Module R57

etcd : Architecture et fonctionnement

24.04

Table des matières

Sur ce document

Formation	Module R57
Titre	etcd : Architecture et fonctionnement
Révision	24.04
PDF	https://dali.bo/r57_pdf
EPUB	https://dali.bo/r57_epub
HTML	https://dali.bo/r57_html
Slides	https://dali.bo/r57_slides
TP	https://dali.bo/r57_tp
TP (solutions)	https://dali.bo/r57_solutions

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

Chers lectrices & lecteurs,

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

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

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

À propos de DALIBO

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

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

[1] mailto:formation@dalibo.com

Remerciements

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

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

Forme de ce manuel

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

Licence Creative Commons CC-BY-NC-SA

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

Marques déposées

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

Versions de PostgreSQL couvertes

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

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

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

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

1/ etcd : Architecture et fonctionnement

1.1 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.

1.2 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)

1.2.1 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.

1.2.2 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 des *followers*
- 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**.

1.2.3 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 consistance 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 consistance entre leurs journaux. Une fois cette position trouvée, le *follower* tronque la suite du journal et applique toutes modifications en provenance du *leader*.

1.2.4 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.

1.2.5 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.

Bien que l'ajout d'un nœud à un cluster de taille impaire semble une bonne idée au premier abord, la tolérance aux pannes est pourtant **pire** : la tolérance de panne reste inchangée, mais comme il y a plus de nœuds, la probabilité de panne est logiquement plus grande.

1.2.6 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

1.2.7 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.

1.2.8 Raft en action

Démo interactive :

- Raft en animation[3]

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*[4].

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

1.3 MÉCANIQUE D'ETCD

- Serveur de stockage distribué par consensus
  - clé/valeur
- Multiples nœuds
- Nécessite 3 nœuds minimum pour avoir une tolérance de panne
- Haute disponibilité et tolérance de panne
- Élection par un quorum

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.

1.3.1 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
- Utilisez l'API v3
  - supportée par Patroni depuis la version 2.0.0 (septembre 2020)

1.3.2 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[5] avec par exemple l'option `--consistency=s` d' `etcdctl`).

[5] 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[6]. 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.

1.3.3 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.

[6] 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*.

1.4 MISE EN ŒUVRE D'ETCD

1.4.1 Contraintes matérielles et logicielles

- VMs 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[7] 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.

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

1.4.2 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 multi-nœud.

```
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[8], 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*
```

[8] 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/
```

1.4.3 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://1.2.3.4:2380`.

`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://1.2.3.4:2379`.

`advertise-client-urls` : URLs locales complètes des ports d'écoute pour les clients communiquées aux autres nœuds. Par exemple : `http://1.2.3.4:2379`. 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://1.2.3.4:2380`.

`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 adresse doit être associée au nom du nœud correspondant par une égalité. Par exemple : `etcd1=http://1.2.3.4:2380,etcd2=http://1.2.3.5:2380,etcd3=http://1.2.3.6:2380`

`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`.

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

1.4.4 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_
```

1.4.5 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 ait 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
```

```
        └── 0000000000000000-0000000000000000.wal

4 directories, 3 files
```

1.5 UTILISATION D'ETCD

1.5.1 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 montre qu'au fur et à mesure des mises à jour, la révision est changée et que les anciennes révisions sont toujours lisibles :

```
$ 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 peut voir ci-dessous que la suppression des anciennes révisions d'une clé ne décrémente pas son compteur de version. Les révisions les plus anciennes sont juste inaccessibles.

On voit également 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[9], (une base clé-valeur focalisée sur la simplicité et la fiabilité) via son propre pilote son bbolt[10]. Les révisions sont stockées dans un arbre B+ tree[11], où chaque révision contient le delta par rapport à la révision précédente. Pour accélérer les accès, un arbre B-tree est stocké en mémoire et pointe vers l'arbre persistant.

[9] https://pkg.go.dev/github.com/boltdb/bolt
[10] https://github.com/etcd-io/bbolt
[11] https://en.wikipedia.org/wiki/B%2B_tree

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 pas 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)
+-----------------+[…]-----------+[…]-----------------------+
|    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
```

1.5.2 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*[12].

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.

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.

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

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
```

1.5.3 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
```

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.

1.6 MAINTENANCES

1.6.1 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.

```
$ 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
```

1.6.2 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.

1.6.3 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 etcd1 \
  --data-dir /var/lib/etcd/new \
  --initial-cluster etcd1=http://10.0.0.11:2380,\
etcd2=http://10.0.0.12:2380,etcd3=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 `etcd2` :

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

$ sudo -u etcd etcdctl snapshot restore /tmp/mysnap.etcd.save \
  --name etcd2 \
  --data-dir /var/lib/etcd/new \
  --initial-cluster e1=http://10.0.0.11:2380,\
etcd2=http://10.0.0.12:2380,etcd3=http://10.0.0.13:2380 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-advertise-peer-urls http://10.0.0.12:2380
```

Sur `etcd3` :

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

$ sudo -u etcd etcdctl snapshot restore /tmp/mysnap.etcd.save \
  --name etcd3 \
  --data-dir /var/lib/etcd/new \
  --initial-cluster etcdc1=http://10.0.0.11:2380,\
etcd2=http://10.0.0.12:2380,etcd3=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.

1.6.4 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 `etcd1`, il faut récupérer son identifiant :

```
$ export ETCDCTL_API=3

$ etcdctl --user root:root -w table member list
+------------------+---------+-------+-[…]
|        ID        | STATUS  | NAME  | […]
+------------------+---------+-------+-[…]
| 8766ab400c82bb8b | started | etcd1 | […]
| 8db8de0f2f58d643 | started | etcd2 | […]
| fdcf639378827052 | started | etcd3 | […]
+------------------+---------+-------+-[…]
```

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 etcd1 --peer-urls=http://10.0.0.11:2380
Member fba06dcbe3cdd363 added to cluster 4c539c130865dd95

ETCD_NAME="etcd1"
ETCD_INITIAL_CLUSTER="etcd2=http://10.0.0.12:2380,etcd3=http://10.0.0.13:2380,etcd1=http://10.0.0.
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 |[…]
+-----------------+------------------+---------+---------+-----------+[…]
```

1.6.5 Autres tâches de maintenance

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

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 confirmé 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. BoltDB stocke l'intégralité de ses données en mémoire afin d'améliorer les performances, c'est un point à considérer si on augmente le quota.

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 etcd2 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 […]|        |
+------------------[…]+---------[…]+---------+
```

1.6.6 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[13].

Graphana supporte Prometheus, on peut donc utiliser les métriques produites par le endpoint pour alimenter de la métrologie. Il existe déjà un dashboard etcd[14] 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 un serveur n'a pas de *leader*, il est indisponible. 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.

[13] https://github.com/etcd-io/etcd/tree/master/contrib/mixin
[14] 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_server_`, 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*.

1.7 QUESTIONS

- C'est le moment !

1.8 QUIZ

https://dali.bo/r57_quiz

1.9 TRAVAUX PRATIQUES

1.9.1 Raft

Nous allons utiliser le simulateur Raftscope[15].

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.

- Arrêter un nœud secondaire (pour un total de 3 nœuds arrêtés).
- Tester en soumettant des écritures au primaire.
- Qu'en est-il de la tolérance de panne ?

- 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 ?

1.9.2 Installation d'etcd sous Debian

[15] https://raft.github.io/raftscope/index.html

- 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

1.9.3 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

1.9.4 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.

1.10 TRAVAUX PRATIQUES (SOLUTIONS)

1.10.1 Raft

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

Les nœuds portent au départ le numéro de mandat 1, il n'y a pas de *leader* :

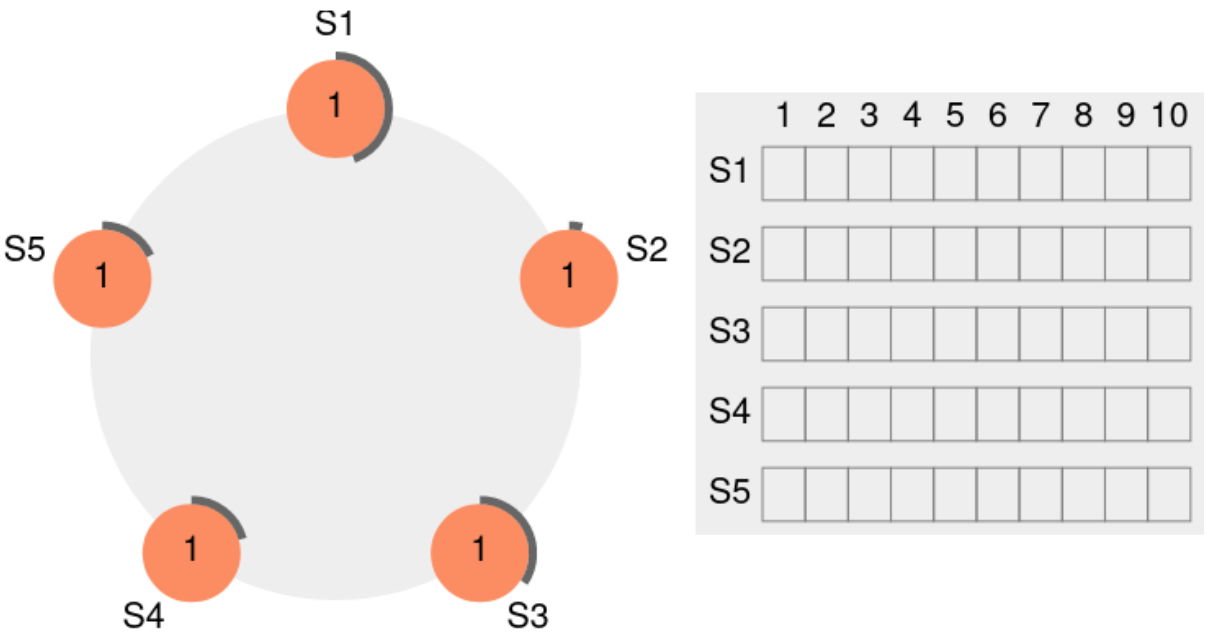


Figure 1/ .1: élection etcd - étape 1

Il faut attendre que l'un des nœuds ait atteint son *timeout* pour qu'il propose une élection. Ici, c'est le nœud S2 qui déclenche l'élection :

