire au nouveau primaire.

Ces mécanismes sont abordé dans le module consacré à Patroni[4].

[4] https://dali.bo/r58_html

3.5 INSTALLATION DE POSTGRESQL DEPUIS LES PAQUETS COMMUNAUTAIRES

L'installation est détaillée ici pour Rocky Linux 8 et 9 (similaire à Red Hat et à d'autres variantes comem Oracle Linux et Fedora), et Debian/Ubuntu.

Elle ne dure que quelques minutes.

3.5.1 Sur Rocky Linux 8 ou 9

ATTENTION : Red Hat, CentOS, Rocky Linux fournissent souvent par défaut des versions de PostgreSQL qui ne sont plus supportées. Ne jamais installer les packages `postgresql`, `postgresql-client` et `postgresql-server` ! L'utilisation des dépôts du PGDG est fortement conseillée.

Installation du dépôt communautaire :

Les dépôts de la communauté sont sur https://yum.postgresql.org/. Les commandes qui suivent sont inspirées de celles générées par l'assistant sur https://www.postgresql.org/download/linux/redhat/, en précisant :

- la version majeure de PostgreSQL (ici la 17) ;
- la distribution (ici Rocky Linux 8) ;
- l'architecture (ici x86_64, la plus courante).

Les commandes sont à lancer sous **root** :

```
# dnf install -y https://download.postgresql.org/pub/repos/yum/reporpms\
/EL-8-x86_64/pgdg-redhat-repo-latest.noarch.rpm

# dnf -qy module disable postgresql
```

Installation de PostgreSQL 17 (client, serveur, librairies, extensions) :

```
# dnf install -y postgresql17-server postgresql17-contrib
```

Les outils clients et les librairies nécessaires seront automatiquement installés.

Une fonctionnalité avancée optionnelle, le JIT (*Just In Time compilation*), nécessite un paquet séparé.

```
# dnf install postgresql17-llvmjit
```

Création d'une première instance :

Il est conseillé de déclarer `PG_SETUP_INITDB_OPTIONS`, notamment pour mettre en place les sommes de contrôle et forcer les traces en anglais :

```
# export PGSETUP_INITDB_OPTIONS='--data-checksums --lc-messages=C'
# /usr/pgsql-17/bin/postgresql-17-setup initdb
# cat /var/lib/pgsql/17/initdb.log
```

Ce dernier fichier permet de vérifier que tout s'est bien passé et doit finir par :

```
Success. You can now start the database server using:

    /usr/pgsql-17/bin/pg_ctl -D /var/lib/pgsql/17/data/ -l logfile start
```

Chemins :

Objet	Chemin
Binaires	`/usr/pgsql-17/bin`
Répertoire de l'utilisateur **postgres**	`/var/lib/pgsql`
`PGDATA` par défaut	`/var/lib/pgsql/17/data`
Fichiers de configuration	dans `PGDATA/`
Traces	dans `PGDATA/log`

Configuration :

Modifier `postgresql.conf` est facultatif pour un premier lancement.

Commandes d'administration habituelles :

Démarrage, arrêt, statut, rechargement à chaud de la configuration, redémarrage :

```
# systemctl start postgresql-17
# systemctl stop postgresql-17
# systemctl status postgresql-17
# systemctl reload postgresql-17
# systemctl restart postgresql-17
```

Test rapide de bon fonctionnement et connexion à psql :

```
# systemctl --all |grep postgres
# sudo -iu postgres psql
```

Démarrage de l'instance au lancement du système d'exploitation :

```
# systemctl enable postgresql-17
```

Ouverture du *firewall* pour le port 5432 :

Voir si le *firewall* est actif :

```
# systemctl status firewalld
```

Si c'est le cas, autoriser un accès extérieur :

```
# firewall-cmd --zone=public --add-port=5432/tcp --permanent
# firewall-cmd --reload
# firewall-cmd --list-all
```

(Rappelons que `listen_addresses` doit être également modifié dans `postgresql.conf`.)

Création d'autres instances :

Si des instances de *versions majeures différentes* doivent être installées, il faut d'abord installer les binaires pour chacune (adapter le numéro dans `dnf install …`) et appeler le script d'installation de chaque version. l'instance par défaut de chaque version vivra dans un sous-répertoire numéroté de `/var/lib/pgsql` automatiquement créé à l'installation. Il faudra juste modifier les ports dans les `postgresql.conf` pour que les instances puissent tourner simultanément.

Si plusieurs instances d'une *même version majeure* (forcément de la même version mineure) doivent cohabiter sur le même serveur, il faut les installer dans des `PGDATA` différents.

- Ne pas utiliser de tiret dans le nom d'une instance (problèmes potentiels avec systemd).
- Respecter les normes et conventions de l'OS : placer les instances dans un nouveau sous-répertoire de `/var/lib/pgsqsl/17/` (ou l'équivalent pour d'autres versions majeures).

Pour créer une seconde instance, nommée par exemple **infocentre** :

- Création du fichier service de la deuxième instance :

```
# cp /lib/systemd/system/postgresql-17.service \
       /etc/systemd/system/postgresql-17-infocentre.service
```

- Modification de ce dernier fichier avec le nouveau chemin :

```
Environment=PGDATA=/var/lib/pgsql/17/infocentre
```

- Option 1 : création d'une nouvelle instance vierge :

```
# export PGSETUP_INITDB_OPTIONS='--data-checksums --lc-messages=C'
# /usr/pgsql-17/bin/postgresql-17-setup initdb postgresql-17-infocentre
```

- Option 2 : restauration d'une sauvegarde : la procédure dépend de votre outil.
- Adaptation de `/var/lib/pgsql/17/infocentre/postgresql.conf` (port surtout).
- Commandes de maintenance de cette instance :

```
# systemctl [start|stop|reload|status] postgresql-17-infocentre
# systemctl [enable|disable] postgresql-17-infocentre
```

- Ouvrir le nouveau port dans le firewall au besoin.

3.5.2 Sur Debian / Ubuntu

Sauf précision, tout est à effectuer en tant qu'utilisateur **root**.

Référence : https://apt.postgresql.org/

Installation du dépôt communautaire :

L'installation des dépôts du PGDG est prévue dans le paquet Debian :

```
# apt update
# apt install -y  gnupg2  postgresql-common
# /usr/share/postgresql-common/pgdg/apt.postgresql.org.sh
```

Ce dernier ordre créera le fichier du dépôt `/etc/apt/sources.list.d/pgdg.list` adapté à la distribution en place.

Installation de PostgreSQL 17 :

La méthode la plus propre consiste à modifier la configuration par défaut avant l'installation :

Dans `/etc/postgresql-common/createcluster.conf`, paramétrer au moins les sommes de contrôle et les traces en anglais :

```
initdb_options = '--data-checksums --lc-messages=C'
```

Puis installer les paquets serveur et clients et leurs dépendances :

```
# apt install postgresql-17 postgresql-client-17
```

La première instance est automatiquement créée, démarrée et déclarée comme service à lancer au démarrage du système. Elle porte un nom (par défaut `main`).

Elle est immédiatement accessible par l'utilisateur système **postgres**.

Chemins :

Objet	Chemin
Binaires	`/usr/lib/postgresql/17/bin/`
Répertoire de l'utilisateur **postgres**	`/var/lib/postgresql`
PGDATA de l'instance par défaut	`/var/lib/postgresql/17/main`
Fichiers de configuration	dans `/etc/postgresql/17/main/`
Traces	dans `/var/log/postgresql/`

Configuration

Modifier `postgresql.conf` est facultatif pour un premier essai.

Démarrage/arrêt de l'instance, rechargement de configuration :

Debian fournit ses propres outils, qui demandent en paramètre la version et le nom de l'instance :

```
# pg_ctlcluster 17 main [start|stop|reload|status|restart]
```

Démarrage de l'instance avec le serveur :

C'est en place par défaut, et modifiable dans `/etc/postgresql/17/main/start.conf`.

Ouverture du firewall :

Debian et Ubuntu n'installent pas de firewall par défaut.

Statut des instances du serveur :

```
# pg_lsclusters
```

Test rapide de bon fonctionnement et connexion à psql :

```
# systemctl --all |grep postgres
# sudo -iu postgres psql
```

Destruction d'une instance :

```
# pg_dropcluster 17 main
```

Création d'autres instances :

Ce qui suit est valable pour remplacer l'instance par défaut par une autre, par exemple pour mettre les *checksums* en place :

- optionnellement, `/etc/postgresql-common/createcluster.conf` permet de mettre en place tout d'entrée les *checksums*, les messages en anglais, le format des traces ou un emplacement séparé pour les journaux :

```
initdb_options = '--data-checksums --lc-messages=C'
log_line_prefix = '%t [%p]: [%l-1] user=%u,db=%d,app=%a,client=%h '
waldir = '/var/lib/postgresql/wal/%v/%c/pg_wal'
```

- créer une instance :

```
# pg_createcluster 17 infocentre
```

Il est également possible de préciser certains paramètres du fichier `postgresql.conf`, voire les chemins des fichiers (il est conseillé de conserver les chemins par défaut) :

```
# pg_createcluster 17 infocentre \
  --port=12345 \
  --datadir=/PGDATA/17/infocentre \
  --pgoption shared_buffers='8GB' --pgoption work_mem='50MB' \
  --  --data-checksums --waldir=/ssd/postgresql/17/infocentre/journaux
```

- adapter au besoin `/etc/postgresql/17/infocentre/postgresql.conf` ;
- démarrage :

```
# pg_ctlcluster 17 infocentre start
```

3.5.3 Accès à l'instance depuis le serveur même (toutes distributions)

Par défaut, l'instance n'est accessible que par l'utilisateur système **postgres**, qui n'a pas de mot de passe. Un détour par `sudo` est nécessaire :

```
$ sudo -iu postgres psql
psql (17.0)
Type "help" for help.
postgres=#
```

Ce qui suit permet la connexion directement depuis un utilisateur du système :

Pour des tests (pas en production !), il suffit de passer à `trust` le type de la connexion en local dans le `pg_hba.conf` :

```
local   all             postgres                                trust
```

La connexion en tant qu'utilisateur `postgres` (ou tout autre) n'est alors plus sécurisée :

```
dalibo:~$ psql -U postgres
psql (17.0)
Type "help" for help.
postgres=#
```

Une authentification par mot de passe est plus sécurisée :

- dans `pg_hba.conf`, paramétrer une authentification par mot de passe pour les accès depuis `localhost` (déjà en place sous Debian) :

```
# IPv4 local connections:
host    all             all             127.0.0.1/32            scram-sha-256
# IPv6 local connections:
host    all             all             ::1/128                 scram-sha-256
```

(Ne pas oublier de recharger la configuration en cas de modification.)

- ajouter un mot de passe à l'utilisateur `postgres` de l'instance :

```
dalibo:~$ sudo -iu postgres psql
psql (17.0)
Type "help" for help.
postgres=# \password
Enter new password for user "postgres":
Enter it again:
postgres=# quit

dalibo:~$ psql -h localhost -U postgres
Password for user postgres:
psql (17.0)
Type "help" for help.
postgres=#
```

- Pour se connecter sans taper le mot de passe à une instance, un fichier `.pgpass` dans le répertoire personnel doit contenir les informations sur cette connexion :

```
localhost:5432:*:postgres:motdepassetrèslong
```

Ce fichier doit être protégé des autres utilisateurs :

```
$ chmod 600 ~/.pgpass
```

- Pour n'avoir à taper que `psql`, on peut définir ces variables d'environnement dans la session voire dans `~/.bashrc` :

```
export PGUSER=postgres
export PGDATABASE=postgres
export PGHOST=localhost
```

Rappels :

- en cas de problème, consulter les traces (dans `/var/lib/pgsql/17/data/log` ou `/var/log/postgresql/`) ;
- toute modification de `pg_hba.conf` ou `postgresql.conf` impliquant de recharger la configuration peut être réalisée par une de ces trois méthodes en fonction du système :

```
root:~# systemctl reload postgresql-17

root:~# pg_ctlcluster 17 main reload

postgres:~$ psql -c 'SELECT pg_reload_conf()'
```

3.6 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/r56_solutions.

Ce TP utilise plusieurs VM : **p1**, **p2**, **p3** et **b1**.
Les commandes `systemctl` doivent être exécutées avec l'utilisateur **root** ou (de préférence) avec la commande `sudo`, depuis un utilisateur qui dispose des droits nécessaire. Par défaut, ce n'est pas le cas pour l'utilisateur **postgres**. Dans les machines virtuelles, l'utilisateur **admin** dispose de ces droits.
Il est possible de changer ce comportement, en exécutant la commande `sudo visudo` et en ajoutant la ligne `postgres` après `root` :

```
root     ALL=(ALL:ALL)
postgres ALL=(ALL) NOPASSWD: ALL
```

L'utilisateur **postgres** sera sudoer au prochain démarrage de session. Attention à la syntaxe et à l'éventuel message d'erreur de `visudo`, une erreur dans le fichier peut empêcher tout accès à la VM. Cette solution n'est pas recommandée en production car elle donne trop de droits a l'utilisateur **postgres**.

3.6.1 Sur Rocky Linux 8 ou 9

Ce TP utilise deux VM : **p1** et **p2**.
Dans la réalité, les accès entre VM seront probablement filtrés par des firewalls qu'il faudra configurer. `firewalld` n'est pas activé sur les VM de TP.

3.6.1.1 Réplication asynchrone en flux avec un seul secondaire

But : Mettre en place une réplication asynchrone en flux.

- Créer l'instance principale dans `/var/lib/pgsql/17/main` sur le serveur **p1**. Les sommes de contrôles devront être activées.

- Mettre en place la configuration de la réplication par *streaming*.
- L'utilisateur dédié sera nommé **repli**.

- Créer la première instance secondaire sur le serveur **p2**, par copie **à chaud** du répertoire de données.

- Démarrer la nouvelle instance sur **p2** et s'assurer que la réplication fonctionne bien avec `ps`.
- Tenter de se connecter au serveur secondaire.
- Créer quelques tables pour vérifier que les écritures se propagent du primaire au secondaire.

3.6.1.2 Promotion de l'instance secondaire

But : Promouvoir un serveur secondaire en primaire.

- En respectant les étapes de vérification de l'état des instances, effectuer une promotion contrôlée de l'instance secondaire.

- Tenter de se connecter au serveur secondaire fraîchement promu.
- Les écritures y sont-elles possibles ?

3.6.1.3 Retour à la normale

But : Revenir à l'architecture d'origine.

- Reconstruire l'instance initiale sur **p1** comme nouvelle instance secondaire en repartant d'une copie complète de **p2** en utilisant `pg_basebackup`.

- Démarrer cette nouvelle instance.
- Vérifier que les processus adéquats sont bien présents, et que les données précédemment insérées dans les tables créées plus haut sont bien présentes dans l'instance reconstruite.

- Inverser à nouveau les rôles des deux instances afin que **p2** redevienne l'instance secondaire, cette fois-ci effectuer la remise en service de l'ancienne instance primaire sans reconstruction.

3.6.1.4 Optionnel: Types de réplication synchrone

But : Montrer le fonctionnement de la réplication synchrone en fonction du paramétrage de `synchronous_standby_names`.

Au début du TP, deux instances sont disponibles **p1**, la primaire, et **p2** sa standby.

- Configurer la réplication synchrone de **p1** vers **p2**.
- Stopper l'instance secondaire **p2**, créer une table `t1`, que se passe-t-il ?
- Redémarrer l'instance **p2**, que se passe-t-il ?
- Stopper l'instance **p2**, créer une table `t2`. Annuler la requête bloquée avec la commande `pg_cancel_backend()`, qu'observez-vous ?

- Suivre la méthode utilisée dans les TP précédants pour mettre en place une seconde instance standby sur p3.

- Configurer la réplication synchrone de **p1** vers **p2** et **p3** en utilisant la syntaxe `p2, p3` pour `synchronous_standby_names`. Observer les champs `sync_priority` et `sync_state` de `pg_stat_replication`.
- Stopper **p2** puis **p3**, redémarrer **p3** puis **p2**. À chaque étape regarder l'état de la vue `pg_stat_replication`.
- Modifier la configuration de **p1** pour demander deux standbys synchrones.

- Configurer `synchronous_standby_names` à `*`. Répéter les mêmes tests que précédemment. Qu'observez-vous ?
- Redémarrer **p2**.

- Mettre en place la réplication synchrone par *quorum* avec un serveur synchrone minimum.

- Désactiver la réplication synchrone.

3.6.1.5 Optionnel: Log shipping et initialisation d'instance avec pgBackRest

But : Montrer l'utilisation de pgBackRest dans une architecture de réplication

Nous allons utiliser la vm **b1** comme serveur de sauvegarde et y créer un dépôt pgBackRest dédié.

- Installer pgBackRest sur tous les serveurs (**p1**, **p2**, **p3**, **b1**).

- Configurer pgBackRest sur le serveur de sauvegarde.
- Configurer pgBackRest sur les serveurs de base de données.
- Créer la stanza, tester l'archivage.
- Lancer une sauvegarde.

- Si **p3** existe, le stopper et détruire le répertoire de données.
- Créer l'instance à partir de la sauvegarde.
- Interdire la connexion à **p1** depuis **p3**, que se passe-t-il sur **p3** ?
- Faire marche arrière sur la modification, que se passe-t-il sur **p3** ?

3.6.1.6 Optionnel: pg_rewind

But : Remonter une instance secondaire après une divergence.

L'instance **p1** est l'instance primaire.

- Promouvoir l'instance **p2**. Générer de l'activité sur **p1**.
- Stopper **p1** et configurer la réplication pour raccrocher **p1** à **p2**. Qu'observez-vous ?

- Utiliser `pg_rewind` pour remettre l'instance **p1** d'aplomb.

3.6.2 Sur Debian 12

Ce TP utilise deux machines virtuelles : **p1** et **p2**.
Dans la réalité, les accès entre VM seront probablement filtrés par des firewalls qu'il faudra configurer. Par défaut, aucun firewall n'est installé sur Debian, cette étape sera donc inutile dans notre cas.

3.6.2.1 Réplication asynchrone en flux avec un seul secondaire

But : Mettre en place une réplication asynchrone en flux.

- Créer l'instance principale en utilisant `pg_createcluster` sur le serveur **p1**. Les sommes de contrôles devront être activées.

- Vérifier la configuration de la réplication par *streaming*.
- L'utilisateur dédié sera nommé **repli**.

- Créer la première instance secondaire sur le serveur **p2**, par copie **à chaud** du répertoire de données. Le répertoire dédié aux fichiers de configuration devra également être copié.

- Démarrer la nouvelle instance sur **p2** et s'assurer que la réplication fonctionne bien avec `ps`.
- Tenter de se connecter au serveur secondaire.
- Créer quelques tables pour vérifier que les écritures se propagent du primaire au secondaire.

3.6.2.2 Promotion de l'instance secondaire

But : Promouvoir un serveur secondaire en primaire.

- En respectant les étapes de vérification de l'état des instances, effectuer une promotion contrôlée de l'instance secondaire.