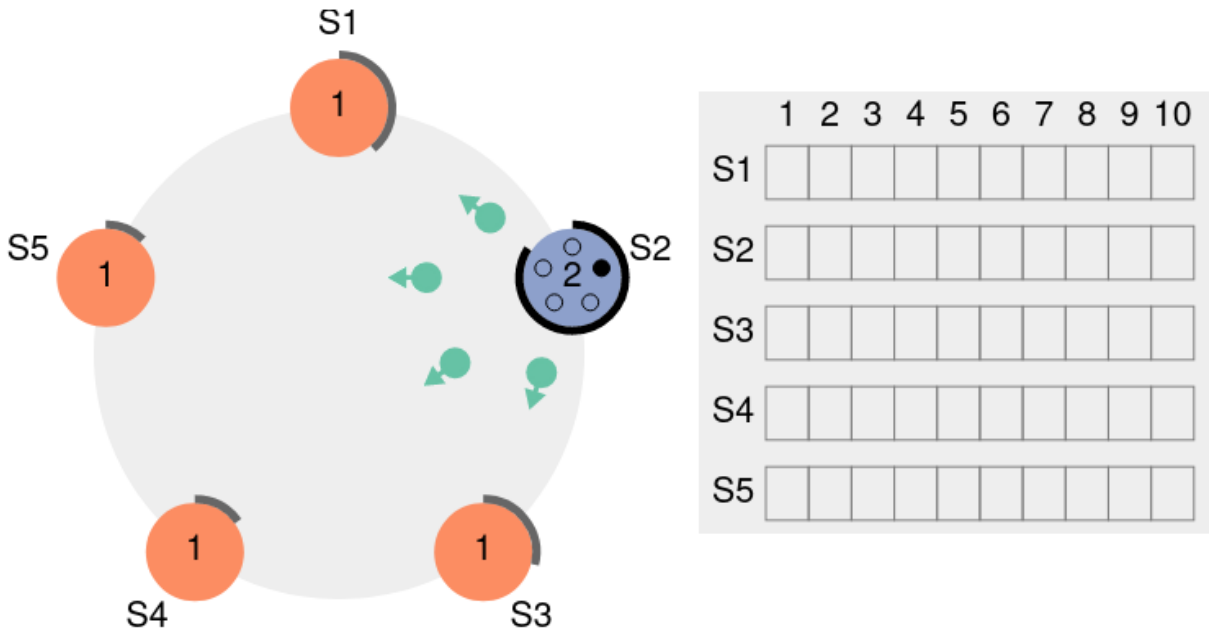


Figure 1/ .2: élection etcd - étape 2

Les petites pastilles sur ce nœud S2 représentent le nombre de votes lui étant favorable. Un vote sur cinq est pour le moment validé : le sien. Les autres nœuds de l'agrégat reçoivent donc la candidature de S2 et y répondent tous favorablement :

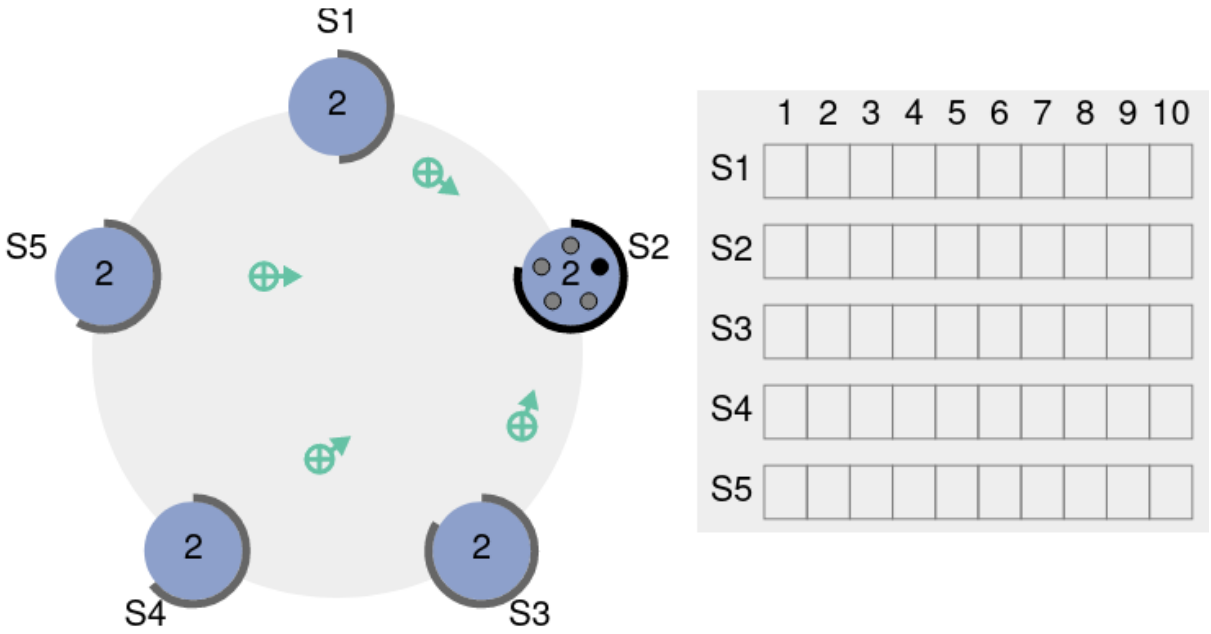


Figure 1/ .3: élection etcd - étape 3

Nous voyons le nombre de vote augmenter au fur et à mesure que S2 les reçoit :

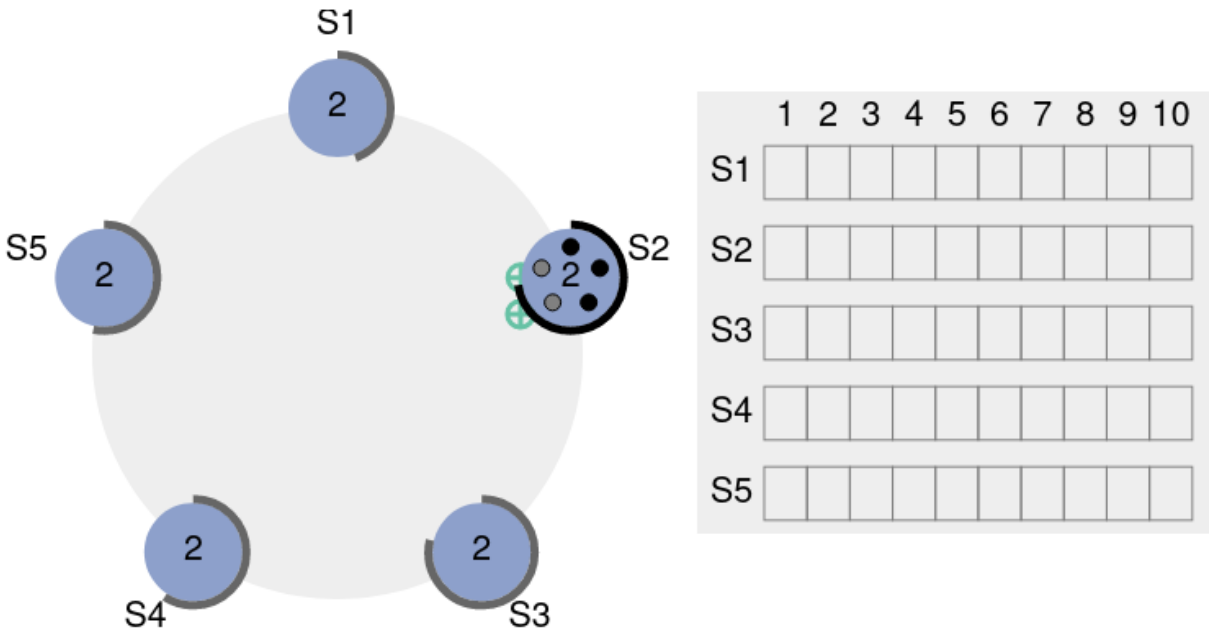


Figure 1/ .4: élection etcd - étape 4

Finalement, S2 remporte l'élection, crée le mandat n°2 et envoie son premier message de *keep alive* :