- Tenter de se connecter au serveur secondaire fraîchement promu.
- Les écritures y sont-elles possibles ?

3.6.2.3 Retour à la normale

But : Revenir à l'architecture d'origine.

- Reconstruire l'instance initiale sur **p1** comme nouvelle instance secondaire en repartant d'une copie complète de **p2** en utilisant `pg_basebackup`.

- Démarrer cette nouvelle instance.
- Vérifier que les processus adéquats sont bien présents, et que les données précédemment insérées dans les tables créées plus haut sont bien présentes dans l'instance reconstruite.

- Inverser à nouveau les rôles des deux instances afin que **p2** redevienne l'instance secondaire, cette fois-ci effectuer la remise en service de l'ancienne instance primaire sans reconstruction.

3.6.2.4 Optionnel: Types de réplication synchrone

But : Montrer le fonctionnement de la réplication synchrone en fonction du paramétrage de `synchronous_standby_names`.

Au début du TP, deux instances sont disponibles **p1**, la primaire, et **p2** sa standby.

- Configurer la réplication synchrone de **p1** vers **p2**.
- Stopper l'instance secondaire **p2**, créer une table `t1`, que se passe-t-il ?
- Redémarrer l'instance **p2**, que se passe-t-il ?
- Stopper l'instance **p2**, créer une table `t2`. Annuler la requête bloquée avec la commande `pg_cancel_backend()`, qu'observez-vous ?

- Suivre la méthode utilisée dans les TP précédants pour mettre en place une seconde instance standby sur p3.

- Configurer la réplication synchrone de **p1** vers **p2** et **p3** en utilisant la syntaxe `p2, p3` pour `synchronous_standby_names`. Observer les champs `sync_priority` et `sync_state` de `pg_stat_replication`.
- Stopper **p2** puis **p3**, redémarrer **p3** puis **p2**. À chaque étape regarder l'état de la vue `pg_stat_replication`.
- Modifier la configuration de **p1** pour demander deux standbys synchrones.

- Configurer `synchronous_standby_names` à `*`. Répéter les mêmes tests que précédemment. Qu'observez-vous ?
- Redémarrer **p2**.

- Mettre en place la réplication synchrone par *quorum* avec un serveur synchrone minimum.

- Désactiver la réplication synchrone.

3.6.2.5 Optionnel: Log shipping et initialisation d'instance avec pgBackRest

But : Montrer l'utilisation de pgBackRest dans une architecture de réplication

Nous allons utiliser la vm **b1** comme serveur de sauvegarde et y créer un dépôt pgBackRest dédié.

- Installer pgBackRest sur tous les serveurs (**p1**, **p2**, **p3**, **b1**).

- Configurer pgBackRest sur le serveur de sauvegarde.
- Configurer pgBackRest sur les serveurs de base de données.
- Créer la stanza, tester l'archivage.
- Lancer une sauvegarde.

- Si **p3** existe, le stopper et détruire le répertoire de données.
- Créer l'instance à partir de la sauvegarde.
- Interdire la connexion à **p1** depuis **p3**, que se passe-t-il sur **p3** ?
- Faire marche arrière sur la modification, que se passe-t-il sur **p3** ?

3.6.2.6 Optionnel: pg_rewind

But : Remonter une instance secondaire après une divergence.

L'instance **p1** est l'instance primaire.

- Promouvoir l'instance **p2**. Générer de l'activité sur **p1**.
- Stopper **p1** et configurer la réplication pour raccrocher **p1** à **p2**. Qu'observez-vous ?

- Utiliser `pg_rewind` pour remettre l'instance **p1** d'aplomb.

4/ etcd : Architecture et fonctionnement

4.1 AU MENU

- L'algorithme Raft
- Implémentation Raft dans etcd
- Installation etcd
- Fonctionnalités d'etcd
- Tâches d'administrations d'un cluster etcd

Le DCS est un composant critique de tout cluster Patroni. Ne pouvant consacrer un module de formation pour chaque DCS supporté par Patroni, nous avons choisi d'approfondir etcd car il est simple, robuste et populaire.

Ce module aborde la théorie du protocole Raft qui est la source de l'implémentation d'etcd. Cette théorie est utile pour comprendre les réactions d'un cluster etcd ou savoir en interpréter les journaux applicatifs.

Les chapitres suivants se consacrent pleinement à l'étude d'etcd, de son installation à son utilisation et administration.

4.2 L'ALGORITHME RAFT

- Algorithme de consensus
- *Replicated And Fault Tolerant*
- *Leader-based requests*

Raft[1] est un algorithme de consensus répliqué et tolérant aux pannes (*Replicated And Fault Tolerant*). Ce genre d'algorithme vise à permettre à un ensemble de serveurs de fonctionner comme un groupe cohérent pouvant survivre à la disparition d'un certain nombre de membres.

L'élaboration de Raft part du constat que les algorithmes de consensus comme Paxos[2] sont complexes à comprendre et donc à implémenter. C'est un problème important, car ils sont utilisés dans des applications critiques qui nécessitent d'être bien comprises et maîtrisées. L'un des principaux objectifs lors de l'élaboration de Raft était donc de le rendre, dans la mesure du possible, simple à comprendre et à implémenter en plus d'être : prouvé, complet et fonctionnel.

Avec Raft, les clients dialoguent uniquement avec le nœud *leader*. Si une requête est envoyée à un nœud secondaire (*follower*), elle va échouer en renvoyant des informations sur l'adresse du *leader*. Néanmoins nous verrons plus loin que son implémentation dans etcd lève partiellement cette restriction.

Dans notre infrastructure, les clients d'etcd sont les processus Patroni, et leurs requêtes correspondent à :

- un changement d'état du nœud ;
- une modification de configuration d'une instance.

[1] https://raft.github.io/raft.pdf
[2] https://fr.wikipedia.org/wiki/Paxos_(informatique)

4.2.1 Raft : Journal et machine à états

- Machine à états : *leader*, *follower* ou *candidate*
- Journal des événements :
  - répliqué depuis le *leader*
  - même résultat sur tous les nœuds
- Mandats :
  - commence par une élection du *leader* (unique)
  - numérotés, croissants

Raft implémente une machine à états déterministe qui utilise un journal répliqué.

La machine à états joue les modifications les unes après les autres, dans leur ordre d'arrivée. Cela garantit que la machine avance d'un état stable à l'autre et produit le même résultat sur tous les nœuds.

L'algorithme de consensus se charge de maintenir la cohérence du journal répliqué.

Dans Raft chaque nœud est dans l'un des trois états suivant :

- *follower* (suiveur) : c'est l'état par défaut lorsqu'on démarre un nœud. Dans cet état, le nœud est passif. Il s'attend à recevoir régulièrement un message du *leader* (*heart beat* via *Append Entries RPC*) et potentiellement de nœuds candidats (*Request Vote RPC*). Il se contente de répondre à ces messages.
- *candidate* (candidat) : c'est l'état que prend un nœud *follower* s'il ne reçoit pas de message du *leader* pendant un certain temps. Dans cet état, il va envoyer des demandes de votes (*Request Vote RPC*) aux autres nœuds pour se faire élire comme nouveau *leader*. S'il perd l'élection, il redevient *follower*.
- *leader* : c'est l'état que prend un candidat après avoir remporté une élection. Dans cet état, le nœud va envoyer des messages d'ajout dans les journaux (*Apprend Entries RPC*) à intervalle régulier. Ces messages servent à la fois pour répliquer les commandes envoyées au *leader* par les clients et pour signifier aux *followers* que le *leader* est toujours vivant (*heart beat*). Si le *leader* découvre un *leader* avec un mandat supérieur au sien, il redevient *follower* (voir règles ci-dessous).

Les mandats (*terms*) servent à déterminer quand des informations sont obsolètes dans l'agrégat.

Les règles suivantes concernent les mandats :

- chaque mandat commence par une élection ;
- les mandats sont numérotés avec des entiers consécutifs ;

- chaque nœud ne peut voter qu'une fois par mandat ;
- il ne peut y avoir qu'un seul *leader* par mandat. Il est possible qu'un mandat n'ait pas de *leader* si une élection a échoué (*split vote*) ;
- chaque serveur garde une trace du mandat dans lequel il évolue ; l'identifiant de mandat est échangé dans tous les messages.
- si un nœud reçoit un message avec un identifiant de mandat supérieur, il met à jour son identifiant de mandat et redevient *follower* s'il ne l'est pas déjà ;
- si un nœud reçoit un message avec un identifiant de mandat inférieur au sien, il ignore la demande et renvoie une erreur à son expéditeur.

Raft sépare en trois parties les éléments essentiels pour le consensus :

- élection d'un *leader* : il ne doit toujours y avoir qu'un seul *leader*, si un *leader* disparaît un nouveau doit le remplacer ;
- réplication du journal : le *leader* reçoit les modifications des clients et les réplique vers les autres serveurs ; le journal du *leader* fait toujours référence ;
- sécurité & cohérence : si un serveur a appliqué une entrée du journal dans sa machine à état, les autres serveurs doivent appliquer la même entrée provenant de la même position dans le journal.

4.2.2 Élection d'un leader

- *Heart beats* depuis le *leader* vers les *followers*
- Si pas de nouvelles du *leader* :
  - démarrage de l'élection d'un nouveau *leader*
- Promotion par consensus entre les membres
- Si échec de l'élection, attente aléatoire
- Tolérance de panne à partir de 3 nœuds

Le *leader* informe régulièrement tous les *followers* de sa présence (*heart beart*) par l'envoi d'un message d'ajout au journal (*Append Entries RPC*) qui peut être vide s'il n'y a pas d'activité.

Si le message tarde à arriver sur un nœud, ce dernier présume que le *leader* a disparu et devient candidat. Il incrémente alors son numéro de mandat, vote pour lui-même et effectue une demande d'élection (*Request Vote RPC*).

Si le candidat obtient la majorité des votes, il remporte l'élection et devient *leader*. Il envoie alors un message d'*heart beat* aux autres serveurs afin de leur signifier la présence d'un nouveau *leader*.

Si le candidat reçoit un message de *heart beat* en provenance d'un autre nœud alors qu'il est en train d'attendre des votes, il vérifie le numéro de mandat. Si le mandat de ce serveur est supérieur ou égal au

mandat du candidat, cela signifie qu'un autre nœud a été élu *leader*. Dans ce cas, le candidat devient *follower*. Sinon, il renvoie un message d'erreur et continue son élection.

Le dernier cas de figure est qu'aucun candidat ne parvienne à remporter l'élection pendant ce mandat. Dans ce cas, les candidats vont atteindre leur *timeout* d'élection et en démarrer une nouvelle. Ce cas de figure est désigné sous le nom de *split vote*.

Le temps d'attente avant le *timeout* est d'une durée aléatoire choisie dans une plage prédéfinie (par exemple 150 ms-300 ms) désignée sous le nom d'*election timeout*. Cette randomisation limite l'occurrence des demandes de vote simultanées et la multiplication de *split votes*.

Pour que la tolérance de panne soit assurée, il faut préserver un nombre de nœuds fonctionnels suffisant pour obtenir une majorité : au minimum **trois**.

4.2.3 Réplication du journal

- Mécanisme de réplication
- Structure des journaux
- Protection contre les incohérences
    - n° de mandat + index

L'application cliente envoie sa commande au *leader*. Cette commande est ajoutée dans son journal avant d'être envoyée aux *followers* dans un message d'ajout (*Append Entries RPC*).

Si une commande est répliquée dans les journaux de la majorité des nœuds de l'agrégat, elle est exécutée par le *leader* (commitée) qui répond ensuite au client. Le *leader* conserve une trace de l'index du dernier enregistrement de log appliqué dans la machine à états (*commit index*), dont la valeur est incluse dans les messages envoyés aux *followers* (*Append Entries RPC*). Cette information permet aux *followers* de savoir quels enregistrements de leur journal peuvent être exécutés par leur machine à états.

Si un *follower* est indisponible, le *leader* tente de renvoyer le message jusqu'à ce qu'il soit reçu.

Raft s'appuie sur les numéros d'index (position dans le journal) et de mandat comme prédicats pour garantir la cohérence du journal (*log matching properties*). Si deux entrées de différents journaux ont ces mêmes numéros :

- elles stockent la même commande ;
- les entrées précédentes sont identiques.

Le premier prédicat est garanti par le fait que le *leader* ne produit qu'une seule entrée avec un numéro d'index donné pendant un mandat. De plus les entrées créées dans le journal ne changent pas de position dans le journal.

Le second prédicat est garanti par la présence supplémentaire des numéros d'index et de mandat de la commande précédente dans chaque demande d'ajout au journal (*Append Entries RPC*). Le *follower* n'écrit un enregistrement dans son journal que si l'index et le mandat de l'enregistrement précédent correspondent à ce qu'a envoyé le *leader*. Sinon la demande du *leader* est rejetée.

En temps normal, ce contrôle de cohérence ne tombe jamais en échec. Cependant, suite à une élection, plusieurs cas de figure sont possibles :

- le *follower* est à jour, il n'y a rien à faire ;
- le *follower* est en retard, il faut lui communiquer les entrées manquantes ;
- le journal du *follower* contient des entrées qui ne sont pas sur le *leader* : cela peut arriver lorsqu'un ancien *leader* redevient *follower* alors qu'il n'avait pas encore reçu de consensus sur des valeurs qu'il avait écrites dans son journal.

Pour gérer ces trois situations, le *leader* maintient un index (*next index*) pour chaque *follower* qui correspond à la position du prochain enregistrement qui sera envoyé à ce *follower*. Suite à l'élection, cet index pointe vers la position de l'enregistrement qui suit le dernier enregistrement du journal. De cette manière si le journal d'un *follower* n'est pas cohérent avec celui du *leader*, le prochain message d'ajout dans les journaux sera en échec. Le *leader* va alors décrémenter son compteur d'une position et renvoyer un nouveau message. Ce processus va se répéter jusqu'à ce que le *leader* et le *follower* trouvent un point de cohérence entre leurs journaux. Une fois cette position trouvée, le *follower* tronque la suite du journal et applique toutes modifications en provenance du *leader*.

4.2.4 Sécurité & cohérence

Lors d'une élection:

- les votants refusent tout candidat en retard par rapport à eux

Quelques protections supplémentaires sont nécessaires pour garantir la cohérence du journal distribué en cas d'incident majeur.

La plus importante est mise en place lors du processus d'élection : les nœuds votants refusent de voter pour un candidat qui est en retard par rapport à eux. Cela est rendu possible grâce à la présence du numéro de mandat et d'index du dernier enregistrement du journal du candidat dans le message de demande de vote (*Request Vote RPC*). Le numéro de mandat du candidat doit être supérieur ou égal à celui du votant. Si les numéros de mandat sont égaux, les numéros d'index sont comparés avec le même critère.

4.2.5 Majorité et tolérance de panne

- Nombre impair de nœuds total recommandé
- Majorité : `quorum = (nombre de nœuds / 2) + 1`
- Tolérance de panne : `nombre de nœuds - quorum`
- Pas de quorum : pas de réponse au client

Pour éviter le statu quo ou l'égalité entre deux nœuds, il faut pouvoir les départager par un déséquilibre du vote et ceci dans un laps de temps donné.

La promotion d'un nœud candidat ne peut avoir lieu que si la majorité des nœuds a voté pour lui.

La tolérance de panne définit combien de nœuds l'agrégat peut perdre tout en restant opérationnel, c'est-à-dire de toujours pouvoir procéder à une élection.

Un agrégat ayant perdu son quorum ne doit pas répondre aux sollicitations des clients.

4.2.6 Tolérance de panne : Tableau récapitulatif

Nombre de nœuds	Majorité	Tolérance de panne
2	2	0
⇒ **3**	2	**1**
4	3	1
⇒ **5**	3	**2**
6	4	2
⇒ **7**	4	**3**
8	5	3

L'ajout d'un nœud à un cluster de taille impaire semble une bonne idée au premier abord. Mais si la probabilité [3] d'avoir un cluster hors service est bien légèrement réduite, la tolérance de panne reste inchangée, à un seul serveur. Le rapport sécurité/investissement n'est donc pas intéressant. Ce rapport

[3] La probabilité d'avoir au moins (tolérance de panne + 1) nœuds en panne, qui est égale à p^2 pour un cluster à trois nœuds, et $p^2 - p^3(1-p)$ pour un cluster à quatre nœuds, où p est la probabilité de panne d'un nœud (le calcul de cette probabilité est laissée en exercice au lecteur).

est toujours plus intéressant avec un nombre **impair** de nœuds. Notons aussi que l'ajout de chaque nœud augmente le trafic réseau au sein du cluster, et le rend moins performant, ce qui est en soi un risque pour la fiabilité de l'ensemble. Enfin et surtout, un cluster avec quatre nœuds présente même un risque accru de panne si deux nœuds sont placés dans un même *datacenter*, car la perte de celui-ci entraîne alors la perte de la majorité. même un risque accru de panne si deux nœuds sont placés dans un même *datacenter*, car la perte de celui-ci entraîne alors la perte de la majorité.

4.2.7 Interaction avec les clients

- Toutes les communications passent par le *leader*
- Protection contre l'exécution en double des requêtes
- Le *leader* valide les entrées non commitées après son élection
- Le *leader* vérifie qu'il est bien *leader* avant de répondre à une demande de lecture

Dans Raft, les clients communiquent uniquement avec le *leader*. Si un autre nœud reçoit un message provenant d'un client, il rejette la demande et communique l'adresse du *leader*.

En l'état, Raft peut rejouer deux fois les mêmes commandes : par exemple, si un *leader* plante après avoir validé un enregistrement mais avant d'avoir confirmé son exécution au client. Dans ce cas, le client risque de renvoyer la commande, qui sera exécutée une seconde fois.

Afin d'éviter cela, le client doit associer à chaque commande un numéro unique. La machine à état garde trace du dernier numéro traité, si elle reçoit une commande avec un numéro qui a déjà été traité, elle répond directement sans ré-exécuter la requête.

Immédiatement après l'élection d'un nouveau *leader*, ce dernier ne sait pas quelles entrées sont exécutées dans son journal. Pour résoudre ce problème immédiatement après son élection, le leader envoie un message vide d'ajout dans le journal (*Append Entries RPC*) avec un index (*next_index*) qui pointe sur l'entrée qui suit la dernière entrée de son journal. De cette manière, tous les *followers* vont se mettre à jour et les entrées du journal du *leader* pourront être exécutées.

Une dernière précaution est que le *leader* envoie toujours un *heart beat* avant de traiter une demande de lecture. De cette manière, on s'assure qu'un autre *leader* n'a pas été élu entre-temps.

4.2.8 Raft en action

Démo interactive :

- Raft en animation[4]

Cette animation permet de simuler le fonctionnement d'un cluster Raft en ajoutant ou enlevant à volonté des nœuds.

Une autre animation plus guidée existe aussi chez *The Secret Lives of Data*[5].