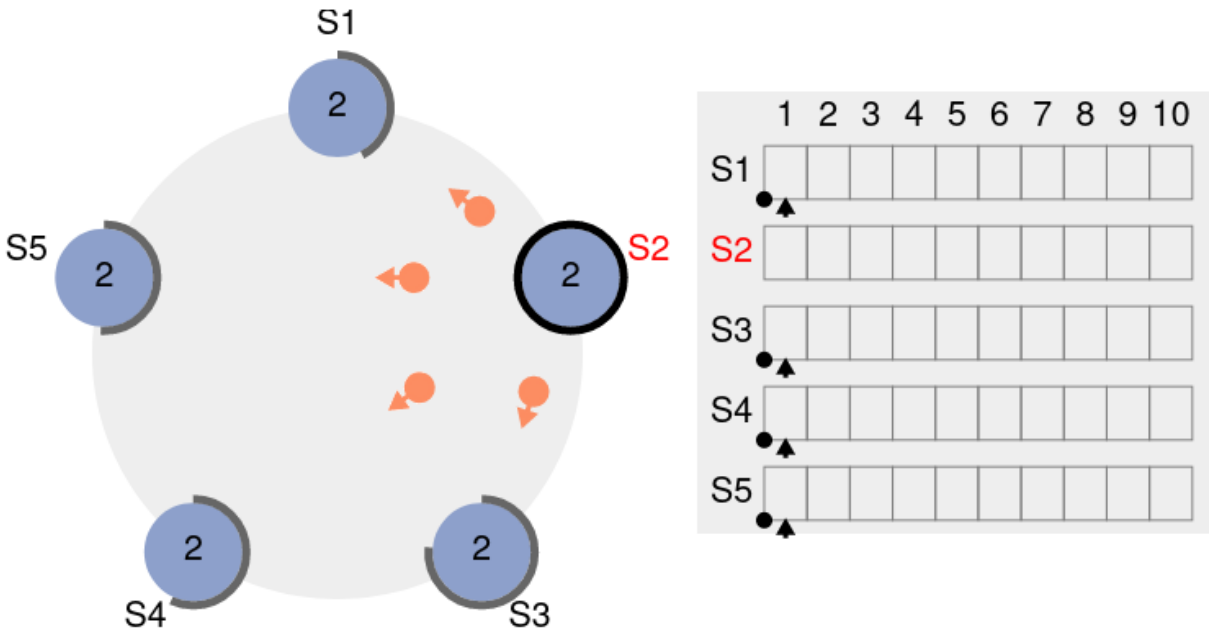


Figure 1/ .5: élection etcd - étape 5

Les autres se raccrochent à lui et entretiennent chacun leur *time out*.

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

Le même phénomène se produit et l'on arrive au mandat 3. Ici, le timeout du nœud S1 expire en premier :

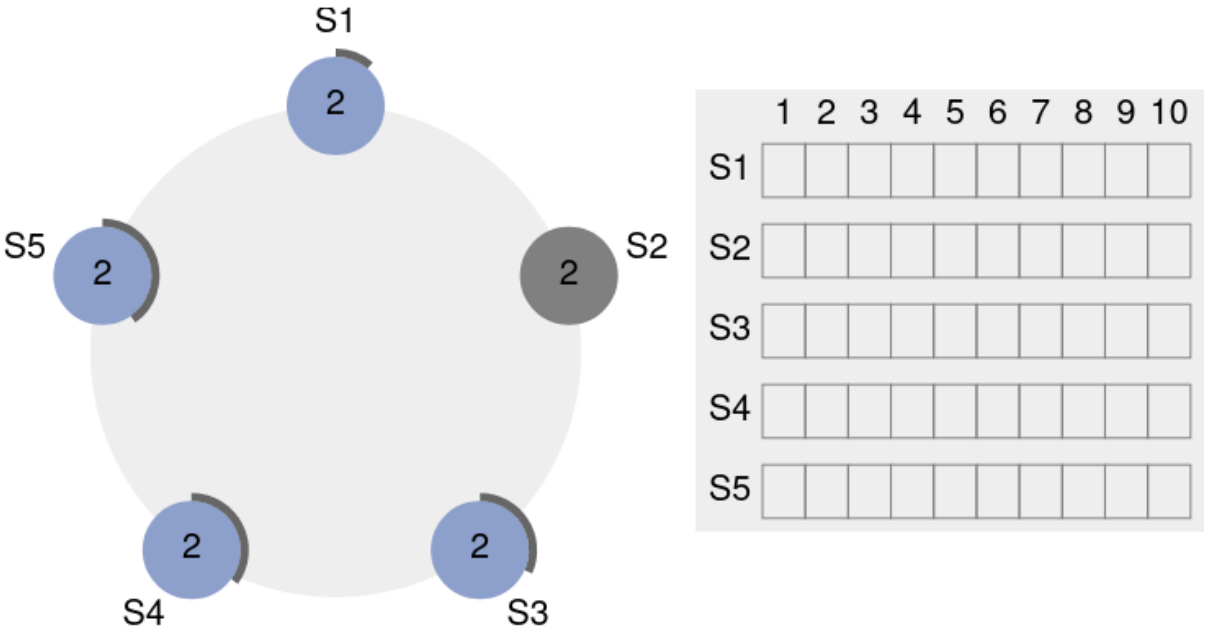


Figure 1/ .6: failover etcd - étape 1

S1 déclenche une élection :

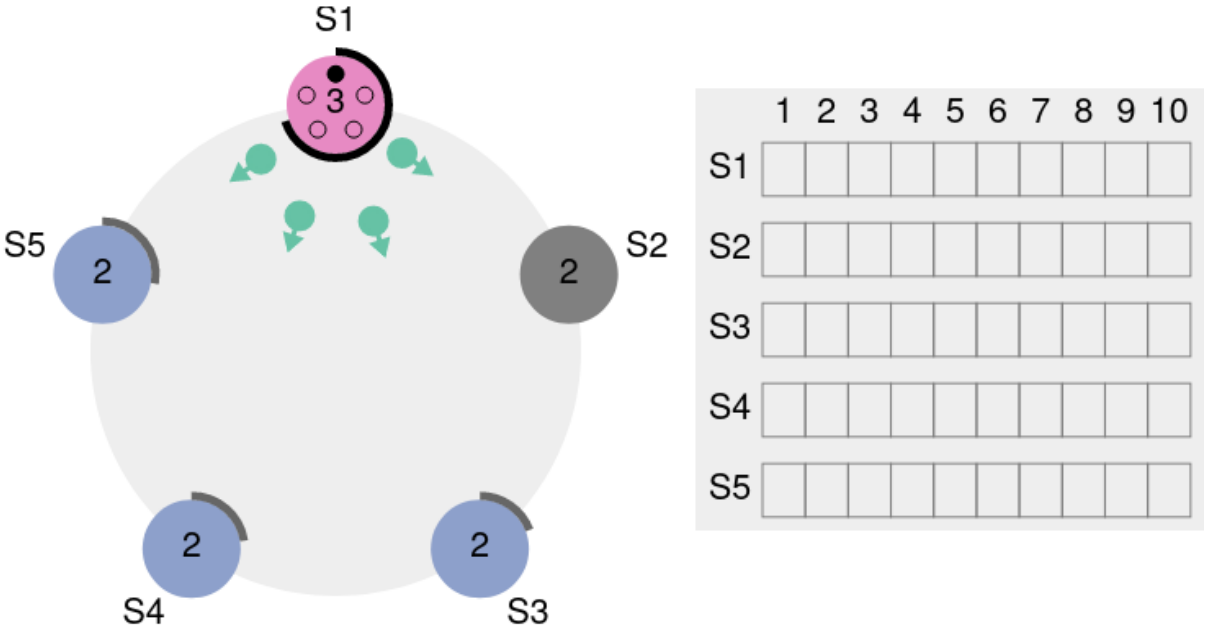


Figure 1/ .7: failover etcd - étape 2

Les autres nœuds accordent leur vote à S1, sauf S2 qui est éteint :

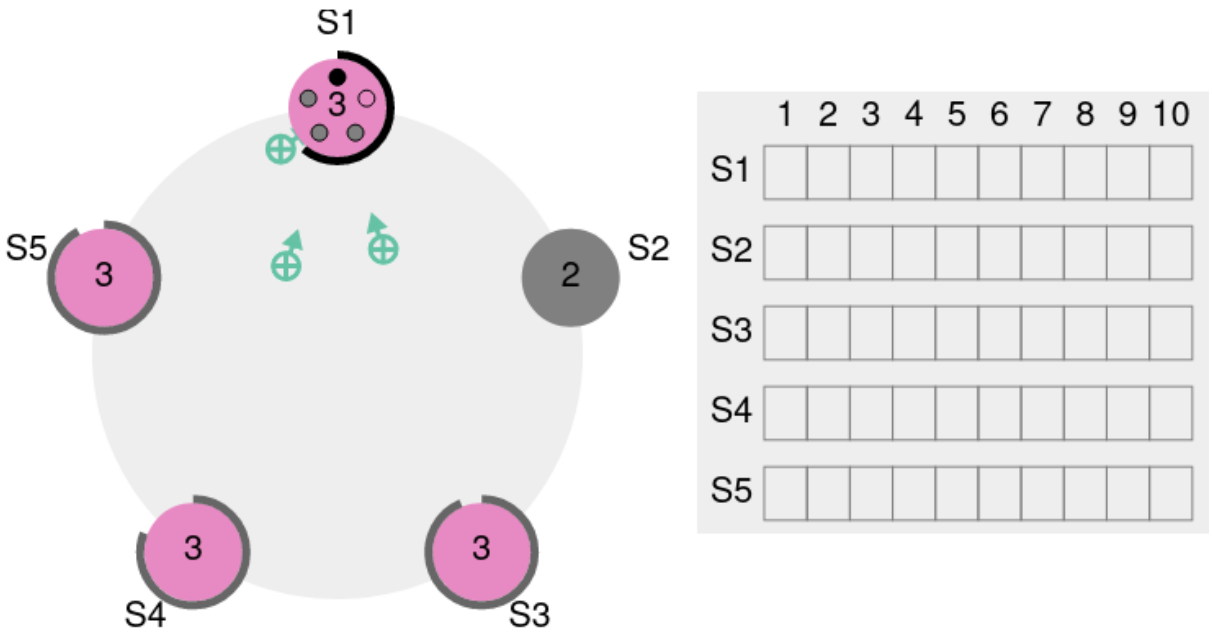


Figure 1/ .8: failover etcd - étape 3

S1 devient *leader* et crée le mandat n°3 :

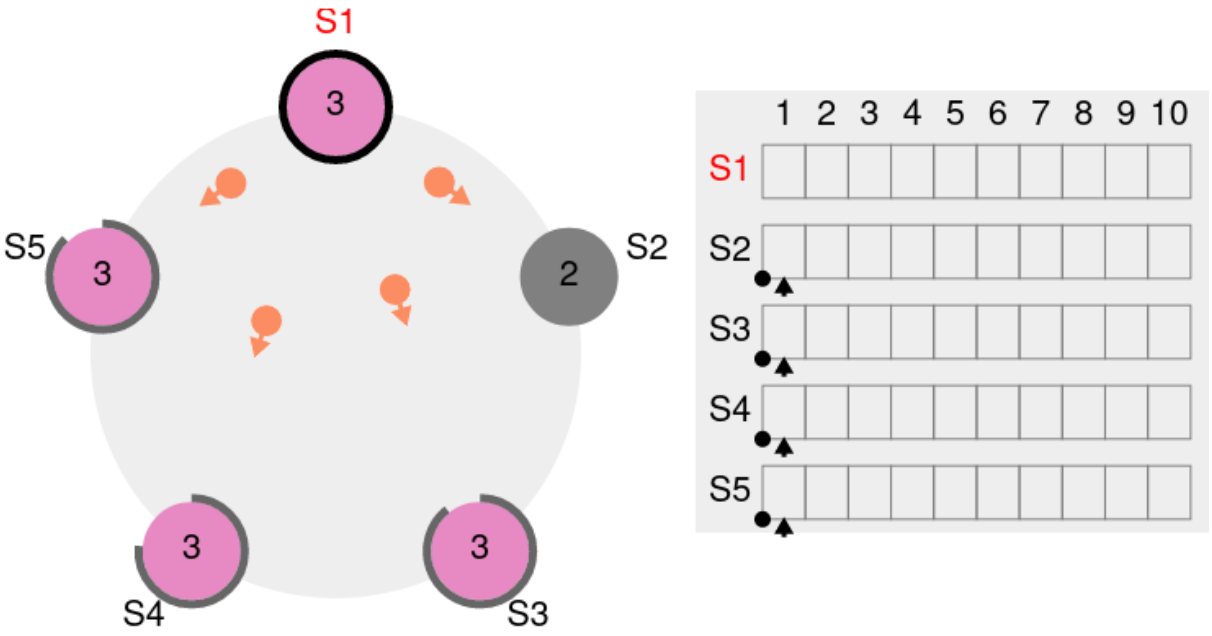


Figure 1/ .9: failover etcd - étape 4

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

Elles apparaissent sur la droite dans le journal au fur et à mesure que le *leader* les diffuse. La diffusion se fait en plusieurs étapes. Écrite et émission depuis S1 :

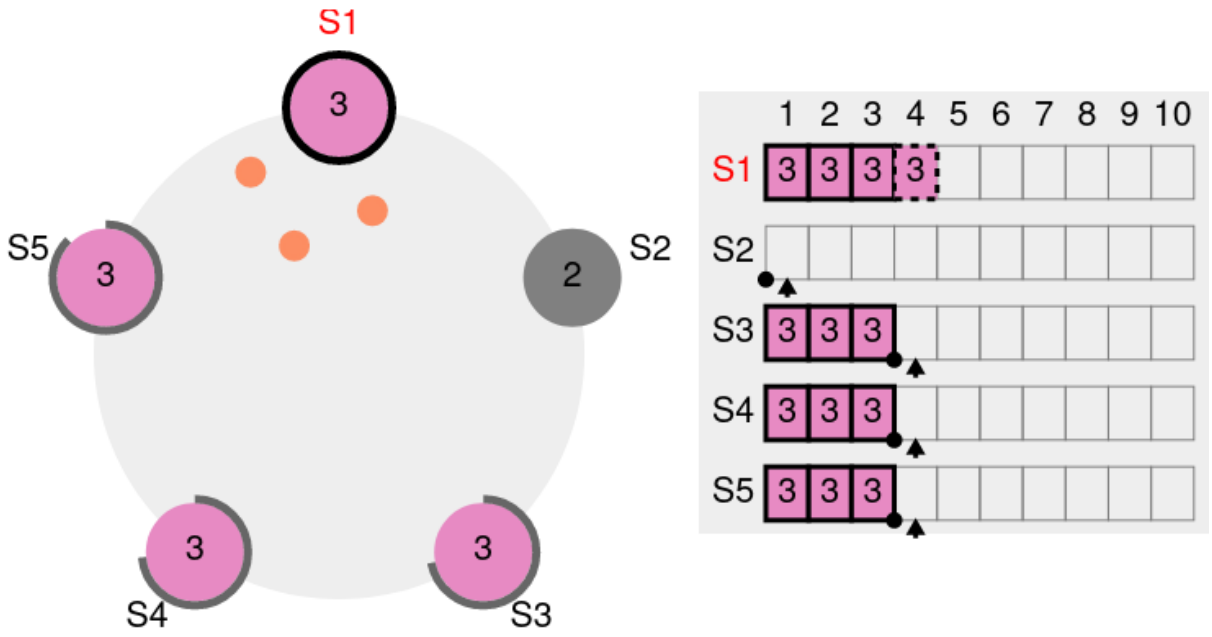


Figure 1/ .10: Requête etcd - étape 1

Bonne réception des nœuds S3, S4 et S5 :

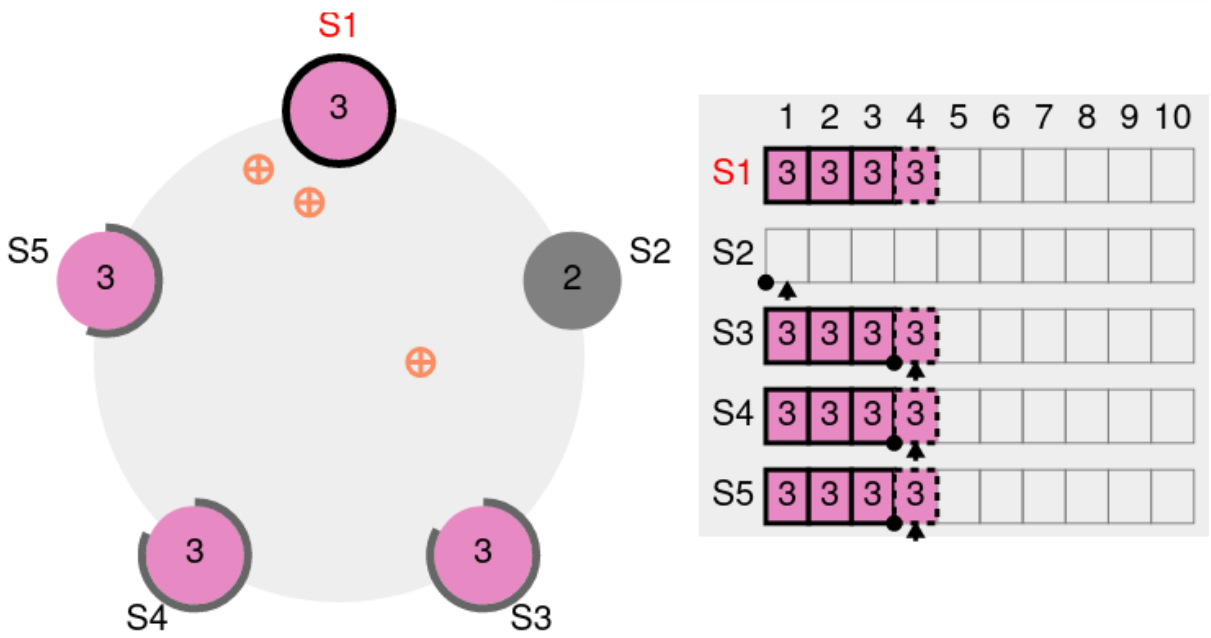


Figure 1/ .11: Requête etcd - étape 2

S1 ayant reçut une majorité de bonne réception valide la valeur auprès des autres nœuds :