[5] http://thesecretlivesofdata.com/raft/

4.3 MÉCANIQUE D'ETCD

- Serveur de stockage distribué par consensus
  - stockage clé/valeur
  - la majorité des actions requièrent d'avoir le quorum
- Haute disponibilité et tolérance de panne
  - Multiples nœuds
  - Nécessite 3 nœuds minimum pour avoir une tolérance de panne

Etcd est un serveur de stockage distribué par consensus, écrit en Go avec pour ligne directrice la haute disponibilité. Le projet etcd a été créé à l'origine par CoreOS, puis développé par une communauté de développeurs. Il est distribué sous licence libre (Apache License 2.0). Le nom etcd signifie « `etc` distribué », `etc` étant le nom du répertoire dédié au stockage de la configuration sous linux. La configuration consiste généralement en un ensemble de clés/valeurs, c'est le format qu'utilise etcd.

Les données stockées sont répliquées sur tous nœuds composants le cluster, l'algorithme Raft assurant le consensus entre eux à propos des données validées et de leur visibilité.

Si etcd est capable de fonctionner avec un seul membre, nous verrons plus loin que la tolérance de panne nécessite un agrégat d'au minimum trois nœuds, assurant ainsi un quorum essentiel lors des élections du leader.

4.3.1 etcd V2 et V3

- Abandon de l'API REST au profit de gRPC
- *endpoints* différents pour la supervision
- Commandes différentes dans `etcdctl` (`ETCDCTL_API`)
- Nouvelles fonctionnalités (ou implémentations différentes):
  - transactions
  - leases
  - quorum & lecture linéarisées / sérialisées
- Possibilité d'activer le protocole v2 :
  - deux bases de données séparées, une par version de l'API
  - retiré dans etcd v3.6
- Utilisez l'API v3
  - supportée par Patroni depuis la version 2.0.0 (septembre 2020)

4.3.2 etcd et Raft

- Implémente l'algorithme Raft
  - hautement disponible: élections et réplication par consensus
  - tolérance de panne possible qu'à partir de 3 nœuds
  - différence : requêtes sur un *followers*
- Nombre de membres impair, recommandé entre 3 et 7
- Mutualisation du cluster etcd pour plusieurs clusters Patroni

Etcd implémente l'algorithme Raft. Les principes expliqués précédemment concernant les mécanismes d'élection, la réplication par consensus, le quorum et la tolérance de panne sont donc toujours valables ici.

Contrairement à ce que préconise Raft, il est possible d'adresser des requêtes à un membre *follower* d'etcd. Toute requête qui nécessite un consensus sera renvoyée automatiquement au *leader*. Les autres peuvent être traitées par n'importe quel membre (eg: les lectures sérialisées[6] avec par exemple l'option `--consistency=s` d'`etcdctl`).

[6] https://github.com/etcd-io/etcd/blob/4d209af83a07ee741a2df7d0e34a28792a2bee57/etcdctl/README.md#output-1

Pour que l'architecture soit résiliente, il faut **au minimum** qu'elle puisse survivre à la perte d'un de ses membres. Pour qu'une élection soit possible dans ce cas, il faut au minimum 3 serveurs.

Bien qu'aucune limite stricte n'existe, un maximum de 7 nœuds est conseillé[7]. Au delà de 7, l'impact sur les performances en écriture peut devenir trop important.

Ajouter un nœud à un nombre *impair* de nœuds n'améliore pas la tolérance de panne et ne fait qu'augmenter la complexité. **Pour augmenter la tolérance de panne, l'ajout de nœuds doit porter le nombre total de nœuds à un nombre impair.**

Un même cluster etcd peut gérer plusieurs agrégats Patroni distincts. En effet, la configuration de Patroni permet de spécifier un chemin différent pour les données de chaque cluster.

De plus, les informations stockées par Patroni sont relativement peu volumineuses. La seule donnée susceptible d'occuper un volume croissant d'espace est l'historique des timelines de PostgreSQL, dont on peut limiter la taille dans le paramétrage de Patroni.

Il est aussi possible d'utiliser le cluster etcd pour d'autres usages. Il faudra alors vérifier que les spécifications du serveur sont suffisantes.

4.3.3 Modes de défaillance d'etcd

- Indisponibilité :
  - du *leader*
  - d'un nombre de membres inférieur ou égal à la tolérance de panne
  - d'un nombre de membres supérieur à la tolérance de panne
- Partition réseau
- Échec lors de l'initialisation du cluster (*bootstrap*)

Indisponibilité du *Leader*

L'indisponibilité du *leader* donne lieu à une élection afin de le remplacer. Cette élection n'est pas instantanée car déclenchée par le timeout d'un des *followers*. Pendant l'élection, toute écriture est impossible. Les écritures qui n'ont pas encore été validées peuvent être perdues en fonction d'où elles ont été répliquées et du nouveau *leader* élu. Les autres actions nécessitant un consensus sont également bloquées.

Si des baux sont encore en cours (*leases*), le nouveau *leader* va étendre leur timeout afin d'éviter qu'ils n'expirent à cause de l'indisponibilité induite par la bascule de leader.

[7] https://etcd.io/docs/v3.5/faq/#what-is-maximum-cluster-size

Indisponibilité d'un nombre de membres inférieur ou égal à la tolérance de panne

Tant qu'il reste suffisamment de membres dans le cluster pour satisfaire le quorum, il pourra continuer à servir les requêtes des utilisateurs. Si les *followers* sont habituellement sollicités pour traiter des opérations qui ne nécessitent pas de consensus, cette charge sera déportée vers les serveurs restants, ce qui peut impacter négativement les performances.

Indisponibilité d'un nombre de membres supérieur à la tolérance de panne

Lorsque le cluster perd un nombre de membres supérieur à la tolérance de panne du cluster, il n'est plus possible de réaliser des opérations nécessitant un consensus.

Si les membres impactés reviennent en service, le quorum est de nouveau établi et une nouvelle élection est alors possible. Le cluster peut alors reprendre son fonctionnement normal avec les mêmes limites que lors de la perte du *leader*.

Si les membres impactés sont définitivement perdus, une intervention est nécessaire. Nous abordons ce sujet plus loin.

Partition réseau

Une partition réseau divise le cluster en deux parties. La partie qui contient suffisamment de serveurs pour maintenir un quorum continue à fonctionner. L'autre partie devient indisponible.

Si le *leader* est présent dans la partition qui continue à fonctionner, la situation correspond au mode de défaillance où un nombre de membres inférieur ou égal à la tolérance de panne est perdu.

Si le *leader* est dans la partition qui devient indisponible, la partition conservant le quorum provoque alors une nouvelle élection.

Lorsque les membres de la partition indisponible reviennent en service, ils rejoignent le cluster et rattrapent leur retard.

Échec d'initialisation du cluster (*bootstrap*)

Le *bootstrap* est considéré comme un succès lorsque suffisamment de membres se joignent au cluster pour former le quorum nécessaire.

En cas d'échec de l'initialisation, il est plus rapide de stopper etcd sur tous les membres, supprimer le répertoire de données de etcd, corriger le problème et redémarrer la procédure de *bootstrap*.

4.4 MISE EN ŒUVRE D'ETCD

4.4.1 Contraintes matérielles et logicielles

- Serveurs dédiées !
- Dimensionner la mémoire pour les données + l'OS
  - quota par défaut : 2 Go (largement supérieur à ce que Patroni utilise)
  - préconisation : 2 à 8 Go
  - attention à l'overcommit / OOM killer
- Des disques rapides pour ne pas ralentir le cluster lors des COMMIT
  - utiliser du SSD
  - tester le stockage avec `fio`
- Un réseau fiable (beaucoup d'échanges entre nœuds etcd)
- Exemple chez CoreOS : proc dual core, 2 Go de RAM, 80 Go de SSD.

CoreOS[8] utilise des serveurs dual core avec 2 Go de RAM et 80 Go de SSD. Cette configuration peut être prise comme référence pour vos serveurs. Étant donné la quantité faible de données stockée dans etcd par Patroni, l'espace disque peut sans doute être revu à la baisse.

Il est important que le serveur ait suffisamment de mémoire pour contenir une quantité de données équivalente au quota (par défaut 2 Go) avec une marge pour les autres opérations. Si le serveur subit des pressions mémoires et que l'overcommit n'est pas désactivé, il risque d'être pris pour cible par l'OOM Killer.

De notre expérience du support, avoir un stockage et un réseau fiable et performant, en latence comme en débit, est essentiel pour la continuité du service.

Il est impératif de dédier des machines virtuelles entières à etcd. Mutualiser avec celles de PostgreSQL est une mauvaise idée : toute saturation de la base pouvant entraîner une indisponibilité du nœud etcd et une bascule.

[8] https://fedoraproject.org/coreos/

4.4.2 Installation des binaires

- RedHat/EL depuis le dépôt PGDG :
  - `dnf install etcd`
  - firewalld
- Debian/Ubuntu :
  - `apt-get install etcd` (Debian 11)
  - `apt-get install etcd-server etcd-client` (Debian 12)
- Installation depuis les sources (Github)

Red Hat et autres EL

RedHat a retiré etcd de ses dépôts depuis sa version 8. Il est toujours disponible mais seulement sur les infrastructures OpenStack de RedHat. Voir à ce propos: https://access.redhat.com/solutions/6487641.

Des paquets etcd à jour sont néanmoins fournis depuis le dépôt communautaire PGDG. Attention, le paquet `etcd` est distribué depuis le dépôt optionnel du PGDG, nommé `pgdg-rhel9-extras`. Voir à ce propos: https://yum.postgresql.org/news/new-repo-extra-packages/. Dans les commandes suivantes, adapter le numéro de version de l'OS :

```
sudo dnf install -y
 ↪  https://download.postgresql.org/pub/repos/yum/reporpms/EL-9-x86_64/\
pgdg-redhat-repo-latest.noarch.rpm
sudo dnf --enablerepo=pgdg-rhel9-extras install -y etcd
```

Les outils `curl` et `jq` sont utiles pour interroger l'API HTTP d'etcd ou travailler avec sa sortie JSON (ou celle de Patroni) :

```
sudo dnf install jq curl
```

Si le firewall est activé dans votre distribution EL, il est nécessaire d'en adapter la configuration pour autoriser etcd à communiquer :

```
sudo firewall-cmd --permanent --new-service=etcd
sudo firewall-cmd --permanent --service=etcd --set-short=Etcd
sudo firewall-cmd --permanent --service=etcd --set-description="Etcd server"
# communication avec les clients
sudo firewall-cmd --permanent --service=etcd --add-port=2379/tcp
# communication avec le cluster
sudo firewall-cmd --permanent --service=etcd --add-port=2380/tcp
sudo firewall-cmd --permanent --add-service=etcd
sudo firewall-cmd --reload
```

Sur Red Hat, aucun service n'est démarré par défaut. Il est donc nécessaire de l'activer explicitement au démarrage :

```
sudo systemctl enable etcd.service
```

Debian/Ubuntu

Les dépôts de Debian et de ses dérivés proposent les paquets client et serveur d'etcd par défaut à partir de la version 12. Par exemple :

```
sudo apt-get update
/* Debian 12 et après */
sudo apt-get install -y etcd-server etcd-client
```

Pour Debian 11 et antérieur, un seul paquet `etcd` regroupait les deux :

```
sudo apt-get update
/* Debian 11 et avant */
sudo apt-get install -y etcd
```

Les outils `curl` et `jq` sont utiles pour interroger l'API HTTP d'etcd ou travailler avec sa sortie JSON (ou celle de Patroni) :

```
sudo apt-get install -y jq curl
```

Sur Debian/Ubuntu, un cluster d'un nœud local est automatiquement créé et le service `etcd` est démarré immédiatement. Ses données sont stockées dans `/var/lib/etcd/default`. Cette politique par défaut d'empaquetage Debian ne convenant pas à notre utilisation, il est nécessaire d'arrêter le service et d'en supprimer le répertoire de données avant de créer un nouveau cluster multinœud.

```
sudo systemctl stop etcd.service
sudo rm -Rf /var/lib/etcd/default
```

Installation depuis les sources

Il est possible d'installer manuellement etcd sans passer par les paquets proposés ci-dessus.

Sur chacun des nœuds, télécharger les binaires depuis le dépôt Github[9], puis les copier dans les répertoires adéquats :

```
ETCD_VER=v3.5.11    # à adapter à la dernière version disponible

curl -L https://github.com/etcd-io/etcd/releases/download/\
${ETCD_VER}/etcd-${ETCD_VER}-linux-amd64.tar.gz \
 -o etcd-${ETCD_VER}-linux-amd64.tar.gz

mkdir etcd-download
tar xzvf etcd-${ETCD_VER}-linux-amd64.tar.gz -C etcd-download --strip-components=1
rm -f etcd-${ETCD_VER}-linux-amd64.tar.gz

sudo cp etcd-download/etcd* /usr/bin/
sudo chmod +x /usr/bin/etcd*
```

[9] https://github.com/etcd-io/etcd/releases/

```
sudo groupadd --system etcd
sudo useradd -s /sbin/nologin --system -g etcd etcd
sudo mkdir -p /var/lib/etcd
sudo chmod 700 /var/lib/etcd
sudo chown -R etcd:etcd /var/lib/etcd/
```

4.4.3 Configuration etcd

Trois méthodes de configuration différentes:

- En arguments à l'exécutable `etcd`
- En variables d'environnement lues par l'exécutable `etcd`
- Dans un fichier YAML pointé `--config-file`

Etcd supporte trois méthodes de configurations. Chaque paramètre de configuration peut être positionné depuis :

- un argument au binaire `etcd` ;
- une variable d'environnement ;
- un fichier au format YAML.

Format

Les paramètres passés en **argument de la ligne de commande** sont préfixés par `--`.

Les paramètres positionnés en tant que **variables d'environnement** doivent être préfixés par `ETCD_` et déclarées au format *screaming snake case* : tout en majuscule et en remplaçant les tirets par des *underscores*.

Le **fichier de configuration** est à rédiger au format YAML. Un exemple commenté est disponible dans les sources du projet : https://github.com/etcd-io/etcd/blob/main/etcd.conf.yml.sample

Voici quelques exemples de paramètres sous leurs différentes formes:

Fichier de configuration	Ligne de commande	Variable d'environnement
name	--name	ETCD_NAME
initial-cluster	--initial-cluster	ETCD_INITIAL_CLUSTER
listen-peer-urls	--listen-peer-urls	ETCD_LISTEN_PEER_URLS

Précédence

Les paramètres passés en arguments ont la précédence sur ceux positionnés en tant que variables d'environnement.

Si vous fournissez un fichier de configuration (via `--config-file` ou `ETCD_CONFIG_FILE`), tout argument ou variable d'environnement est alors ignoré.

Principaux paramètres de configuration

Un nœud etcd écoute sur deux ports distincts : l'un pour les communications avec ses pairs au sein de l'agrégat (par défaut `2380`), l'autre pour les échanges avec les clients (par défaut `2379`).

Plusieurs thèmes différents sont abordés dans la configuration d'etcd: la configuration locale du nœud, des autres membres, du cluster, de la sécurité, de l'authentification, de la métrologie et des journaux applicatifs, etc.

Voici la liste des principaux paramètres qui nous intéressent :

`config-file` : Fichier de configuration à utiliser, au format YAML. Optionnel si les autres paramètres sont passés en tant qu'arguments sur la ligne de commande ou positionnés en tant que variables d'environnement.

`name` : Nom du nœud. Doit être unique au sein de l'agrégat.

`data-dir` : Chemin vers le répertoire de données.

`listen-peer-urls` : URLs locales complètes des ports d'écoute du nœud pour la communication inter-nœuds au sein de l'agrégat. Par exemple : `http://10.0.0.11:2380`.

`listen-client-urls` : URLs locales complètes des ports d'écoute du nœud pour les clients. Par exemple : `http://127.0.0.1:2379,http://[::1]:2379,http://10.0.0.11:2379`.

`advertise-client-urls` : URLs locales complètes des ports d'écoute pour les clients communiquées aux autres nœuds. Par exemple : `http://10.0.0.11:2379`. Attention, ne positionnez pas ici d'adresse de *loopback* comme `localhost`, `127.0.0.1` ou `::1` ! Cela indiquerait à tort à chaque client distant qu'il peut joindre sur une adresse qui lui serait locale le nœud etcd distant. Si le paramètre `listen-client-urls` contient bien des adresses de *loopback*, ce n'est que par facilité pour utiliser le CLI `etcdctl` depuis le nœud etcd sans avoir à connaître son adresse IP externe.

`enable-grpc-gateway` : Active la passerelle gRPC vers JSON pour le protocole v3 d'etcd. Patroni utilise l'API JSON d'etcd pour interagir avec lui. Normalement activé par défaut, sauf avant la version 3.5 d'ectd si un fichier de configuration est utilisé.

Les paramètres suivants sont utiles lorsqu'un nœud rejoint un agrégat, pendant ou après la création de ce dernier :

`initial-cluster-state` : Indique si le nœud est ajouté dans le cadre de la création d'un agrégat (`new`) ou dans un agrégat pré-existant (`existing`).

`initial-advertise-peer-urls` : URLs locales complètes des ports d'écoute communiqués aux autres nœuds lors de la création de l'agrégat etcd. Par exemple : `http://10.0.0.11:2380`. Généralement identique à `listen-peer-urls` sauf quand la résolution DNS[10] est utilisée.

`initial-cluster` : Liste des nœuds attendus lors de la création de l'agrégat etcd. Regroupe toutes les adresses consacrées à la communication entre les nœuds composants l'agrégat. Chaque

[10] https://etcd.io/docs/v3.3/op-guide/clustering/#bootstrap-the-etcd-cluster-using-dns

adresse doit être associée au nom du nœud correspondant par une égalité. Par exemple : `e1=http://10.0.0.11:2380,e2=http://10.0.0.12:2380,e3=http://10.0.0.13:2380`

`initial-cluster-token` : Jeton d'identification utilisé lors de la création de l'agrégat etcd. Ce jeton doit être unique et utilisé une seule fois. Vous pouvez utiliser n'importe quelle chaîne de caractère. Par exemple : `token-01`

`enable-v2` : Activation de l'API v2 si positionné à `true` (déprécié en v3.5).

La liste complète des paramètres est disponible à l'adresse suivante : https://etcd.io/docs/v3.5/op-guide/configuration/#command-line-flags

4.4.4 Services etcd

Configurations employées par les principaux services etcd:

- Service `etcd.service`
- Variables d'environnements…
- … chargées depuis un fichier :
  - `/etc/etcd/etcd.conf` (paquet PGDG Red Hat & dérivées)
  - `/etc/default/etcd` (Debian et dérivées)
- Installation manuelle
  - créer le fichier de service
  - adapter la configuration

Installation par les paquets :

Avec les paquets, les services `etcd` chargent la configuration d'etcd en tant que variables d'environnements lues depuis un fichier. Par défaut, ce fichier est `/etc/etcd/etcd.conf` pour le paquet PGDG pour Red Hat et dérivés, et `/etc/default/etcd` pour Debian/Ubuntu et leurs dérivés. Les paramètres doivent donc y être renseignés avec leur format de variable d'environnement.

Installation manuelle :

Si vous avez installé etcd manuellement, vous devrez créer vous-même votre fichier de service. Voici un exemple :

```
sudo tee /etc/systemd/system/etcd.service <<_EOF_
[Unit]
Description=Etcd Server
After=network.target

[Service]
User=etcd
Type=notify
```

```
WorkingDirectory=/var/lib/etcd/
EnvironmentFile=/etc/etcd/etcd.conf
ExecStart=/usr/bin/etcd
Restart=on-failure
LimitNOFILE=65536

[Install]
WantedBy=multi-user.target
_EOF_

sudo systemctl daemon-reload
```

Créer et compléter le fichier de configuration :

```
sudo mkdir /etc/etcd
sudo tee /etc/etcd/etcd.conf <<_EOF_
ETCD_NAME="[…]"
ETCD_DATA_DIR="[…]"

ETCD_LISTEN_PEER_URLS="[…]"
ETCD_LISTEN_CLIENT_URLS="[…]"
ETCD_ADVERTISE_CLIENT_URLS="[…]"

ETCD_INITIAL_ADVERTISE_PEER_URLS="[…]"
ETCD_INITIAL_CLUSTER="[…]"
ETCD_INITIAL_CLUSTER_TOKEN="[…]"
ETCD_INITIAL_CLUSTER_STATE="new"
_EOF_
```

4.4.5 Démarrage du cluster

- `systemctl start etcd`
  - attente du quorum dès démarrage
- Premier contact avec `etcdctl`
- Création automatique du `data-dir`

Une fois etcd installé et configuré (en accord avec le service utilisé), il est possible de démarrer le service sur tous les nœuds de l'agrégat.

Que ce soit sous RedHat, Debian ou via l'installation manuelle décrite plus haut, la commande est la suivante :

```
systemctl start etcd
```

Les services systemd `etcd` ici présentés sont tous configurés avec le paramètre `Type=notify`. Ce paramètre implique que le démon `etcd` envoie un signal au gestionnaire de service Systemd une fois

démarré correctement. Les démons `etcd` envoient cette notification dès que l'agrégat est capable d'accepter les lectures/écritures des clients, donc lorsque qu'au moins un nœud est déclaré *leader*, donc après que le quorum du cluster a été atteint.

Tant qu'aucun *leader* n'est disponible, le statut du service stagne à `activating` :

```
# systemctl status etcd
● etcd.service - etcd - highly-available key value store
     Loaded: loaded (/lib/systemd/system/etcd.service; disabled; preset: enabled)
     Active: activating (start) since Mon 2024-03-11 14:13:18 UTC; 58s ago
[…]
```

Ci-dessus, le service est démarré depuis 58 secondes, mais aucun *leader* n'a encore été élu. Une fois suffisamment de nœuds démarrés pour atteindre le quorum et que l'un d'eux remporte l'élection, les démons `etcd` signalent à Systemd que le service est devenu disponible et son statut passe alors à `active` :

```
# systemctl status etcd
● etcd.service - etcd - highly-available key value store
     Loaded: loaded (/lib/systemd/system/etcd.service; disabled; preset: enabled)
     Active: active (running) since Mon 2024-03-11 14:14:26 UTC; 4s ago
[…]
```

Il est désormais possible d'interroger l'agrégat etcd, par exemple sur sa santé et son statut :

```
$ etcdctl endpoint --cluster health
http://[…]:2379 is healthy: successfully committed proposal: took = 1.25561ms
http://[…]:2379 is healthy: successfully committed proposal: took = 2.151457ms
http://[…]:2379 is healthy: successfully committed proposal: took = 2.264859ms

$ etcdctl --write-out=table endpoint --cluster status
+-----------------+------------------+---------+---------+-----------+-[…]
|    ENDPOINT     |        ID        | VERSION | DB SIZE | IS LEADER | […]
+-----------------+------------------+---------+---------+-----------+-[…]
| http://[…]:2379 | 2d43bd9a99e8f81c |  3.4.23 |   20 kB |     false | […]
| http://[…]:2379 | 66a8f19c2accae34 |  3.4.23 |   20 kB |      true | […]
| http://[…]:2379 | 738403184a12712e |  3.4.23 |   20 kB |     false | […]
+-----------------+------------------+---------+---------+-----------+-[…]
```

Pour récupérer ces informations, `etcdctl` se connecte par défaut sur le port client d'etcd sur l'interface `localhost` (`127.0.0.1:2379`). Si vous n'avez pas configuré vos démons pour écouter sur cette interface (paramètre `listen-client-urls`), vous pouvez préciser où le CLI doit se connecter en utilisant le paramètre `--endpoints` , par exemple : `--endpoints=etcd0.hapat.vm:2379` .

Au premier démarrage du cluster, chaque membre crée l'arborescence de travail dans le répertoire pointé par le paramètre etcd `data-dir` :

```
~# tree /var/lib/etcd/acme
/var/lib/etcd/acme
└── member
    ├── snap
    │   └── db
```

```
    └── wal
        ├── 0.tmp
        └── 000000000000000-000000000000000.wal

4 directories, 3 files
```

4.5 UTILISATION D'ETCD

4.5.1 Stockage distribué

- Manipulation des clés avec : `get`, `put`, `del` (ou l'API)
- Anciennes versions des clés accessibles… jusqu'au passage d'une purge
- Lectures linéarisées et sérialisées
- Patroni utilise etcd comme serveur de configurations distribuées pour :
  - stocker des informations sur l'état du cluster
  - stocker la configuration dynamique de Patroni

etcd est un moteur de stockage clé-valeur hautement disponible et distribué. Il permet d'ajouter/modifier, supprimer et consulter des clés respectivement avec les commandes `put`, `del` et `get`. Il utilise un système MVCC pour préserver les anciennes valeurs d'une clé.

L'exemple ci-dessous illustre la sémantique des quatre champs suivants :

- **Revision** : global à la base de données etcd, s'incrémente à chaque écriture dans celle-ci ;
- **ModRevision** : révision de la dernière modification de la clé ;
- **CreateRevision** : révision de la création de la clé ;
- **Version** : spécifique à la clé, s'incrémente à chaque mise à jour de celle-ci.

```
$ export ETCDCTL_API=3

$ etcdctl put 'database' 'PostgreSQL'

$ etcdctl -w fields get 'database' | grep -E 'Revision|Key|Version|Value'
"Revision" : 246
"Key" : "database"
"CreateRevision" : 246
"ModRevision" : 246
"Version" : 1
"Value" : "PostgreSQL"

/* rev 247 */
$ etcdctl put 'database' 'PostgreSQL 16'

$ etcdctl -w fields get 'database' | grep -E 'Revision|Key|Version|Value'
"Revision" : 247
"Key" : "database"
"CreateRevision" : 246
"ModRevision" : 247
"Version" : 2
"Value" : "PostgreSQL 16"
```

```
/* rev 248 */
$ etcdctl put 'other_database' 'None'

/* rev 249 */
$ etcdctl put 'database' 'PostgreSQL 17'

$ function list_revs(){
   key=$1
   min=$2
   max=$3
   for (( rev=$min ; rev<=$max; rev++ )); do
      echo -n "\"rev\" : $rev;"
      etcdctl -w fields --rev $rev get $key | grep -E
 ↪  'CreateRevision|Key|Version|Value' | tr "\n" ";" | sed -e
 ↪  "s/CreateRevision/CRev/g";
      echo
   done
}

$ list_revs 'database' 246 249
"rev": 246;"Key" : "database";"CRev" : 246;"Version" : 1;"Value" : "PostgreSQL";
"rev": 247;"Key" : "database";"CRev" : 246;"Version" : 2;"Value" : "PostgreSQL 16";
"rev": 248;"Key" : "database";"CRev" : 246;"Version" : 2;"Value" : "PostgreSQL 16";
"rev": 249;"Key" : "database";"CRev" : 246;"Version" : 3;"Value" : "PostgreSQL 17";
```

Les anciennes versions ne sont pas conservées indéfiniment et peuvent être purgées automatiquement ou manuellement. Ce sujet sera abordé plus en détail dans le paragraphe dédié à la maintenance.

On voit ci-dessous que si une clé est supprimée mais que les révisions précédant sa suppression sont toujours accessibles, on peut lire les valeurs de cette clé. La suppression compte comme une révision et donne lieu à l'écriture d'un enregistrement de type *tombstone* (pierre tombale).

```
/* rev 250 */
$ etcdctl del 'database'

/* rev 251 */
$ etcdctl put 'database' 'Still Pg'

/* purge les révisions avant 248 */
$ etcdctl compact 248
compacted revision 248

$ list_revs 'database' 246 251
"rev": 246;[…]Error: etcdserver: mvcc: required revision has been compacted
"rev": 247;[…]Error: etcdserver: mvcc: required revision has been compacted
"rev": 248;"Key" : "database";"CRev" : 246;"Version" : 2;"Value" : "PostgreSQL 16";
"rev": 249;"Key" : "database";"CRev" : 246;"Version" : 3;"Value" : "PostgreSQL 17";
"rev": 250;
"rev": 251;"Key" : "database";"CRev" : 251;"Version" : 1;"Value" : "Still Pg";
```

Pour son stockage physique, etcd utilise une base BoltDB[11], (une base clé-valeur focalisée sur la simplicité et la fiabilité) via son propre pilote son bbolt[12]. Les révisions sont stockées dans un arbre B+

[11] https://pkg.go.dev/github.com/boltdb/bolt
[12] https://github.com/etcd-io/bbolt

tree[13], où chaque révision contient le delta par rapport à la révision précédente. Pour accélérer certains types de requêtes, un arbre B-tree contenant uniquement les clés et des pointeurs vers l'arbre persistant est stocké en mémoire.

Patroni utilise etcd comme stockage pour y conserver les informations sur l'état du cluster.

De même, la configuration dynamique de Patroni est maintenue dans etcd après la création (*bootstrap*) du cluster. Cette configuration est commune à l'ensemble des membres du cluster Patroni et assure un fonctionnement harmonieux des membres. En plus de la configuration propre à Patroni, elle peut contenir tout ou partie du paramétrage de PostgreSQL.

Contrairement à l'API v2, qui utilisait un espace de stockage avec des répertoires, l'API v3 utilise un espace de stockage « à plat ». Il est cependant toujours possible de spécifier des caractères `/` dans les noms de clés pour simuler une arborescence.

L'ensemble des paramètres écrits par Patroni peut être listé avec la commande :

```
$ etcdctl get --keys-only --prefix /service
/service/acme/config
/service/acme/failover
/service/acme/history
/service/acme/initialize
/service/acme/leader
/service/acme/members/p1
/service/acme/members/p2
/service/acme/members/p3
/service/acme/status
```

L'API d'etcd supporte deux modes de lectures : linéarisées et sérialisées.

Les lectures linéarisées sont des lectures par quorum, elles garantissent que les données lues ont été commitées et donc présentes sur une majorité de serveurs. Ce type de lecture doit passer par le *leader*. Cela induit un coup supplémentaire pour les performances puisque cela empêche l'équilibrage de charge sur les *followers*. Aussi, si la connexion cliente est faite sur un *follower*, elle doit être redirigée vers le *leader*.

Les lectures sérialisées peuvent être faites directement sur les *followers*, c'est l'équivalent des lectures sales (*dirty reads*) sur une base de données relationnelle. Ce genre de lecture est possible même si le cluster a perdu le quorum.

Il est possible de choisir le type de lecture avec `etcdctl` comme suit :

```
etcdctl get --prefix --consistency="l" /
etcdctl get --prefix --consistency="s" /
```

Si deux serveurs etcd sur trois sont arrêtés manuellement, le cluster est marqué comme *unhealthy* :

```
$ etcdctl -w table endpoint --cluster status
[…]
Failed to get the status of endpoint http://[…]:2379 (context deadline exceeded)
[…]
Failed to get the status of endpoint http://[…]:2379 (context deadline exceeded)
+-----------------+[…]-----------+[…]-----------------------+
```

[13] https://en.wikipedia.org/wiki/B%2B_tree

```
|    ENDPOINT     |[…] IS LEADER |[…]        ERRORS         |
+-----------------+[…]-----------+[…]-----------------------+
| http://[…]:2379 |[…]     false |[…] etcdserver: no leader |
+-----------------+[…]-----------+[…]-----------------------+
```

On peut voir que, à la différence des lectures linéarisées, les lectures sérialisées fonctionnent toujours.

```
$ etcdctl get --prefix --consistency="l" key
[…]
Error: context deadline exceeded

$ get --prefix --consistency="s" key
key
myvalue
```

4.5.2 Notion de bail

- Notion de *lease* dans etcd
  - associe une durée de validité (TTL) à une clé
  - le client peut renouveler le bail
  - etcd détruit la clé à l'expiration du bail
- Utilisation par Patroni
  - permet d'identifier le *leader* Patroni et de garantir que la *leader key* expire une fois le TTL expiré

Une fonctionnalité très importante pour Patroni est la possibilité d'associer un bail à une clé. La notion de bail dans etcd est nommée *lease*[14].

Un *lease* est défini par un identifiant unique et une durée de validité appelé TTL (*Time To Live*) et peut être associé à une clé.

Si un *lease* est associé à une clé, cette dernière est automatiquement détruite par etcd à l'expiration de son TTL. Afin de conserver la clé, l'application cliente doit donc régulièrement émettre des *keepalives* pour renouveler le TTL avant son expiration.

Cette mécanique est utilisée par Patroni pour identifier de façon fiable et unique le nœud *leader* au sein du cluster. Lors de l'élection du primaire, chaque nœud Patroni éligible tente de **créer** en premier la clé *leader* avec son propre nom et un bail dont l'expiration est fixée par le paramètre `ttl` de la configuration Patroni.

[14] https://etcd.io/docs/v3.5/learning/api/#lease-api

La requête effectuée par Patroni spécifie explicitement que la clé doit être **créée** et non mise à jour. Grâce à cette contrainte, si plusieurs nœuds Patroni tentent de créer cette clé *leader* en même temps, etcd ne peut en valider qu'une seule et unique. La clé est alors créée avec le nom du nœud dont la requête est validée et celui-ci reçoit une confirmation. Tous les autres nœuds reçoivent une erreur.

Du point de vue de Patroni, le nœud désigné par cette clé *leader* détient le *leader lock* et peut promouvoir son instance PostgreSQL locale. Le processus Patroni est alors en charge de maintenir le bail de sa clé *leader* aussi longtemps que possible.

Si le bail de la clé est résilié ou vient à expirer, celle-ci est donc détruite par etcd. La destruction de cette clé est détectée par l'ensemble des nœuds du cluster et une nouvelle élection Patroni et course au *leader lock* est alors déclenchée.

Ci-après quelques exemples d'utilisation d'un bail avec etcd.

Acquisition d'un bail avec une durée de 300 secondes puis création d'une clé utilisant ce bail :

```
$ export ETCDCTL_API=3

$ etcdctl lease grant 300
lease 33f88b6b082411c8 granted with TTL(300s)

$ etcdctl put sample value --lease=33f88b6b082411c8
OK

$ etcdctl get sample
sample
value
```

Révocation d'un bail et vérification de la clé :

```
$ export ETCDCTL_API=3

$ etcdctl lease revoke 33f88b6b082411c8
lease 33f88b6b082411c8 revoked

$ etcdctl get sample
```

Cas où une clé disparaît à cause d'un non renouvellement du bail :

```
$ export ETCDCTL_API=3

$ etcdctl lease grant 30 # acquérir un bail de 30s
lease 33f88b6b082411d1 granted with TTL(30s)

$ etcdctl put sample value --lease=33f88b6b082411d1 # création de la clé
OK

/* renouvellement dans les temps */

$ etcdctl lease keep-alive --once 33f88b6b082411d1
  lease 33f88b6b082411d1 keepalived with TTL(30)

/* lecture tardive sans renouvellement */

$ sleep 30
```

```
$ etcdctl get sample
```

4.5.3 Unicité des clés

- Création d'une transaction (`etcdctl txn`)
- Création conditionnelle d'une clé
  - action si la clé n'existe pas
  - action si la clé existe
- Permet à Patroni de garantir qu'il n'y a qu'un seul *leader*

Lors de l'élection d'un primaire, plusieurs nœuds peuvent être candidats en même temps, mais un seul d'entre eux doit être élu pour éviter une situation de *split-brain*. Pour les départager, le leader est le nœud qui **crée** la clé. Si la clé existe déjà, toute tentative de création échoue.

L'API v3 d'etcd permet d'implémenter ce comportement via des transactions. Ces transactions se déroulent en trois étapes: `compare`, `success` et `failure`. Si les prédicats de l'étape `compare` réussissent, les commandes de l'étape `success` sont exécutées. Dans le cas contraire, les commandes de l'étape `failure` sont exécutées. Dans tous les cas, les opérations de la transaction sont toutes validées ou annulées de façon atomique.

Ce mécanisme est utilisé par Patroni afin de garantir qu'un seul nœud est capable de créer la clé *leader*, quel que soit le nombre de concurrents simultanés. Schématiquement, sa transaction est la suivante:

```
compare: créer la clé "leader"
success: positionner la valeur leader=<nodename>
failure: ERREUR
```

L'exemple ci-dessous illustre ce comportement. Cinq processus concurrents se disputent la création de la variable `leader` avec l'outil `etcdctl`. Les lignes vides sont importantes, elles permettent de délimiter les trois étapes et de laisser une erreur remonter lors de l'étape `failure`:

```
$ export ETCDCTL_API=3

$ for i in {1..5}
> do {
>  etcdctl txn &
> } <<_EOS_
> create("leader") = "0"
>
> put leader "$i"
>
```

```
>
> _EOS_
> done
FAILURE
SUCCESS

OK
FAILURE
FAILURE
FAILURE

$ etcdctl get leader
leader
4
```

La condition `create("leader") = "i"` vérifie que la clé *leader* a été créée en révision *i*, ou que la clé n'exsite pas quand *i* vaut 0. Voir aussi cette documentation[15] pour la sémantique des transactions, et celle-ci[16] pour des précisions supplémentaires.

Nous constatons que sur les cinq tentatives simultanées, une seule réussit, quatre échouent. Les processus s'exécutant en parallèle, les résultats sont affichés dans un ordre indéfini. Dans cet exemple, il semble que ce soit le quatrième processus qui ait créé la clé (indice `4`).

En conclusion, lors d'une élection Patroni, tous les candidats acceptés se font concurrence pour créer cette clé en premier. Grâce à etcd, nous avons la garantie qu'un seul l'obtient réellement.

[15] https://etcd.io/docs/v3.5/tutorials/how-to-transactional-write/
[16] https://github.com/etcd-io/etcd/blob/main/etcdctl/README.md#txn-options

4.6 MAINTENANCES

4.6.1 Authentification

- Activation de l'authentification requise
  - `etcdctl auth enable`
- `user` : pour l'authentification
  - `etcdctl user [add|del|get|list]`
  - `etcdctl user [grant-role|revoke-role]`
- `role` : conteneur pour les droits, assigné aux utilisateurs
  - `etcdctl role [add|del|get|list]`
  - `etcdctl role [grant-permission|revoke-permission]`

Le protocole v3 utilise gRPC pour son transport. Ce protocole permet à etcd de faire de l'authentification à chaque connexion, et non à chaque requête comme le faisait le protocole v2.