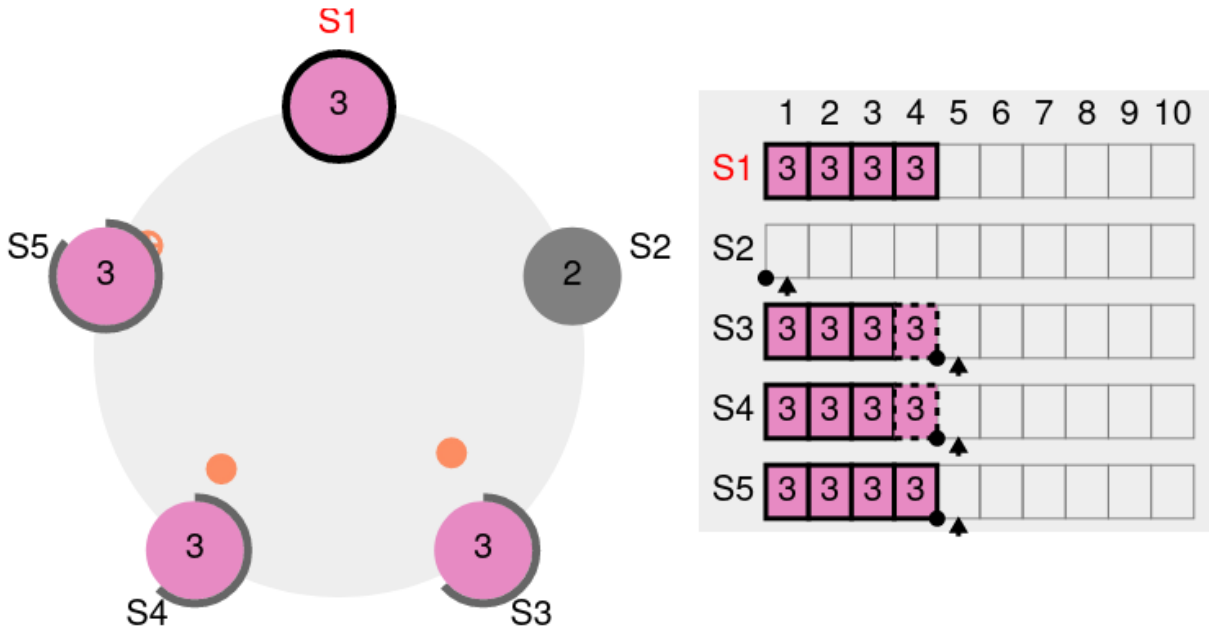


Figure 1/ .12: Requête etcd - étape 3

Tous les nœuds ont validés la valeur :

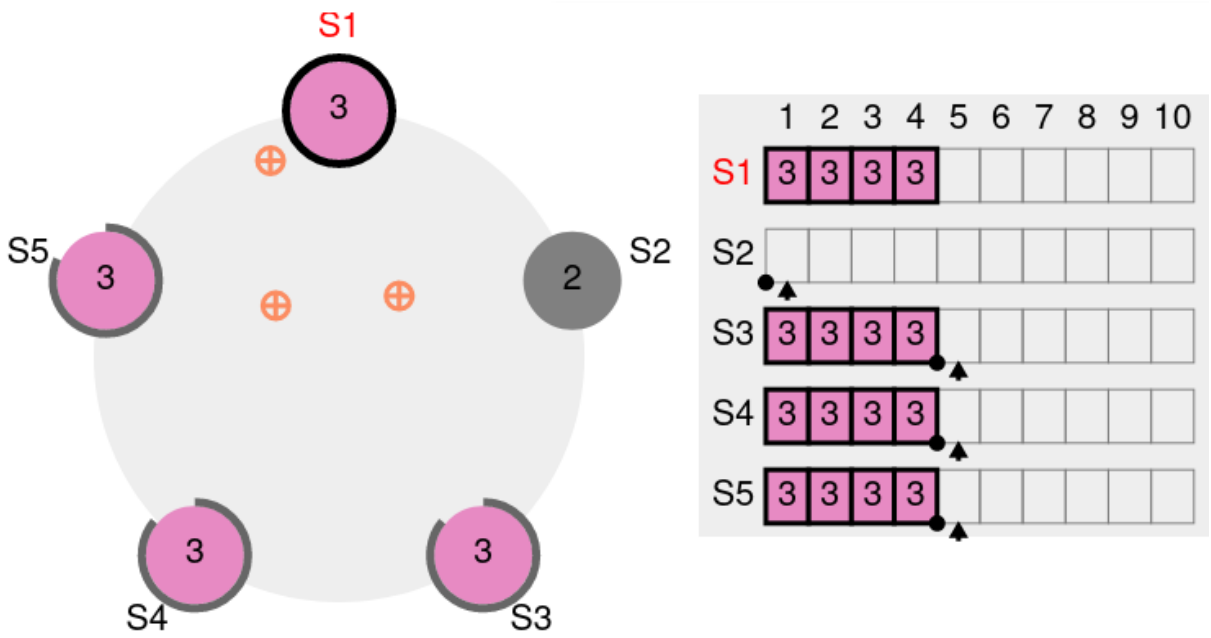


Figure 1/ .13: Requête etcd - étape 4

- Remettre en route l'ancien *leader*.

Celui-ci redémarre à 2 (en tant que *follower*) :

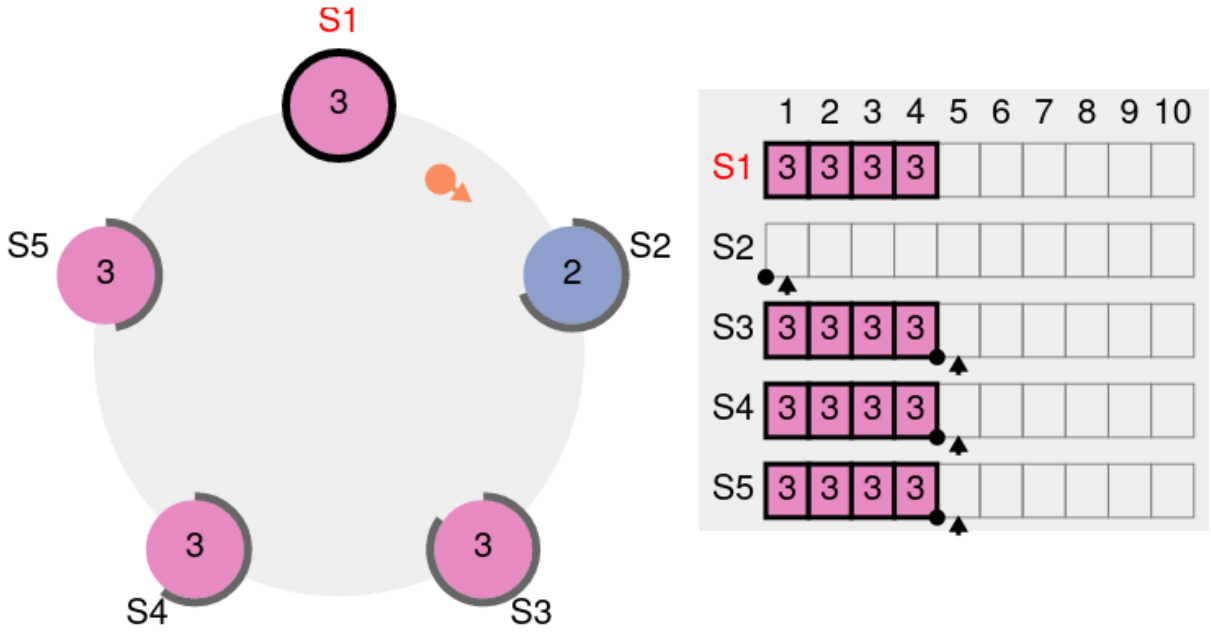


Figure 1/ .14: failback etcd - étape 1

Il bascule sur le mandat 3 au premier *heart beat* reçu, et commence à remplir son journal :