Les méta-données utilisées pour la gestion des droits sont aussi stockées dans etcd, l'algorithme de consensus Raft est donc utilisé pour les gérer.

etcd permet de créer des utilisateurs et des rôles. On peut donner des droits à un rôle que l'on assigne aux utilisateurs. Les opérations de gestions des utilisateurs et rôles peuvent être effectuées avec les sous-commandes des sections `user` et `role` de `etcdctl`.

```
$ export ETCDCTL_API=3

$ etcdctl user add root
Password of root:
Type password of root again for confirmation:
User root created

$ etcdctl user add patroni
Password of patroni:
Type password of patroni again for confirmation:
User patroni created

$ etcdctl user add admin
Password of admin:
Type password of admin again for confirmation:
User admin created

$ etcdctl role add root
```

```
Role root created

$ etcdctl role add patroni
Role patroni created

$ etcdctl user grant-role patroni patroni
Role patroni is granted to user patroni

$ etcdctl user grant-role admin root
Role root is granted to user admin
```

Le rôle `root` peut être attribué à n'importe quel utilisateur. Il a les permissions pour :

- accéder à l'ensemble de données en lecture et écriture ;
- changer la configuration d'authentification ;
- réaliser des tâches de maintenance.

La liste des utilisateurs et rôles peut être consultée avec la sous-commande `list` et le détail de leur configuration peut être consulté avec la sous-commande `get`.

L'authentification doit être activée avec `etcdctl` pour être effective. Sans action supplémentaire, l'accès à etcd est ouvert à tous. Contrairement à l'API V2, aucun utilisateur `guest` n'est créé lors de l'activation, il n'y a donc pas d'action supplémentaire à prévoir de ce côté.

```
$ etcdctl auth enable
[…]
Authentication Enabled
```

Il faut désormais s'authentifier pour utiliser la plupart des fonctions d'etcd.

```
$ etcdctl get --prefix /
[…]
Error: etcdserver: user name is empty
```

Il faut également avoir les droits pour accéder aux clés. La syntaxe est `etcdctl --user <user>:<passwd> [...]` ou simplement `etcd <user> [...]` et dans ce cas le mot de passe est demandé interactivement.

```
$ etcdctl --user root:root put key value
OK

$ etcdctl --user patroni:patroni get
[…]
Error: etcdserver: permission denied

$ etcdctl --user root:root get key
key
value

$ etcdctl --user admin:admin get key
key
value
```

etcd est un stockage clé-valeur ce qui permet de définir des droits sur une plage de données, contrairement à un système de fichiers par exemple. Cela signifie que l'on peut définir des droits sur des clés qui n'existent pas encore.

```
$ etcdctl --user root:root role grant-permission --prefix patroni readwrite /service/

$ etcdctl --user root:root role get patroni
Role patroni updated
Role patroni
KV Read:
    [/service/, /service0) (prefix /service/)
KV Write:
    [/service/, /service0) (prefix /service/)
```

On peut désormais écrire et lire les clés de l'intervalle `[/service/, /service0)` :

```
$ etcdctl --user patroni:patroni put /service/mydata 'is secure'
OK

$ etcdctl --user patroni:patroni get --prefix /service/
/service/mydata
is secure
```

Suivant le besoin, les permissions peuvent être révoquées en retirant la permission au rôle :

```
$ etcdctl --user root:root role revoke-permission --prefix patroni /service/
Permission of range [/service/, /service0) is revoked from role patroni

$ etcdctl --user patroni:patroni get --prefix /service/
[…]
Error: etcdserver: permission denied
```

ou en retirant le rôle à l'utilisateur :

```
$ etcdctl --user root:root user revoke-role admin root
Role root is revoked from user admin

$ etcdctl --user admin:admin get --prefix /service/
[…]
Error: etcdserver: permission denied
```

Les rôles et utilisateurs peuvent également être supprimés :

```
$ etcdctl --user root:root user del patroni

$ etcdctl --user root:root role del patroni
```

4.6.2 Chiffrement des communications

- Communications avec les clients :
  - `--cert-file`
  - `--key-file`
  - `--client-cert-auth`
  - `--trusted-ca-file`
- Communications au sein du cluster etcd :
  - `--peer-cert-file`
  - `--peer-key-file`
  - `--peer-client-cert-auth`
  - `--peer-trusted-ca-file`

Communications avec les clients

Plusieurs options permettent de configurer le chiffrement des communications avec les clients :

`--cert-file` : Certificat utilisé pour le chiffrement des connexions clientes. Si cette option est activée `advertise-client-urls` peut utiliser HTTPS.

`--key-file` : Clé pour le certificat.

`--client-cert-auth` : Quand ce paramètre est activé, etcd va vérifier toutes les requêtes HTTPS reçues pour s'assurer que le certificat client est signé par l'autorité de certification. Si l'authentification est activée, le certificat peut être utilisé uniquement si le nom d'utilisateur spécifié correspond à celui spécifié dans les champs *common name* du certificat.

`--trusted-ca-file` : Certificat de l'autorité de certification.

Communications au sein du cluster

Plusieurs options permettent de configurer le chiffrement des communications entre les membres du cluster `etcd` :

`--peer-cert-file` : Certificat utilisé pour le chiffrement des connexions entre les membres du cluster.

`--peer-key-file` : Clé pour le certificat.

`--peer-client-cert-auth` : Quand ce paramètre est activé, etcd va vérifier toutes les requêtes entre les membres du cluster pour s'assurer que les certificats sont signés par l'autorité de certification.

`--peer-trusted-ca-file` : Certificat de l'autorité de certification.

Autres paramètres

Plusieurs autres paramètres sont valides de manière transverse :

`--cipher-suites` : liste de suite de cipher TLS supportés.
`--tls-min-version` : version minimale de TLS supportée par etcd.
`--tls-max-version` : version maximale de TLS supportée par etcd.

Cas de Patroni

À cause d'un problème dans la librairie gRPC de python, l'authentification par certificat client comportant un *common name* pour l'association avec un utilisateur etcd ne fonctionne pas avec gRPC/Patroni.

4.6.3 Sauvegarde et restauration

- Nécessité de recréer le cluster
- Sauvegarde des données
  - `etcdctl … snapshot save …`
  - copie de la base de données (`$ETCD_DATA_DIRECTORY/member/snap/db`)
- Restauration
  - nouveau cluster
  - `etcdctl snapshot restore …`
- Démarrer le nouveau cluster

Lorsque le quorum est irrémédiablement perdu, il est nécessaire de recréer le cluster etcd.

Sauvegarde

Il est possible de sauvegarder un serveur à chaud grâce à l'outil `etcdctl` . La sauvegarde doit être exécutée sur un membre spécifique. Voici un exemple depuis un serveur sur lequel l'authentification est activée :

```
$ unset ETCDCTL_ENDPOINTS  #  pour éviter un conflit avec --endpoints
$ etcdctl --endpoints http://10.0.0.11:2379 \
        --user root:root \
        snapshot save /tmp/mysnap.etcd.save
{}
{"level":"info","ts":"2023-10-
↪  26T16:10:06.347+0200","caller":"clientv3/maintenance.go:200","msg":"opened
↪  snapshot stream; downloading"}
```

```
{"level":"info","ts":1698329406.3474212,"caller":"snapshot/v3_snapshot.go:127","msg":"fetching
 ↪  snapshot","endpoint":"http://10.0.0.11:2379"}
{"level":"info","ts":"2023-10-
 ↪  26T16:10:06.349+0200","caller":"clientv3/maintenance.go:208","msg":"completed
 ↪  snapshot read; closing"}
{"level":"info","ts":1698329406.3582983,"caller":"snapshot/v3_snapshot.go:142","msg":"fetched
 ↪  snapshot","endpoint":"http://10.0.0.11:2379","size":"53 kB","took":0.11860325}
{"level":"info","ts":1698329406.358513,"caller":"snapshot/v3_snapshot.go:152","msg":"saved","path"
Snapshot saved at /tmp/mysnap.etcd.save
```

Si le serveur est arrêté, il est possible de copier le fichier `$ETCD_DATA_DIRECTORY/member/snap/db`.

Il est possible de consulter des métadonnées sur les snapshots avec la commande suivante :

```
$ etcdctl --write-out=table snapshot status /tmp/mysnap.etcd.save
+----------+----------+------------+------------+
|   HASH   | REVISION | TOTAL KEYS | TOTAL SIZE |
+----------+----------+------------+------------+
| 3e3d7361 |       11 |         20 |      25 kB |
+----------+----------+------------+------------+
```

Restauration

Le fichier sauvegardé peut être restauré avec la commande `etcdctl`. L'ensemble des membres du cluster doivent être restaurés en utilisant la même sauvegarde. L'opération crée un nouveau répertoire de données et écrase certaines métadonnées, comme les identifiants des membres et du cluster. Les membres assument donc une nouvelle identité ce qui les empêche de joindre l'ancien cluster.

Par défaut, `etcdctl` va vérifier l'intégrité de la sauvegarde lors de la restauration. Cela n'est possible qu'avec une sauvegarde réalisée avec `etcdctl`, ce dernier ajoutant à la sauvegarde ces données d'intégrité. Pour restaurer à partir de la copie du fichier de base de données, il est nécessaire d'ajouter l'option `--skip-hash-check`.

La restauration crée un nouveau répertoire de données pour chaque membre.

```
/* sur le membre e1 */
$ sudo chown etcd:etcd /var/lib/etcd

$ sudo -u etcd etcdctl snapshot restore /tmp/mysnap.etcd.save \
  --name e1 \
  --data-dir /var/lib/etcd/new \
  --initial-cluster e1=http://10.0.0.11:2380,\
                    e2=http://10.0.0.12:2380,\
                    e3=http://10.0.0.13:2380 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-advertise-peer-urls http://10.0.0.11:2380
{"level":"info","ts":1701874064.4195023,"caller":"snapshot/v3_snapshot.go:296","msg":"restoring
 ↪  snapshot","path":"/tmp/mysnap.etcd.save","wal-
 ↪  dir":"/var/lib/etcd/new/member/wal","data-dir":"
/var/lib/etcd/new","snap-dir":"/var/lib/etcd/new/member/snap"}
{"level":"info","ts":1701874064.4401946,"caller":"mvcc/kvstore.go:388","msg":"restored
 ↪  last compact revision","meta-bucket-name":"meta","meta-bucket-name-
 ↪  key":"finishedCompactRev","restored-
compact-revision":4}
```

```
{"level":"info","ts":1701874064.4586802,"caller":"membership/cluster.go:392","msg":"added
 ↪  member","cluster-id":"3c425733f8f92ab","local-member-id":"0","added-peer-
 ↪  id":"392a31edc7fd2f21","add
ed-peer-peer-urls":["http://10.0.0.11:2380"]}
{"level":"info","ts":1701874064.4682066,"caller":"membership/cluster.go:392","msg":"added
 ↪  member","cluster-id":"3c425733f8f92ab","local-member-id":"0","added-peer-
 ↪  id":"8db8de0f2f58d643","added-peer-peer-urls":["http://10.0.0.12:2380"]}
{"level":"info","ts":1701874064.4689171,"caller":"membership/cluster.go:392","msg":"added
 ↪  member","cluster-id":"3c425733f8f92ab","local-member-id":"0","added-peer-
 ↪  id":"fdcf639378827052","add
ed-peer-peer-urls":["http://10.0.0.13:2380"]}
{"level":"info","ts":1701874064.5160108,"caller":"snapshot/v3_snapshot.go:309","msg":"restored
 ↪  snapshot","path":"/tmp/mysnap.etcd.save","wal-
 ↪  dir":"/var/lib/etcd/new/member/wal","data-dir":"/
var/lib/etcd/new","snap-dir":"/var/lib/etcd/new/member/snap"}
```

Exécuter ensuite la même commande sur les autres membres en adaptant les adresses. Sur `e2` :

```
$ sudo chown etcd:etcd /var/lib/etcd

$ sudo -u etcd etcdctl snapshot restore /tmp/mysnap.etcd.save \
  --name e2 \
  --data-dir /var/lib/etcd/new \
  --initial-cluster e1=http://10.0.0.11:2380,\
                    e2=http://10.0.0.12:2380,\
                    e3=http://10.0.0.13:2380 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-advertise-peer-urls http://10.0.0.12:2380
```

Sur `e3` :

```
$ sudo chown etcd:etcd /var/lib/etcd

$ sudo -u etcd etcdctl snapshot restore /tmp/mysnap.etcd.save \
  --name e3 \
  --data-dir /var/lib/etcd/new \
  --initial-cluster e1=http://10.0.0.11:2380,\
                    e2=http://10.0.0.12:2380,
                    e3=http://10.0.0.13:2380 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-advertise-peer-urls http://10.0.0.13:2380
```

Vérifier que le fichier de configuration utilisé par le service contient les bonnes informations, puis démarrer le service sur chaque serveur :

```
# systemctl start etcd
```

Vérifier la santé du cluster :

```
$ etcdctl -w table --user root:root endpoint --cluster health
+-----------------+--------+-------------+-------+
|    ENDPOINT     | HEALTH |    TOOK     | ERROR |
+-----------------+--------+-------------+-------+
|  10.0.0.13:2379 |   true | 89.852914ms |       |
|  10.0.0.12:2379 |   true | 64.684492ms |       |
|  10.0.0.11:2379 |   true |  87.22009ms |       |
```

```
+-----------------+--------+-------------+-------+

$ etcdctl -w table --user root:root endpoint --cluster status
+-----------------+------------------+---------+---------+-----------+-[…]
|    ENDPOINT     |        ID        | VERSION | DB SIZE | IS LEADER | […]
+-----------------+------------------+---------+---------+-----------+-[…]
|  10.0.0.11:2379 | 392a31edc7fd2f21 |  3.4.23 |   25 kB |     false | […]
|  10.0.0.12:2379 | 8db8de0f2f58d643 |  3.4.23 |   25 kB |     false | […]
|  10.0.0.13:2379 | fdcf639378827052 |  3.4.23 |   25 kB |      true | […]
+-----------------+------------------+---------+---------+-----------+-[…]
```

Cas particulier de Patroni

Patroni effectue une sauvegarde régulière de sa configuration en cas de problème rendant le cluster etcd complètement inopérant. Il est donc possible de recréer un nouveau cluster etcd à la même adresse, Patroni y recrée sa configuration dès qu'il peut s'y reconnecter. En fonction de la configuration choisie, il sera peut être nécessaire de recréer la configuration de l'authentification au préalable.

4.6.4 Remplacement de membre

Pour remplacer un membre sans risquer de perdre le quorum :

- d'abord retirer l'ancien membre
- puis ajouter le nouveau membre
- et jamais l'inverse !

Si un agrégat est dans un état où il ne peut plus tolérer de panne, l'ajout d'un nœud **avant** de supprimer des nœuds défaillants est **dangereux**. L'ajout d'un nœud va augmenter le quorum minimal pour que le cluster soit fonctionnel. Or, si le nouveau nœud ne parvient pas à s'enregistrer auprès de l'agrégat, suite à une erreur de configuration par exemple, le quorum est alors définitivement perdu !

Pour retirer un membre, ici `e1`, il faut récupérer son identifiant :

```
$ export ETCDCTL_API=3

$ etcdctl --user root:root -w table member list
+------------------+---------+-------+-[…]
|        ID        | STATUS  | NAME  | […]
+------------------+---------+-------+-[…]
| 8766ab400c82bb8b | started |    e1 | […]
| 8db8de0f2f58d643 | started |    e2 | […]
```

```
| fdcf639378827052 | started |    e3 | […]
+------------------+---------+-------+-[…]
```

On peut ensuite retirer le membre du cluster en utilisant son identifiant :

```
$ etcdctl --user root:root member remove 8766ab400c82bb8b
Member 8766ab400c82bb8b removed from cluster 4c539c130865dd95
```

Le nouveau membre peut ensuite être déclaré :

```
$ etcdctl --user root:root member add e1 --peer-urls=http://10.0.0.11:2380
Member fba06dcbe3cdd363 added to cluster 4c539c130865dd95

ETCD_NAME="e1"
ETCD_INITIAL_CLUSTER="e2=http://10.0.0.12:2380,e3=http://10.0.0.13:2380,e=http://10.0.0.11:2380"
ETCD_INITIAL_ADVERTISE_PEER_URLS="http://10.0.0.11:2380"
ETCD_INITIAL_CLUSTER_STATE="existing"
```

Le nœud peut ensuite être réintroduit dans le cluster. Dans ce cas, il s'agit du même serveur, il faut donc supprimer le répertoire de données et mettre à jour la configuration pour spécifier que le cluster existe déjà.

```
# rm -Rf /var/lib/etcd/acme

# grep ETCD_INITIAL_CLUSTER_STATE /etc/default/etcd
ETCD_INITIAL_CLUSTER_STATE="existing"
```

Démarrer le service sur le nouveau membre :

```
systemctl start etcd
```

Vérifier la santé du cluster :

```
$ etcdctl -w table --user root:root endpoint --cluster health
+-----------------+--------+------------+-------+
|    ENDPOINT     | HEALTH |    TOOK    | ERROR |
+-----------------+--------+------------+-------+
|  10.0.0.13:2379 |   true | 9.238291ms |       |
|  10.0.0.12:2379 |   true | 6.268266ms |       |
|  10.0.0.11:2379 |   true | 8.744718ms |       |
+-----------------+--------+------------+-------+

$ etcdctl -w table --user root:root endpoint --cluster status
+-----------------+------------------+---------+---------+-----------+[…]
|    ENDPOINT     |        ID        | VERSION | DB SIZE | IS LEADER |[…]
+-----------------+------------------+---------+---------+-----------+[…]
|  10.0.0.11:2379 | fba06dcbe3cdd363 |  3.4.23 |   25 kB |     false |[…]
|  10.0.0.12:2379 | 169e6880896105ec |  3.4.23 |   25 kB |     false |[…]
|  10.0.0.13:2379 | 9153db243246fd2b |  3.4.23 |   25 kB |      true |[…]
+-----------------+------------------+---------+---------+-----------+[…]
```

4.6.5 Autres tâches de maintenance

- Journal Raft et espace de stockage :
  - rétention
  - quota
  - compactage et défragmentation manuelle et automatique

Rétention du journal Raft

etcd conserve un certain nombre d'entrées Raft en mémoire avant de compacter l'historique Raft. Ce nombre est défini avec `--snapshot-count`. Une fois le seuil atteint, les données sont écrites sur disque et les entrées tronquées de l'historique. Quand un *follower* demande des informations concernant un index déjà compacté, le *leader* est alors forcé de lui renvoyer un snapshot complet des données.

Une valeur importante de `--snapshot-count` peut causer des pressions mémoires sur le serveur et impacter les performances. Il est donc conseillé de ne pas dépasser 100 000. Une valeur trop basse cause un compactage fréquent des données ce qui sollicite beaucoup de *garbage collector* de Go. Suivant les versions, ce paramètre est configuré à 10 000 ou 100 000. Pour un serveur etcd dédié à Patroni, changer ce paramètre est inutile.

Compaction manuelle du journal de Raft et maintenance automatique

etcdctl permet de compacter l'historique Raft manuellement avec la commande `etcdctl compact <revision>`.

Voici un exemple :

```
$ export ETCDCTL_API=3

$ etcdctl -w fields --user root:root put somekey somevalue
"ClusterID" : 271383054466912939
"MemberID" : 4119159706516008737
"Revision" : 12
"RaftTerm" : 2

$ etcdctl -w fields --user root:root put somekey somenewvalue
"ClusterID" : 271383054466912939
"MemberID" : 4119159706516008737
"Revision" : 13
"RaftTerm" : 2

$ etcdctl -w fields --user root:root get --rev 12 somekey
"ClusterID" : 271383054466912939
"MemberID" : 4119159706516008737
"Revision" : 13
"RaftTerm" : 2
"Key" : "somekey"
```

```
"CreateRevision" : 12
"ModRevision" : 12
"Version" : 1
"Value" : "somevalue"
"Lease" : 0
"More" : false
"Count" : 1

$ etcdctl --user root:root compact 13
compacted revision 13

$ etcdctl -w fields --user root:root get --rev 12 somekey
[…]
Error: etcdserver: mvcc: required revision has been compacted
```

etcd dispose d'un système de compactage automatique de l'historique Raft qui connait des améliorations et évolutions à chaque version.

Il est possible de définir une période de rétention en heures avec le paramètre `--auto-compaction-retention`. Le compacteur est lancé automatiquement toutes les heures depuis la version 3.2.0. En cas d'erreur, il se relance après cinq minutes. Pour un serveur etcd dédié à Patroni, cet automatisme est suffisant.

Défragmentation

La compaction de l'historique Raft provoque une fragmentation de la base de donnée etcd. Cette fragmentation crée de l'espace à l'intérieur de la base, utilisable par etcd mais pas rendu au système.

Il est possible de récupérer cet espace avec `etcdctl` :

```
$ export ETCDCTL_API=3
$ etcdctl --user root:root defrag --cluster
Finished defragmenting etcd member[http://10.0.0.11:2379]
Finished defragmenting etcd member[http://10.0.0.12:2379]
Finished defragmenting etcd member[http://10.0.0.13:2379]
```

Étant donné la faible quantité de données et de modifications dans un cluster etcd dédié à Patroni, cette opération n'a pas besoin d'être fréquente. Il faut surveiller que l'espace de stockage reste inférieur au quota défini (voir plus loin).

Space quota

Sans quota, etcd pourrait avoir des performances médiocres si l'espace de donnée est trop important. Il pourrait également remplir l'espace disque disponible, ce qui entrainerait un comportement non prévu de etcd.

Lorsque l'espace de stockage d'etcd dépasse le quota défini, une alerte est déclenchée sur tout le cluster qui passe en mode maintenance. Seules les lectures et les suppressions de clés sont alors autorisées. Il faut faire de la place dans la base ou la défragmenter et acquitter l'erreur de quota pour que le cluster reprenne un comportement normal.

Pour l'exemple, nous allons diminuer la taille du quota de 2 Go à 512 Mo.

```
$ echo "ETCD_QUOTA_BACKEND_BYTES=$((512*1024*1024))" \
  | sudo tee -a /etc/default/etcd

$ systemctl restart etcd
```

On remplit ensuite l'espace de stockage des clés en mettant à jour en permanence la clé `key` :

```
$ while [ 1 ]; do
> dd if=/dev/urandom bs=1024 count=1024 \
>   | ETCDCTL_API=3 etcdctl --user root:root put key || break;
> done
1024+0 records in
1024+0 records out
1048576 bytes (1.0 MB, 1.0 MiB) copied, 0.0859895 s, 12.2 MB/s
[…]
OK
1024+0 records in
1024+0 records out
1048576 bytes (1.0 MB, 1.0 MiB) copied, 0.0344256 s, 30.5 MB/s
{"level":"warn","ts":"2023-12-
 ↪  07T10:10:02.012Z","caller":"clientv3/retry_interceptor.go:62","msg":"retrying of
 ↪  unary invoker failed","target":"endpoint://client-78488c2e-dec2-422c-9099-
 ↪  a9e9fd8eb1d2/10.0.0.11:2379","attempt":0,"error":"rpc error: code =
 ↪  ResourceExhausted desc = etcdserver: mvcc: database space exceeded"}
Error: etcdserver: mvcc: database space exceeded
```

Dans les traces on voit :

```
Dec 07 10:01:02 e2 etcd[1740]: alarm NOSPACE raised by peer 8db8de0f2f58d643
```

Si on consulte la santé du cluster, on voit que le cluster est toujours marqué `healthy`, mais le statut montre une alarme.

```
$ etcdctl -w table --user root:root endpoint status --cluster
+------------------[…]+---------+[…]-------------------------------+
|        ENDPOINT  […]| DB SIZE |[…]            ERRORS             |
+------------------[…]+---------+[…]-------------------------------+
|  http://10.0.0.11[…]|  537 MB |[…] memberID:10212156311862826563 |
|                   […]|         |[…]                alarm:NOSPACE   |
|  http://10.0.0.12[…]|  537 MB |[…] memberID:10212156311862826563 |
|                   […]|         |[…]                alarm:NOSPACE   |
|  http://10.0.0.13[…]|  537 MB |[…] memberID:10212156311862826563 |
|                   […]|         |[…]                alarm:NOSPACE   |
+------------------[…]+---------+[…]-------------------------------+

$ etcdctl -w table --user root:root endpoint health --cluster
+-------------------------+--------+-------------+-------+
|        ENDPOINT         | HEALTH |    TOOK     | ERROR |
+-------------------------+--------+-------------+-------+
|  http://10.0.0.11:2379 |   true | 24.293137ms |       |
|  http://10.0.0.13:2379 |   true | 23.117282ms |       |
|  http://10.0.0.12:2379 |   true | 11.770651ms |       |
+-------------------------+--------+-------------+-------+
```

En consultant la clé, il est possible de trouver le nombre de versions créées :

```
$ etcdctl --user root:root -w json get --prefix key | jq "
>.kvs[] | {
>   key: .key |
>   @base64d,
>   create_revision: .create_revision,
```

```
>   mod_revision: .mod_revision,
>   version: .version
> }"
{
  "key": "key",
  "create_revision": 520,
  "mod_revision": 1020,
  "version": 508
}
```

Pour compacter, il faut récupérer le numéro de révision le plus récent afin de l'utiliser comme référence.

```
$ export ETCDCTL_API=3

$ rev=$(etcdctl endpoint status --write-out="json" \
  | jq ".[].Status.header.revision")

$ etcdctl --user root:root compact $rev
compacted revision 1516
```

Il est ensuite possible de défragmenter :

```
$ etcdctl --user root:root defrag --cluster
Finished defragmenting etcd member[http://10.0.0.11:2379]
Finished defragmenting etcd member[http://10.0.0.12:2379]
Finished defragmenting etcd member[http://10.0.0.13:2379]
```

Et ne pas oublier de lever l'alarme :

```
$ etcdctl alarm disarm
memberID:10212156311862826563 alarm:NOSPACE
```

Note : L'identifiant est fourni sous forme numérique, c'est aussi le cas si vous affichez les informations au format JSON. La plupart des commandes n'acceptent que des `memberid` en hexadécimal. Si on convertit en hexadécimal l'identifiant ci-dessus, on obtient bien `8db8de0f2f58d643`.

La vérification de l'état des nœuds montre ensuite que l'alerte est levée et l'espace récupéré.

```
$ etcdctl -w table --user root:root endpoint status --cluster
+------------------[…]+---------[…]+---------+
|        ENDPOINT  […]| DB SIZE […]| ERRORS |
+------------------[…]+---------[…]+---------+
|  http://10.0.0.11[…]|  1.1 MB […]|         |
|  http://10.0.0.12[…]|  1.1 MB […]|         |
|  http://10.0.0.13[…]|  1.1 MB […]|         |
+------------------[…]+---------[…]+---------+
```

4.6.6 Supervision et métrologie

- endpoint `/debug`
- endpoint `/metrics`
  - destiné à prometheus
  - Grafana dispose d'une source de données Prometheus
  - surveiller
    * présence d'un leader, nombre d'élections
    * statistiques sur les consensus, performances du stockage et du réseau
    * l'utilisation du quota
- endpoint `/health`

endpoint `/debug`

Si les traces sont configurées en mode débogage (`--log-level=debug`), le serveur etcd exporte les informations de débogage sur le endpoint `/debug`. Cela impacte les performances et augmente la quantité de traces produites.

Le endpoint `/debug/pprof` peut être utilisé pour profiler le CPU, la mémoire, les mutex et les routines Go.

endpoint `/metrics`

Le endpoint `/metrics` permet d'exporter des statistiques vers Prometheus. Des alertes par défaut sont disponibles sur github[17].

Graphana supporte Prometheus, on peut donc utiliser les métriques produites par le endpoint pour alimenter de la métrologie. Il existe déjà un dashboard etcd[18] prévu à cet effet.

Ce endpoint permet d'exposer de nombreuses statistiques.

Les statistiques concernant le serveur sont préfixées par `etcd_server_`. On trouve notamment :

- `has_leader` : vaut 1 si le serveur a un *leader* ; Si aucun serveur du cluster n'a de leader_, le cluster est indisponible.
- `leader_changes_seen_total` : comptabilise le nombre de changements de *leader* vu par le serveur. Si ce nombre est élevé cela signifie que le *leader* est instable.
- `proposals_committed_total` : comptabilise le nombre de propositions de consensus commitées sur le serveur. Il est possible d'avoir des valeurs différentes sur les serveurs ; une différence importante indique que le nœud est à la traîne.

[17] https://github.com/etcd-io/etcd/tree/master/contrib/mixin
[18] https://etcd.io/docs/v3.5/op-guide/grafana.json

- `proposals_applied_total` : comptabilise le nombre de propositions de consensus appliqués par le serveur. Les propositions commitées sont appliquées de manière asynchrone. Une différence importante entre demandes commitées et appliquées indique que le serveur est lent ou surchargé.
- `proposals_pending` : comptabilise le nombre de propositions en attente de commit.
- `proposals_failed_total` : comptabilise le nombre de propositions qui ont échoué. Cela peut être dû a deux choses : une élection qui échoue ou un perte de quorum du cluster.

Voici un exemple :

```
$ curl -s -X GET http://10.0.0.11:2379/metrics | grep \
    -e "^etcd_server_has_leader" \
    -e "^etcd_server_leader_changes_seen_total" \
    -e "^etcd_server_proposals_.*\s"
etcd_server_has_leader 1
etcd_server_leader_changes_seen_total 1
etcd_server_proposals_applied_total 2330
etcd_server_proposals_committed_total 2330
etcd_server_proposals_failed_total 4
etcd_server_proposals_pending 0
```

Concernant le stockage, les statistiques sont préfixées par `etcd_disk_`, par exemple :

- `wal_fsync_duration_seconds` : la latence des fsync réalisés par etcd pour persister les entrées de log sur disque (dans les WAL) avant de les appliquer ;
- `backend_commit_duration_seconds` : la latence des fsync appelés par des backends pour commiter sur disque un snapshot incrémental de ses plus récents changements.

Des latences importantes au niveau de ces deux métriques indiquent souvent des problèmes au niveau du disque et peuvent provoquer des élections à cause de *timeouts*.

```
$ curl -s -X GET http://10.0.0.11:2379/metrics \
  | grep -e "^etcd_disk_wal_fsync_duration_seconds" \
         -e "^etcd_disk_backend_commit_duration_seconds"
etcd_disk_backend_commit_duration_seconds_bucket{le="0.001"} 0
[…]
etcd_disk_backend_commit_duration_seconds_bucket{le="+Inf"} 13
etcd_disk_backend_commit_duration_seconds_sum 0.273494819
etcd_disk_backend_commit_duration_seconds_count 13
etcd_disk_wal_fsync_duration_seconds_bucket{le="0.001"} 0
[…]
etcd_disk_wal_fsync_duration_seconds_bucket{le="+Inf"} 55
etcd_disk_wal_fsync_duration_seconds_sum 0.3880873379999999
etcd_disk_wal_fsync_duration_seconds_count 55
```

Il existe aussi des statistiques concernant le quota, comme :

- `etcd_server_quota_backend_bytes` : quota défini sur le cluster ;
- `etcd_mvcc_db_total_size_in_use_in_bytes` : la taille de l'espace de stockage logique ;
- `etcd_mvcc_db_total_size_in_bytes` (ou `etcd_debugging_mvcc_db_total_size_in_bytes` avant la version 3.4) : taille de l'espace de stockage physique en incluant l'espace libre récupérable par une défragmentation.

Par exemple:

```
$ curl -s -X GET http://10.0.0.11:2379/metrics | grep \
  -e "^etcd_server_quota_backend_bytes" \
  -e "^etcd_mvcc_db_total_size_in_bytes" \
  -e "^etcd_mvcc_db_total_size_in_use_in_bytes"
etcd_mvcc_db_total_size_in_bytes 1.081344e+06
etcd_mvcc_db_total_size_in_use_in_bytes 1.077248e+06
etcd_server_quota_backend_bytes 5.36870912e+08
```

Et encore, des statistiques concernant le réseau, préfixées par `etcd_network_`, par exemple :

- `peer_sent_bytes_total` : quantité de données envoyées aux autres serveurs, répartie par ID (en byte).
- `peer_received_bytes_total` : quantité de données reçues des autres serveurs, répartie par ID (en byte).
- `peer_sent_failures_total` : nombre d'échecs d'envoi vers les autres serveurs, réparti par ID.
- `peer_received_failures_total` : nombre d'échecs de réception depuis les autres serveurs, réparti par ID.
- `peer_round_trip_time_seconds` : histogramme des temps d'aller-retours vers les autres membres.

En voici un exemple écourté :

```
$ curl -s -X GET http://10.0.0.11:2379/metrics | grep -e "^etcd_network_.*\s"
etcd_network_active_peers{Local="3e2cecbaad2c97d4",Remote="8db8de0f2f58d643"} 1
etcd_network_active_peers{Local="3e2cecbaad2c97d4",Remote="fdcf639378827052"} 1
etcd_network_client_grpc_received_bytes_total 534
etcd_network_client_grpc_sent_bytes_total 4.199389e+06
etcd_network_disconnected_peers_total{Local="3e2cecbaad2c97d4",Remote="8db8de0f2f58d643"}
 ↪  1
etcd_network_disconnected_peers_total{Local="3e2cecbaad2c97d4",Remote="fdcf639378827052"}
 ↪  1
etcd_network_peer_received_bytes_total{From="0"} 856980
etcd_network_peer_received_bytes_total{From="fdcf639378827052"} 5.39147414e+08
etcd_network_peer_round_trip_time_seconds_bucket{To="8db8de0f2f58d643",le="0.0001"}
 ↪  0
[…]
etcd_network_peer_round_trip_time_seconds_bucket{To="8db8de0f2f58d643",le="3.2768"}
 ↪  794
etcd_network_peer_round_trip_time_seconds_bucket{To="8db8de0f2f58d643",le="+Inf"}
 ↪  794
etcd_network_peer_round_trip_time_seconds_sum{To="8db8de0f2f58d643"}
 ↪  2.489237533999997
etcd_network_peer_round_trip_time_seconds_count{To="8db8de0f2f58d643"} 794
etcd_network_peer_round_trip_time_seconds_bucket{To="fdcf639378827052",le="0.0001"}
 ↪  0
[…]
etcd_network_peer_round_trip_time_seconds_bucket{To="fdcf639378827052",le="3.2768"}
 ↪  794
etcd_network_peer_round_trip_time_seconds_bucket{To="fdcf639378827052",le="+Inf"}
 ↪  794
etcd_network_peer_round_trip_time_seconds_sum{To="fdcf639378827052"}
 ↪  2.558094078999999
```

```
etcd_network_peer_round_trip_time_seconds_count{To="fdcf639378827052"} 794
etcd_network_peer_sent_bytes_total{To="8db8de0f2f58d643"} 428400
etcd_network_peer_sent_bytes_total{To="fdcf639378827052"} 6.036105e+06
etcd_network_snapshot_receive_inflights_total{From="fdcf639378827052"} 0
etcd_network_snapshot_receive_success{From="fdcf639378827052"} 1
etcd_network_snapshot_receive_total_duration_seconds_bucket{From="fdcf639378827052",le="0.1"}
 ↪  0
[…]
etcd_network_snapshot_receive_total_duration_seconds_bucket{From="fdcf639378827052",le="+Inf"}
 ↪  1
etcd_network_snapshot_receive_total_duration_seconds_sum{From="fdcf639378827052"}
 ↪  25.882678753
etcd_network_snapshot_receive_total_duration_seconds_count{From="fdcf639378827052"}
 ↪  1
```

Il est important de superviser etcd car les indisponibilités d'etcd impactent directement la disponibilité des bases de données PostgreSQL contrôlées par Patroni.

health check

Le endpoint `/health` permet de vérifier que le cluster est en bonne santé, c'est-à-dire qu'il a un *leader*.

4.7 QUESTIONS

- C'est le moment !

4.8 QUIZ

https://dali.bo/r57_quiz

4.9 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/r57_solutions.

4.9.1 Raft

Nous allons utiliser le simulateur Raftscope[19].

Les 5 nœuds portent tous le dernier numéro de mandat qu'ils ont connu. Le *leader* est cerclé de noir.

Le *timeout* de chaque *follower* se décrémente en permanence et est réinitialisé quand un message de *heart beat* du *leader* est reçu.

Le journal de chaque nœud est visible sur la droite pour suivre la propagation des informations de chaque *request*.

Il est possible de mettre en pause, d'accélérer ou ralentir.

Les actions se font par clic droit sur chaque nœud :

- *stop* / *resume* / *restart* pour stopper/(re)démarrer un nœud ;
- *time out* sur un nœud provoque une élection ;
- *request* est une interrogation client (à faire uniquement sur le *leader*).

- Observer la première élection et l'échange des *heart beats* par la suite.

- Mettre en défaut le *leader* (*stop*) et attendre une élection.

- Lancer plusieurs écritures (*requests*) sur le *leader*.

- Remettre en route l'ancien *leader*.

- Arrêter deux nœuds, dont le *leader*, et observer le comportement du cluster.
- Qu'en est-il de la tolérance de panne ?

- Arrêter un nœud secondaire (pour un total de 3 nœuds arrêtés).
- Tester en soumettant des écritures au primaire.

- Rallumer un nœud pour revenir à 3 nœuds actifs dont un *leader*.
- Éteindre le *leader*. Tenter de soumettre des écritures.
- Que se passe-t-il ?

[19] https://raft.github.io/raftscope/index.html

4.9.2 Installation d'etcd sous Debian

- Installer etcd sur les 3 nœuds `e1`, `e2` et `e3`.
- Supprimer le nœud etcd créé sur chaque serveur.
- Configurer un cluster etcd sur les 3 nœuds `e1`, `e2` et `e3`.
- Démarrez le cluster etcd

4.9.3 Installation d'etcd sous Rocky Linux 9

- Installer etcd sur les 3 nœuds `e1`, `e2` et `e3`.
- Configurer un cluster etcd sur les 3 nœuds `e1`, `e2` et `e3`.
- Activez et démarrez le cluster etcd

4.9.4 etcd : manipulation (optionnel)

Ces exercices utilisent l'API v3 d'etcd.