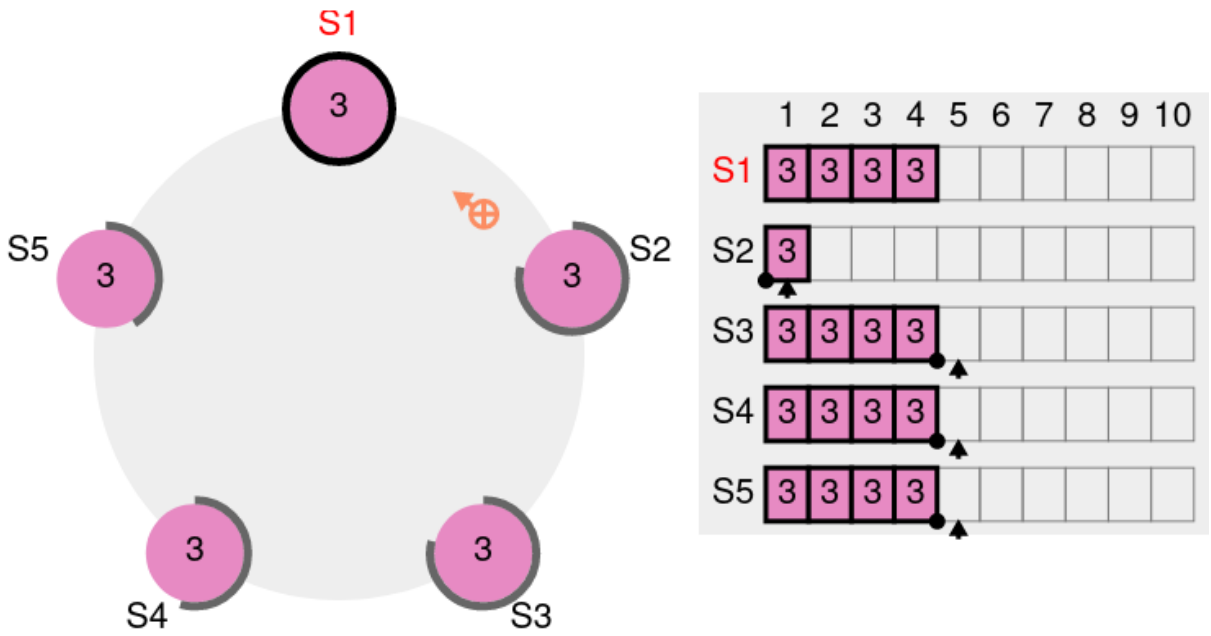
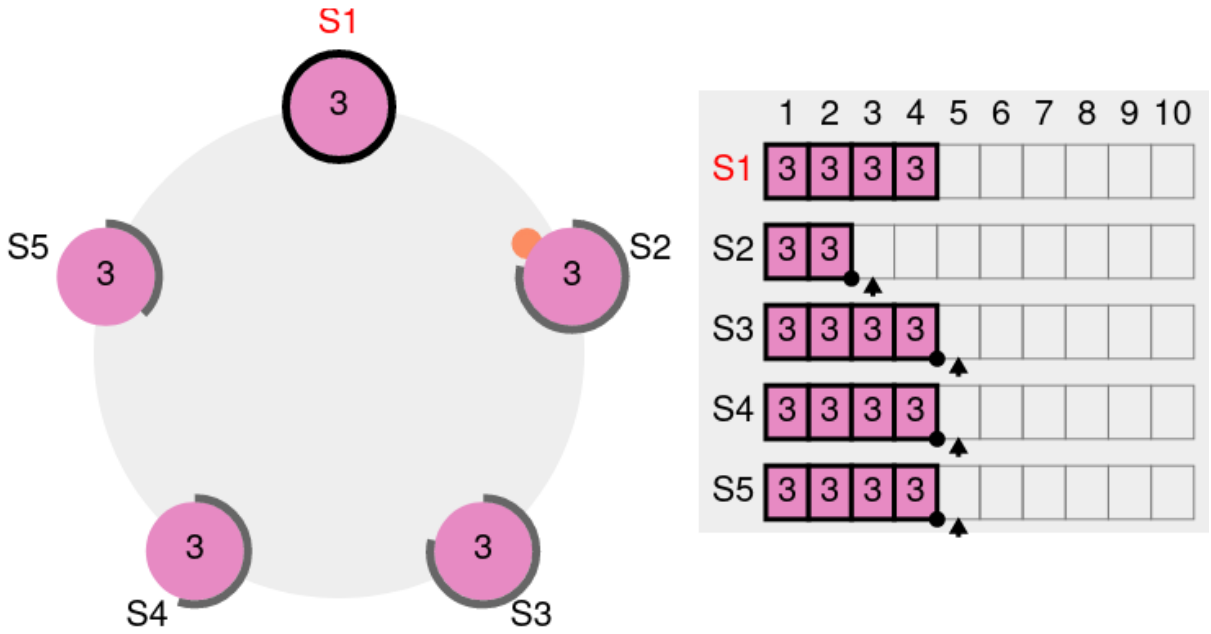
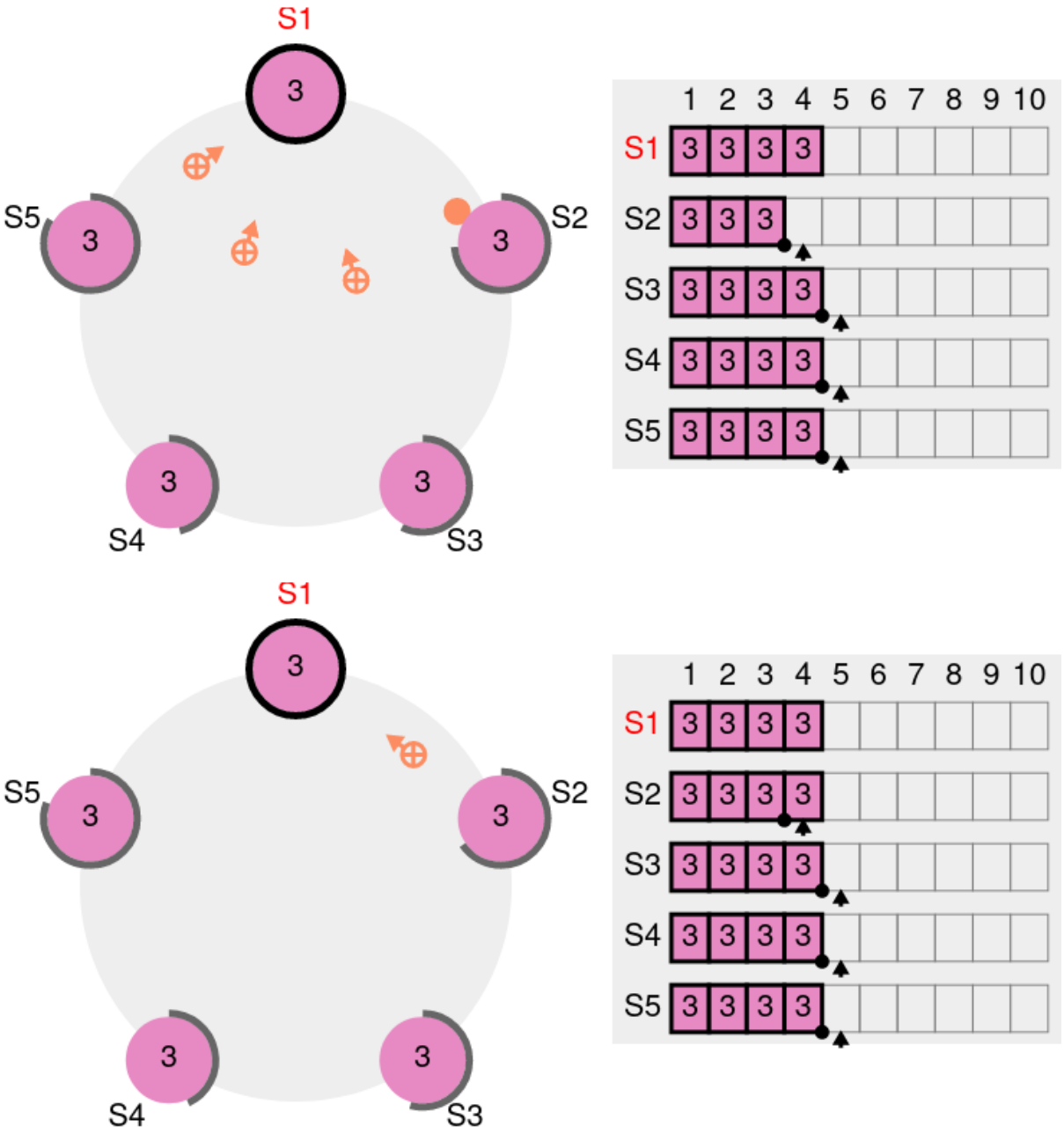


Figure 1/ .15: failback etcd - étape 2

Il rattrape ensuite son journal avec autant d'échange avec le *leader* que nécessaire :





- Arrêter deux nœuds, dont le *leader*, et observer le comportement du cluster.

Le quorum demeure au sein du cluster, il est donc toujours possible de déclencher une élection. Comme précédemment, les trois nœuds restants s'accordent sur un *leader*. Dans notre exemple, tous les nœuds déclenchent leur propre élection en même temps :

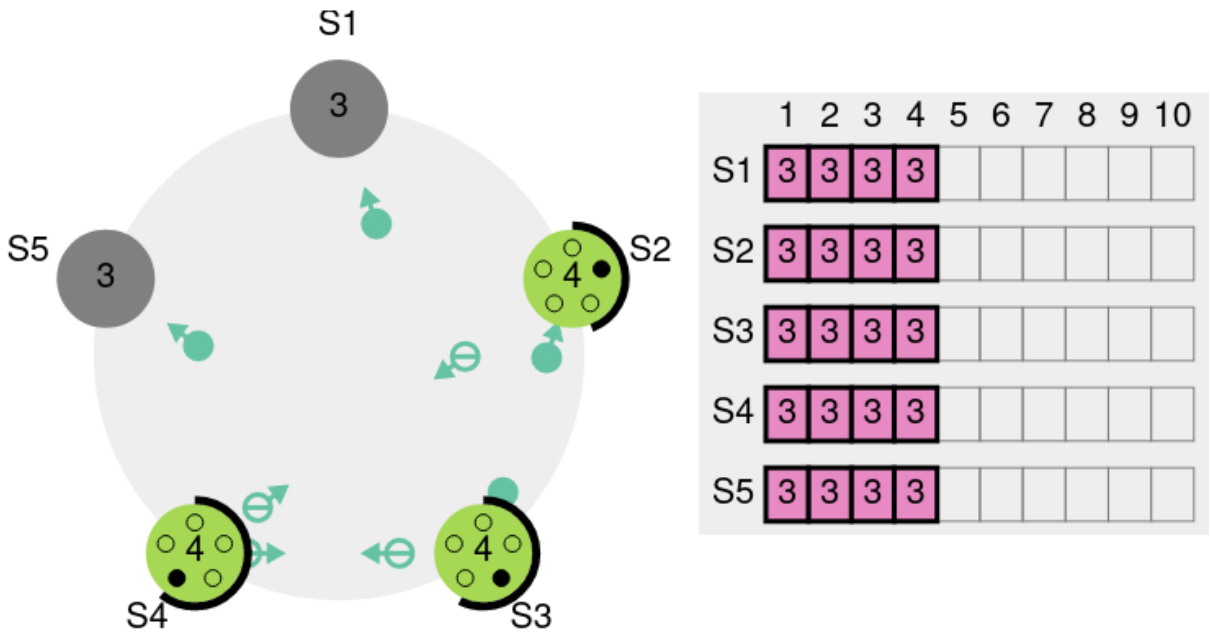


Figure 1/ .16: second failover etcd - étape 1

Une seconde élection est donc nécessaire :

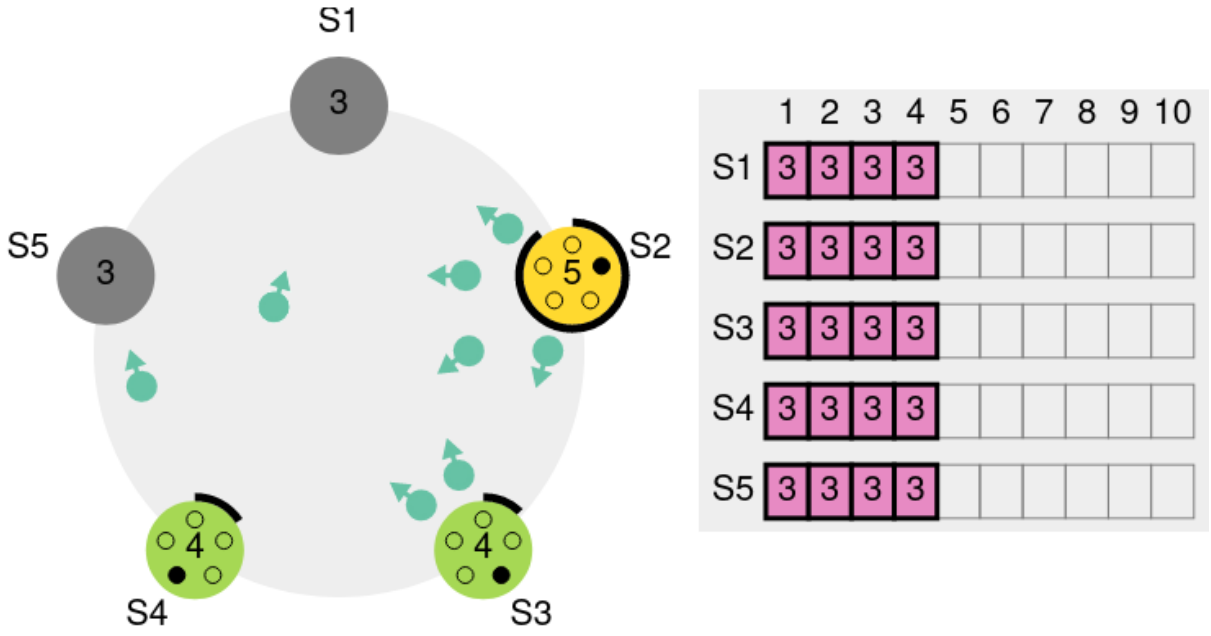


Figure 1/ .17: second failover etcd - étape 2

Elle conduit ici à l'élection du nœud S2 :

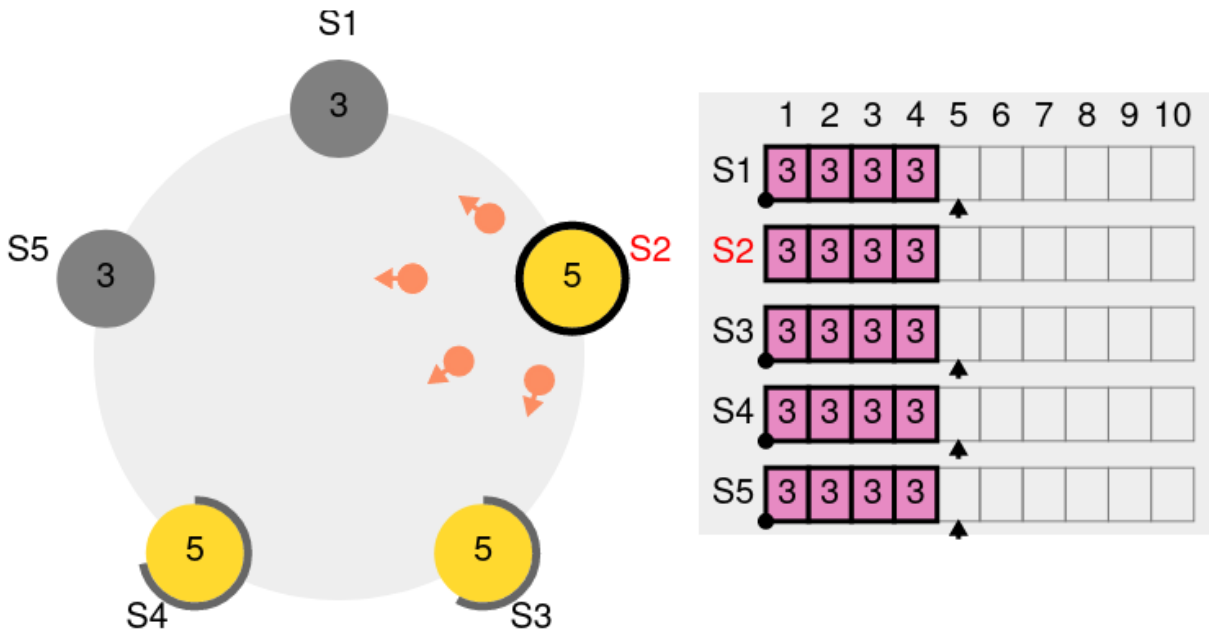


Figure 1/ .18: second failover etcd - étape 3

- Arrêter un nœud secondaire (pour un total de 3 nœuds arrêtés).
- Tester en soumettant des écritures au primaire.
- Qu'en est-il de la tolérance de panne ?

Le cluster continue de fonctionner : on peut lui soumettre des écritures (*requests*) :

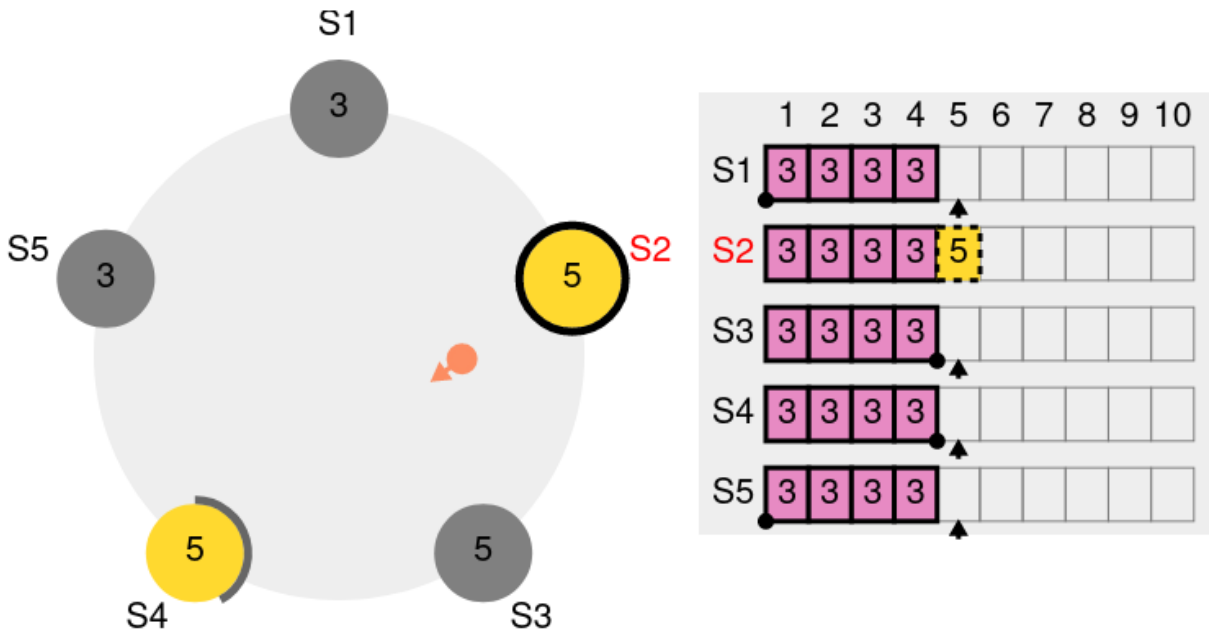


Figure 1/ .19: perte quorum etcd - étape 1

Elles sont bien répliquées mais ne sont pas exécutées (ou commitées) par le *leader* :