But : manipuler la base de données distribuée d'Etcd.

- Depuis un nœud, utiliser `etcdctl put` pour écrire une clef `foo` à la valeur `bar`.
- Récupérer cette valeur depuis un autre nœud.
- Modifier la valeur à `baz`.
- Créer un répertoire `food` contenant les clés/valeurs `poisson: bar` et `vin: blanc`.
- Récupérer toutes les clefs du répertoire `food` en exigeant une réponse d'un quorum.

But : constater le comportement d'Etcd conforme à l'algorithme Raft.

- Tout en observant les logs de etcd et la santé de l'agrégat, procéder au *fencing* du *leader* avec `virsh suspend <nom machine>`.

- Geler le nouveau *leader* de la même manière et voir les traces du nœud restant.

Patroni : Mise en œuvre

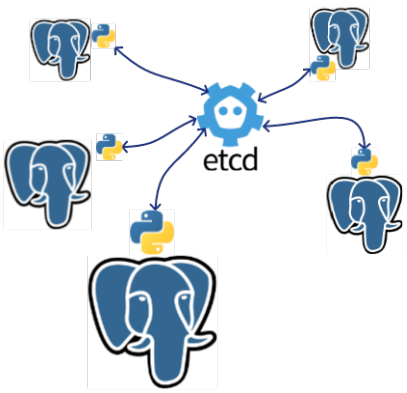


Figure 4/ .1: PostgreSQL

4.10 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.

4.11 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[20].

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
```

[20] *Network Time Protocole*

4.12 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.

Liste des actions disponibles sur le cluster et commandes associées dans `patronictl` :

- superviser (via l'API REST) / observer la topologie :

```
list         List the Patroni members for a given Patroni
topology     Prints ASCII topology for given cluster
```

- manipuler le service (en plus de l'arrêt/démarrage via le service) :

```
restart      Restart cluster member
flush        Discard scheduled events
```

- réaliser des bascules et contrôler la bascule automatique :

```
switchover   Switchover to a replica
failover     Failover to a replica
history      Show the history of failovers/switchovers
pause        Disable auto failover
resume       Resume auto failover
reinit       Reinitialize cluster member
```

- gérer la configuration dynamique de Patroni :

```
show-config  Show cluster configuration
edit-config  Edit cluster configuration
reload       Reload cluster member configuration
```

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

4.12.1 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[21] et pg_basebackup[22] sont utilisés pour construire ou reconstruire les instances secondaires. Nous les décrivons dans le module de formation I4[23].

Cette capacité est étendue grâce à la possibilité d'utiliser des outils de la communauté comme barman[24], pgBackRest[25] ou WAL-G[26].

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.

[21] https://www.postgresql.org/docs/current/app-pgrewind.html
[22] https://www.postgresql.org/docs/current/app-pgbasebackup.html
[23] https://dali.bo/i4_html
[24] https://www.pgbarman.org/
[25] https://pgbackrest.org/
[26] https://github.com/wal-g/wal-g

4.12.2 Bascule automatique

- *split-brain*
- *leader lock*
- *heart beat*
- 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*.

4.12.3 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.

4.12.4 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.

4.12.5 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.

4.12.6 Bootstrap de nœud

- Création du premier nœud
  - initdb
  - pgBackRest depuis sauvegarde PITR
- Création / Reconstruction de réplica
  - pg_basebackup depuis primaire
  - pgBackRest depuis sauvegarde PITR (delta !)

L'opération de *bootstrap* consiste à créer ou recréer l'instance PostgreSQL d'un nœud à partir du nœud primaire, d'une sauvegarde ou avec `initdb` (premier nœud). Par défaut, Patroni créé la première instance avec `initdb` et lance un `pg_basebackup` pour créer les réplicas.

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.

4.12.7 Répartition sur deux sites

- Prévoir la perte d'un des 2 sites
- Tolérance de panne problèmatique pour un cluster etcd à trois serveurs répartis sur deux sites
  - au pire, passage en *read only* de PostgreSQL !

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 des etcd sur le site A
  - si perte du site A : etcd sur le site B n'a pas le quorum, pas de bascule
  - si perte du site B : etcd maintient le quorum, aucun impact sur le primaire
- Cas 2 : primaire sur le site A, majorité des etcd sur le site B
  - si perte du site A : bascule vers le site B
  - si perte du site B : perte du quorum au niveau d'etcd, **PostgreSQL passe en mode *standby* sur le site A**

Dans le cas d'un agrégat multisite, la réponse à cette problématique est d'utiliser au minimum trois sites avec chacun un serveur etcd.

En cas d'isolation réseau d'un des sites, les instances etcd qui s'y trouvent ne peuvent plus y former de quorum, ce qui empêche la plupart des actions dans etcd. Les instances PostgreSQL isolées sur ce site ne peuvent donc y être disponible qu'en *read only* (*standby*).

Les deux serveurs etcd restants (un par site) continuent à communiquer entre eux et conservent ainsi le quorum, ce qui leur permet de fonctionner normalement. Le leader du cluster Patroni est donc fonctionnel, l'instance PostgreSQL qu'il manage peut être démarrée ou maintenue sur l'un de ces deux sites en tant que primaire sans risque de *split-brain*.

En conclusion : Il faut considérer séparément la haute disponibilité de etcd et des instances PostgreSQL. En effet, la disponibilité de etcd est un pré-requis à la mise en œuvre d'une mécanique de bascule automatique fiable et anti-*split-brain*. La répartition des serveurs doit donc permettre à etcd de continuer à fonctionner en cas de perte d'un site.
La répartition des instances PostgreSQL sur les sites doit permettre de respecter la tolérance de panne définie lors de la phase d'initialisation du projet de HA.

Standby cluster

Il est possible de créer un second cluster indépendant sur un second site qui réplique les données depuis le cluster principal. On a alors deux clusters etcd distincts, ce qui nous affranchit des problèmes décrits précédemment. Chaque site héberge un cluster Patroni indépendant, l'un d'eux étant entièrement en mode *standby*, aussi appelé *standby cluster*. La réplication entre site peut être mise en place via la configuration de Patroni. En cas de perte du premier site, une promotion manuelle doit être effectuée sur le second site.

4.12.8 Répartition sur trois sites

- Placer 1 etcd sur chacun des 3 sites autorise une tolérance de panne d'un site
- Changement de site lors d'une bascule
- Perte de deux sites

Quorum de sites

La présence de trois sites permet de disposer de suffisamment de nœuds pour maintenir le fonctionnement de etcd. Ce qui permet à Patroni de fonctionner et déclencher des bascules.

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.

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.

Le tag `failover_priority` ne permet pas de régler ce problème, car il permet de privilégier un nœud par rapport aux autres nœuds **qui ont un lag équivalent**. De plus, rien ne permet de mettre à jour cette liste dynamiquement en fonction du datacenter où se trouve les instances.

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 cluster** avec les nœuds restants.

Si la tolérance de panne du DCS est dépassée, il est toujours possible de désactiver Patroni et promouvoir manuellement une instance sur le site restant, et si besoin d'y raccrocher les instances *standbys* restantes afin de maintenir le service.

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.

4.13 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[27].

4.13.1 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
```

[27] 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[28].

4.13.2 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
```

4.13.3 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`.

[28] 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

4.14 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[29] 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.

[29] 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`.

4.14.1 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/p0
/service/acme/members/p1
/service/acme/status
```

4.14.2 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:
    - 10.0.0.11:2379
    - 10.0.0.12:2379
    - 10.0.0.13:2379
  username: patroniaccess
  password: secret
[…]
```

4.14.3 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
primary_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`, au minimum `3s`. Cela correspond au temps nominal de la boucle de vérification. Un problème au niveau du DCS ou du réseau supérieur ou égal à cette valeur provoque un *demote* du leader.

`ttl` : Temps avant l'initialisation d'un *failover*, par défaut `30s`, au minimum `20s`. 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`, au minimum `3s`.

`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.

`primary_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.

`primary_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. Les trois valeurs possibles pour ce paramètre sont : `on`, `quorum`, et `off`.

`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.

Depuis la version 3.2.0, Patroni s'assure que la règle suivante est valide et que `ttl`, `retry_timeout` et `ttl` respectent les valeurs minimales décrites plus tôt :

```
loop_wait + 2 * retry_timeout <= ttl
```

Si cette règle n'est pas respectée, Patroni ajuste le paramétrage en préférant réduire `loop_wait`, à moins qu'il soit déjà à sa valeur minimale, dans ce cas il diminue `retry_timeout` :

- **SI** `min_loop_wait + 2 * retry_timeout > ttl` **ALORS**
  - `loop_wait = min_loop_wait = 1s`
  - `retry_timeout = (ttl - min_loop_wait) // 2`
- **SINON SI** `loop_wait + 2 * retry_timeout > ttl` **ALORS**
  - `loop_wait = ttl - 2 * retry_timeout`

4.14.4 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`
- scripts `bootstrap.post_bootstrap` et `bootstrap.post_init`

La section `bootstrap.method` permet de décrire la manière dont la première instance du cluster doit être créée. La méthode `initdb` est la méthode par défaut, mais il est possible d'utiliser d'autres commandes[30].

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
[…]
```

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

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 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_leader` : 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_leader: True
    no_params: True
  basebackup:
    - 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.

La section `bootstap.users` permet de créer et configurer des rôles PostgreSQL supplémentaires. Elle est dépréciée en 3.2.0 et sera supprimée en 4.0.0.

```
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.

4.14.5 Configuration de PostgreSQL

- Configuration de l'agrégat et des instances
- Configuration dynamique initialisée depuis la section `bootstrap.dcs.postgresql` (`parameters`, `pg_hba`, `pg_ident`)
- Configuration statique 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.

`member_slots_ttl` Durée de rétention des slots de réplication lorsqu'un nœud est arrêté et que sa clé disparaît du DCS. Si cette durée est à zéro, l'ancien comportement est conservé : le slot est supprimé lorsque la clé disparaît du DCS. La valeur par défaut est : `30min`.

`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: 10.0.0.21:5432
  data_dir: /var/lib/pgsql/17/data
  bin_dir: /usr/pgsql-17/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 10.0.0.21/32 scram-sha-256
    - host replication replicator 10.0.0.22/32 scram-sha-256
    - host replication replicator 10.0.0.23/32 scram-sha-256
  pg_ident:
    - superusermapping root postgres
    - superusermapping dba postgres
```

4.14.6 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.

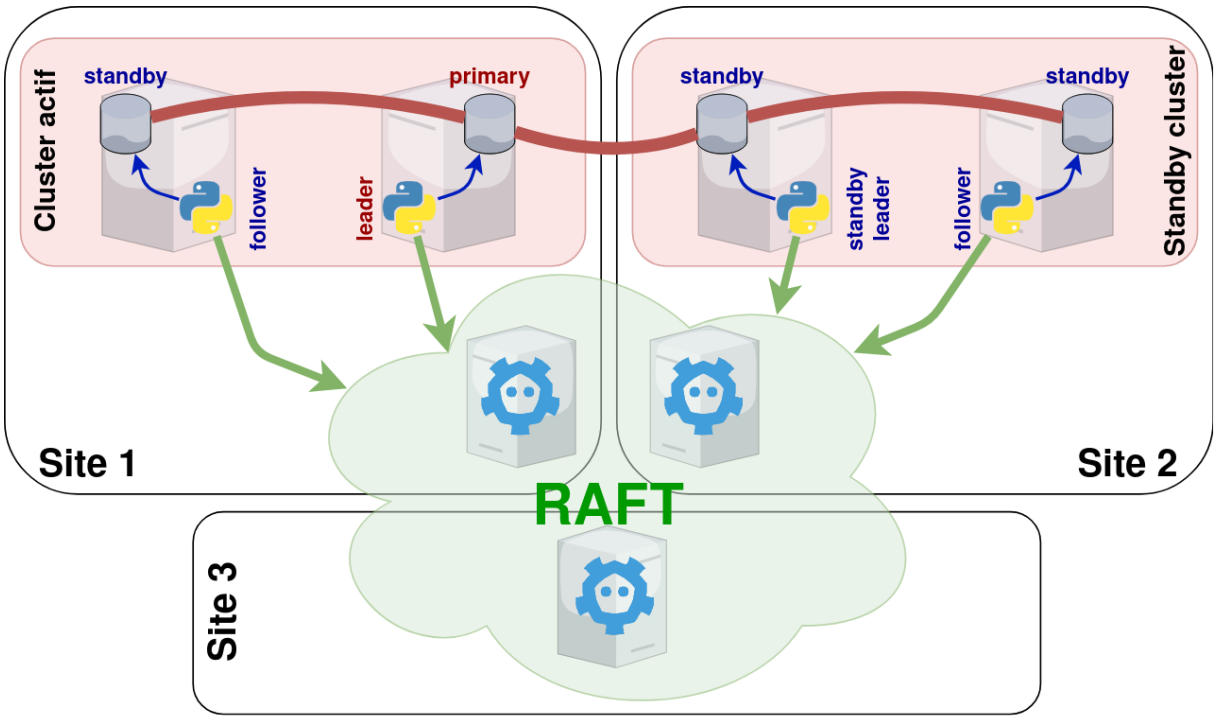


Figure 4/ .2: Standby cluster Patroni

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[31]. 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.

[31] 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: 10.0.0.21 # p1
      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,
  "primary_start_timeout": 300,
  "postgresql": {
    "use_pg_rewind": false,
    "use_slots": true,
    "parameters": {
      "archive_mode": "on",
      "archive_command": "/bin/true"
    }
  },
  "standby_cluster": {
    "host": "10.0.0.21",
    "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     | 10.0.0.23 | Standby Leader | running   |  1 |           |
| p4     | 10.0.0.24 | Replica        | streaming |  1 |         0 |
+--------+-----------+----------------+-----------+----+-----------+
```

La promotion d'un agrégat de secours se fait en supprimant la section `standby_cluster` de la configuration dynamique du cluster.

Il est possible de réaliser un switchover manuel entre deux les deux clusters :

- passer le cluster initial en mode maintenances afin d'éviter une bascule quand la primaire est arrêtée ;
- stopper la primaire, puis ses standby afin de s'assure que toutes les modifications sont répliquées sur tous les clients de réplication ;
- supprimer la section `standby_cluster` du cluster de secours, le *standby leader* devient leader ;
- modifier la configuration du cluster initial en y ajoutant la section `standby_cluster`
- retirer le mode maintenance ;
- démarrer les serveurs du cluster initial.

4.14.7 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 et 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 présence 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
```

4.14.8 Réplication synchrone et asynchrone

- Objectif : ne pas perdre de données en cas de bascule
- Patroni :
  - `synchronous_mode: "{on|off|quorum}"`
  - `synchronous_node_count`
  - `synchronous_mode_strict: "{on|off}"`
- PostgreSQL :
  - `synchronous_commit = "on"`
  - `synchronous_standby_name`

Comme dans PostgreSQL, le mode de réplication par défaut dans Patroni est le mode asynchrone. Il est possible d'activer le mode synchrone en configurant `synchronous_mode` à `on` ou `quorum`. Ce mode permet de limiter les risques de perte de données en cas de bascule non programmée. Pour cela, Patroni n'autorise que les serveurs candidats à la réplication synchrone et le leader à participer aux votes.

Le facteur de réplication synchrone `synchronous_node_count` définit le nombre de standby synchrones qui doivent recevoir les modifications avant de confirmer à l'utilisateur que ses données ont été commitées.

- `synchronous_mode: "on"`

  Avec cette configuration, Patroni active la réplication synchrone et constitue une liste d'instances candidates à partir de la liste des instances disponibles. Cette liste est réévaluée à chaque boucle HA en fonction du lag des serveurs, de leur statut et de leur configuration.

  Sur un cluster à trois nœuds : `p1`, `p2`, `p3` ; le paramétrage suivant :

  ```
  synchronous_mode: on
  synchronous_node_count: 1
  ```

  ajoute l'entrée suivante dans le DCS :

```
{"leader":"p1","quorum":0,"sync_standby":"p2"}
```

et la configuration suivante dans l'instance PostgreSQL sur le leader `p1` :

```
           name            | setting
---------------------------+---------
 synchronous_commit        | on
 synchronous_standby_names | p2
```

Avec `patronictl`, on voit que le statut de la standby est `Sync Stanbdy` :

```
+ Cluster: patroni-demo (7421168130564934130) --+----+…
| Member | Host      | Role         | State     | TL |…
+--------+-----------+--------------+-----------+----+…
| p1     | 10.0.0.21 | Leader       | running   |  7 |…
| p2     | 10.0.0.22 | Sync Standby | streaming |  7 |…
| p3     | 10.0.0.23 | Replica      | streaming |  7 |…
+--------+-----------+--------------+-----------+----+…
```

Si `synchronous_node_count` est égal à 2, la valeur de quorum est décrémentée de 1 dans l'entrée `/sync` du DCS :

```
{"leader":"p1","quorum":0,"sync_standby":"p2, p3"}
```

et `synchronous_standby_names` devient :

```
           name            | setting
---------------------------+---------
 synchronous_standby_names | 2 (p2,p3)
```

Note : `2 (p2, p3)` est équivalent à `FIRST 2 (p2,p3)`

La liste des serveurs peut être influencée par les tags `nosync` et `nostream` pour lesquels Patroni va retirer un serveur de la liste des serveurs candidats à la réplication synchrone, et, si c'est nécessaire, modifier la valeur du paramètre `synchronous_standby_names` dans PostgreSQL. En effet, avec ce mode de réplication, Patroni ne liste que les standby synchrone (pas les candidats).

Le paramètre `failover_priority` n'influence pas cette liste.

- `synchronous_mode: "quorum"`

  Avec cette configuration, Patroni active la réplication synchrone et constitue une liste d'instances candidates à partir de la liste des instances disponibles. Cette liste est réévaluée à chaque boucle HA en fonction de leur statut et de leur configuration.

  Sur un cluster à trois nœuds : `p1`, `p2`, `p3` ; le paramétrage suivant :

  ```
  synchronous_mode: quorum
  synchronous_node_count: 1
  ```

  ajoute l'entrée suivante dans le DCS :

  ```
  {"leader":"p1","quorum":1,"sync_standby":"p2, p3"}
  ```

et la configuration suivante dans l'instance PostgreSQL sur le leader `p1` :

```
           name            | setting
---------------------------+---------
 synchronous_commit        | on
 synchronous_standby_names | ANY 1 (p2,p3)
```

Avec `patronictl`, on voit que le statut de la standby est `Sync Stanbdy` :

```
+ Cluster: patroni-demo (7421168130564934130) ----+----+…
| Member | Host      | Role           | State     | TL |…
+--------+-----------+----------------+-----------+----+…
| p1     | 10.0.0.21 | Leader         | running   |  7 |…
| p2     | 10.0.0.22 | Quorum Standby | streaming |  7 |…
| p3     | 10.0.0.23 | Quorum Standby | streaming |  7 |…
+--------+-----------+----------------+-----------+----+…
```

La liste des serveurs peut être influencée par les tag `nosync` et `nostream` pour lesquels Patroni va retirer un serveur de la liste des serveurs candidats à la réplication synchrone et modifier la valeur du paramètre `synchronous_standby_names` dans PostgreSQL.

Le paramètre `failover_priority` n'influence pas cette liste.

Patroni favorise la haute disponibilité du service. Il lui est possible de dégrader la réplication synchrone en réduisant le nombre de nœuds requis (`synchronous_node_count`) jusqu'à zéro. C'est pour cette raison que Patroni reconstruit la liste des serveurs candidats à chaque boucle HA et change la configuration de PostgreSQL en fonction. Cela a l'avantage d'éviter de bloquer les `COMMIT` sur le leader pendant trop longtemps en cas de perte d'une standby synchrone. Il faut juste attendre la prochaine boucle HA pour que le standby soit retirée de la liste, on attendra donc au pire `ttl` secondes et en moyenne `loop_wait / 2` secondes.

S'il ne reste plus de standby synchrone, en cas de panne du leader, le failover n'est pas possible.

Pour empêcher la désactivation du mode synchrone et garantir qu'en cas de panne aucun commit n'est perdu, Patroni dispose de l'option `synchronous_mode_strict` qui positionne le paramètre `synchronous_standby_names` à `*` dans PostgreSQL quand il n'y a plus assez de candidats. Cela bloque les écritures en attendant le retour d'un *follower* synchrone ce qui provoque une indisponibilité du service.

Note : Il est possible de positionner Patroni avec `synchronous_mode` à `off` et, dans la section `parameters` dédiée à PostgreSQL, de configurer les paramètres `synchronous_commit = on` et `synchronous_standby_names = *`. puisqu'il gère la bascule automatique et peut éviter la promotion d'une standby asynchrone et donc la perte de données.

4.14.9 Journaux applicatifs

- Par défaut dans le fichier de trace système
- Quelques paramètres de la section `log` :
  - `type` : format des traces (`pain` ou `json`)
  - `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 :

`type` : Le format des traces peut être `plain` ou `json`.

`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
[…]
```

4.14.10 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

Depuis Patroni 3.2.0, Patroni utilise une connexion dédiée au serveur PostgreSQL.

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: 10.0.0.21:8009
  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' | jq

{
  "conn_url": "postgres://10.0.0.21:5432/postgres",
  "api_url": "http://10.0.0.21:8008/patroni",
  "state": "running",
  "role": "primary",
  "version": "4.0.4",
  "xlog_location": 1023513512,
  "timeline": 22
}
```

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,
>     "primary_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
> }' 10.0.0.21:8008/config | jq

{
  "loop_wait": 10,
  "primary_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
```

4.14.11 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.

4.14.12 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.

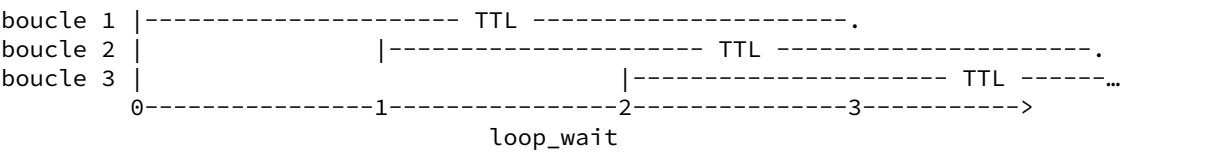
Mécanique et paramétrage

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.

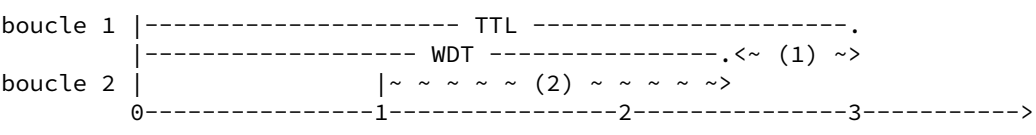


À chaque exécution de la boucle, après avoir déterminé que l'instance est primaire, Patroni doit :

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.



```
                                loop_wait

(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'arrête pas proprement dans les temps, le watchdog arrête toute la machine brutalement, avec risque de perte des données non encore envoyées aux secondaires.

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. La plus petite valeur autorisée de `loop_wait` est `1s` ;
- diminuer `retry_timeout` : rends le système plus sensible aux problèmes réseaux ou charge du DCS. La plus petite valeur autorisée de `retry_timeout` est `3s`.

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` .

4.14.13 Marqueurs d'instance

Patroni supporte différents marqueurs dans la section `tags` :

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

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.

`failover_priority` : Ce tag permet de définir la priorité que doit avoir le nœud en cas d'élection. Si deux nœuds ont reçu et rejoué la même quantité de données, celui qui a la priorité la plus élevée sera choisi. Une priorité inférieure ou égale à zéro est équivalente à `nofailover: true`.

`nostream` : Si ce tag est positionné a `true`, le nœud n'utilisera pas le protocole de réplication mais uniquement le *log shipping* (à condition que `restore_command` soit configuré). Cela désactive la copie et la synchronisation des slots de réplication logique sur ce nœud et toutes ses standby.

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     | 10.0.0.21 | Leader  | running   | […] | montag: mon tag a moi|
|        |           |         |           | […] | noloadbalance: true  |
+--------+-----------+---------+-----------+-[…]-+----------------------+
| p2     | 10.0.0.22 | Replica | streaming | […] |                      |
+--------+-----------+---------+-----------+-[…]-+----------------------+
```

4.14.14 Agrégat multinœud Citus

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

Patroni permet de déployer un cluster multinoœud CITUS[32]. Le CLI `patronictl` a également été adapté pour afficher les informations sur les groupes de serveurs Citus.

4.14.15 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.

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

4.15 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 |
+--------+----------------+--------------+-----------+-----+-----------+
| p1     | 10.0.0.21:5432 | Leader       | running   | 122 |           |
| + p2   | 10.0.0.22:5432 | Sync Standby | streaming | 122 |         0 |
| + p3   | 10.0.0.23:5432 | Sync Standby | streaming | 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 /etc/patroni/config.yml
scope: acme

$ patronictl -c ./acme -d e3://10.0.0.13 topology

$ cat /etc/patroni/config.yml
scope: acme
etcd3:
  hosts:
    - 10.0.00.11:2379
    - 10.0.00.12:2379
    - 10.0.00.13:2379

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

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

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

```
Usage: patronictl [OPTIONS] COMMAND [ARGS]...

  Command-line interface for interacting with Patroni.

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

Commands:
  dsn          Generate a dsn for the provided member, defaults to a dsn...
  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
  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...
```

4.15.1 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 |
+--------+----------------+--------------+-----------+-----+-----------+
| p1     | 10.0.0.21:5432 | Sync Standby | running   | 123 |         0 |
| p2     | 10.0.0.22:5432 | Leader       | streaming | 123 |           |
| p3     | 10.0.0.23:5432 | Sync Standby | streaming | 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 |
+--------+----------------+--------------+-----------+-----+-----------+
| p2     | 10.0.0.21:5432 | Leader       | running   | 123 |           |
| + p1   | 10.0.0.22:5432 | Sync Standby | streaming | 123 |         0 |
| + p3   | 10.0.0.23:5432 | Sync Standby | streaming | 123 |         0 |
+--------+----------------+--------------+-----------+-----+-----------+
```

La colonne `state` indique l'état du serveur, pour une instance secondaire, `streaming` signifie que la réplication en flux est active. `running` signifie que la réplication ne fonctionne pas, soit le serveur est incapable de rattraper son retard, soit il est en train de le faire via le log shipping.

4.15.2 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
    log_min_duration_statement: -1
    wal_keep_size: '1GB'
  use_pg_rewind: true
  use_slots: true
retry_timeout: 10
synchronous_mode: true
synchronous_node_count: 2
ttl: 30
```

4.15.3 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 est fortement conseillé de positionner le paramètre `allow_alter_system` à `off` (à partir de PostgreSQL 17) pour éviter une erreur de manipulation.

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:
    shared_buffers: 64MB
    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
```

4.15.4 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 [p1]:
Candidate ['p2', 'p3'] []: p2
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 |
+--------+----------------+--------------+-----------+-----+-----------+
| p1     | 10.0.0.21:5432 | Leader       | running   | 124 |           |
| p2     | 10.0.0.22:5432 | Sync Standby | streaming | 124 |         0 |
| p3     | 10.0.0.23:5432 | Sync Standby | streaming | 124 |         0 |
+--------+----------------+--------------+-----------+-----+-----------+
Are you sure you want to switchover cluster acme, demoting current leader p1?
[y/N]: y
```

```
2021-05-28 13:48:45.22331 Successfully switched over to "p2"
+ Cluster: acme (6876375338380834518) ----------+-----+-----------+
| Member | Host           | Role    | State     |  TL | Lag in MB |
+--------+----------------+---------+-----------+-----+-----------+
| p1     | 10.0.0.21:5432 | Replica | stopped   |     |   unknown |
| p2     | 10.0.0.22:5432 | Leader  | running   | 124 |           |
| p3     | 10.0.0.23:5432 | Replica | streaming | 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 p2 --candidate p1 --force
Current cluster topology
+ Cluster: acme (6876375338380834518) ---+-----------+-----+-----------+
| Member | Host           | Role         | State     |  TL | Lag in MB |
+--------+----------------+--------------+-----------+-----+-----------+
| p1     | 10.0.0.21:5432 | Sync Standby | streaming | 125 |         0 |
| p2     | 10.0.0.22:5432 | Leader       | running   | 125 |           |
| p3     | 10.0.0.23:5432 | Sync Standby | streaming | 125 |         0 |
+--------+----------------+--------------+-----------+-----+-----------+
2021-05-28 14:03:28.68678 Successfully switched over to "p1"
+ Cluster: acme (6876375338380834518) ----------+-----+-----------+
| Member | Host           | Role    | State     |  TL | Lag in MB |
+--------+----------------+---------+-----------+-----+-----------+
| p1     | 10.0.0.21:5432 | Leader  | running   | 125 |           |
| p2     | 10.0.0.22:5432 | Replica | stopped   |     |   unknown |
| p3     | 10.0.0.23:5432 | Replica | streaming | 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 ['p1', 'p3'] []: p1
Current cluster topology
+ Cluster: acme (6876375338380834518) -+-----------+-----+-----------+
| Member | Host          | Role         | State     |  TL | Lag in MB |
+--------+---------------+--------------+-----------+-----+-----------+
| p1     | 10.0.0.21:5432 | Sync Standby | streaming | 123 |         0 |
| p2     | 10.0.0.22:5432 | Leader       | running   | 123 |           |
| p3     | 10.0.0.23:5432 | Sync Standby | streaming | 123 |         0 |
+--------+---------------+--------------+-----------+-----+-----------+
Are you sure you want to failover cluster acme, demoting current leader p2?
```

```
[y/N]: y

2021-05-28 13:46:47.41163 Successfully failed over to "p1"
+ Cluster: acme (6876375338380834518) ----------+-----+-----------+
| Member | Host           | Role    | State     |  TL | Lag in MB |
+--------+----------------+---------+-----------+-----+-----------+
| p1     | 10.0.0.21:5432 | Leader  | running   | 123 |           |
| p2     | 10.0.0.22:5432 | Replica | stopped   |     |   unknown |
| p3     | 10.0.0.23:5432 | Replica | streaming | 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.

4.15.5 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 p2
+ Cluster: acme (6876375338380834518) -----+----+-----------+
| Member | Host      | Role    | State     | TL | Lag in MB |
+--------+-----------+---------+-----------+----+-----------+
| p1     | 10.0.0.21 | Leader  | running   |  8 |           |
| p2     | 10.0.0.22 | Replica | streaming |  8 |         0 |
+--------+-----------+---------+-----------+----+-----------+
Are you sure you want to reinitialize members p2? [y/N]: y
Success: reinitialize for member p2
```

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

```
$ patronictl reinit acme p1
+ Cluster: acme (7457836878924670834) -----+----+-----------+
| Member | Host      | Role    | State     | TL | Lag in MB |
+--------+-----------+---------+-----------+----+-----------+
| p1     | 10.0.0.21 | Leader  | running   | 22 |           |
| p2     | 10.0.0.22 | Replica | streaming | 22 |         0 |
| p3     | 10.0.0.23 | Replica | streaming | 22 |         0 |
+--------+-----------+---------+-----------+----+-----------+
Error: No replica among provided members
```

4.15.6 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 p1
+ Cluster: acme (6876375338380834518) ---+-----------+-----+-----------+
| Member | Host           | Role         | State     |  TL | Lag in MB |
+--------+----------------+--------------+-----------+-----+-----------+
| p1     | 10.0.0.21:5432 | Leader       | running   | 122 |           |
| p2     | 10.0.0.22:5432 | Sync Standby | streaming | 122 |         0 |
| p3     | 10.0.0.23:5432 | Sync Standby | streaming | 122 |         0 |
+--------+----------------+--------------+-----------+-----+-----------+
Are you sure you want to reload members p1? [y/N]: y
Reload request received for member p1 and will be processed within 10 seconds
```

Les traces montrent l'opération :

```
p2 patroni@acme[21335]: 2021-08-04 15:51:40,473
                                INFO: Reloading PostgreSQL configuration.
p2 patroni@acme[21335]: envoi d'un signal au serveur
p2 patroni@acme[21335]: 2021-08-04 15:51:41,762
                                INFO: Lock owner: p1; I am p2
```

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`.

4.15.7 *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://10.0.0.23:8008/replica?tag_is_candidate=true
HTTP/1.0 200 OK

$ curl -I -s http://10.0.0.23:8008/replica?tag_is_here=true
HTTP/1.0 503 Service Unavailable

$ curl -I -s http://10.0.0.23:8008/replica?tag_doesnt_exist=false
HTTP/1.0 503 Service Unavailable
```

```
$ curl -I -s http://10.0.0.23:8008/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://10.0.0.23:8008/switchover -XPOST \
>      -d'{"leader":"p2", "candidate":"p1"}'

Successfully switched over to "p1"
```

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

```
$ curl -s http://10.0.0.23:8008/switchover -XPOST -d \
    '{"leader":"p1", "candidate":"p3", "scheduled_at":"2023-03-08T11:30+00"}'
Switchover scheduled

$ curl -s http://10.0.0.23:8009/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://10.0.0.23:8008/switchover -XDELETE
scheduled switchover deleted

$ curl -s http://10.0.0.23:8009/restart -XDELETE
scheduled restart deleted
```

4.16 PROXY, VIP ET POOLERS DE CONNEXIONS

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

4.16.1 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`.

4.16.2 Chaîne de connexion

- Chaînes de connexion multihôtes
- 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"
```

4.16.3 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 10.0.0.21:5432 maxconn 100 check port 8008
  server node2 10.0.0.22:5432 maxconn 100 check port 8008
  server node3 10.0.0.23:5432 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 10.0.0.21:5432 maxconn 100 check port 8008
  server node2 10.0.0.22:5432 maxconn 100 check port 8008
  server node3 10.0.0.23:5432 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`.

4.16.4 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 {
        10.0.0.50/24
    }
    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.

4.17 QUESTIONS

– C’est le moment !

4.18 QUIZ

https://dali.bo/r58_quiz

4.19 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/r58_solutions.

4.19.1 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 multiinstance 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 ?

4.19.2 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`.

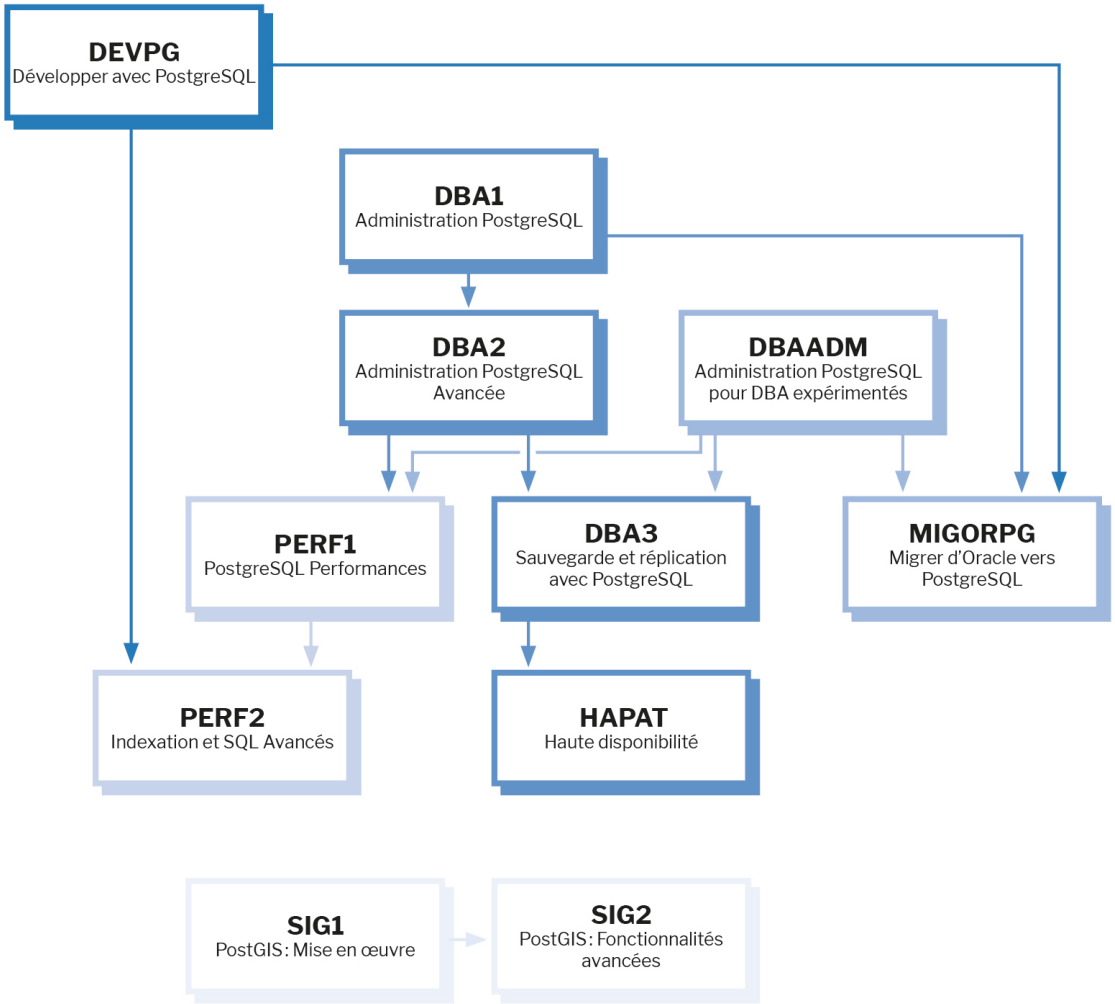
- Modifier les paramètres `shared_buffers` et `work_mem`. Si besoin, redémarrer les nœuds.

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.