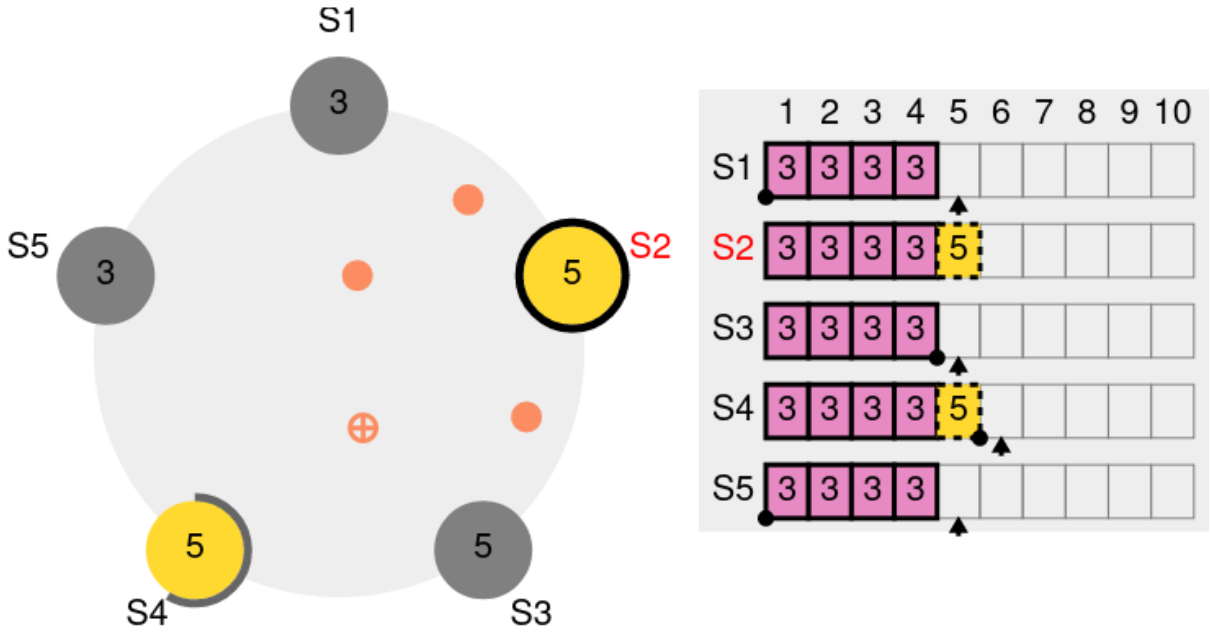


Figure 1/ .20: perte quorum etcd - étape 2

Pour cela, il faut que les écritures aient été répliquées vers la majorité des nœuds du cluster. Cela

signifie que le client reste en attente de confirmation du traitement de sa demande.

La tolérance de panne est maintenant nulle.

- 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 ?

L'un des deux nœuds survivants lance une élection :

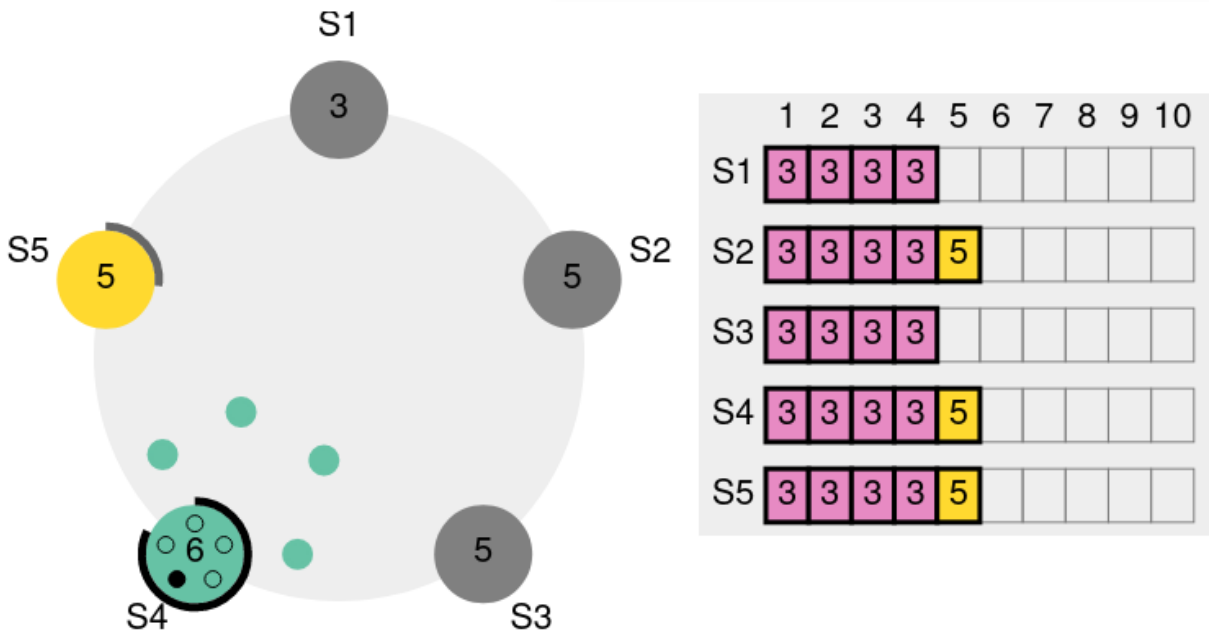


Figure 1/ .21: élection sans quorum etcd - étape 1

L'autre le suit :

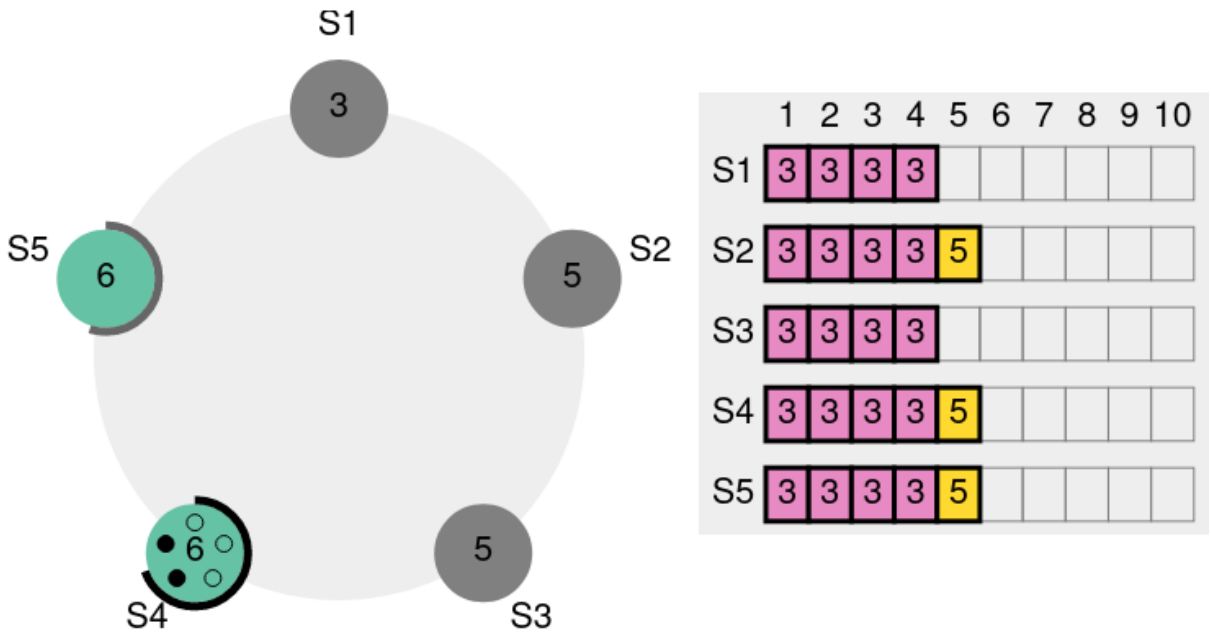


Figure 1/ .22: élection sans quorum etcd - étape 2

Mais comme le quorum de 3 nœuds (moitié de 5 plus 1) n'est pas obtenu, l'élection échoue. Les *time out* continuent d'expirer et de nouvelles élections sont lancées :

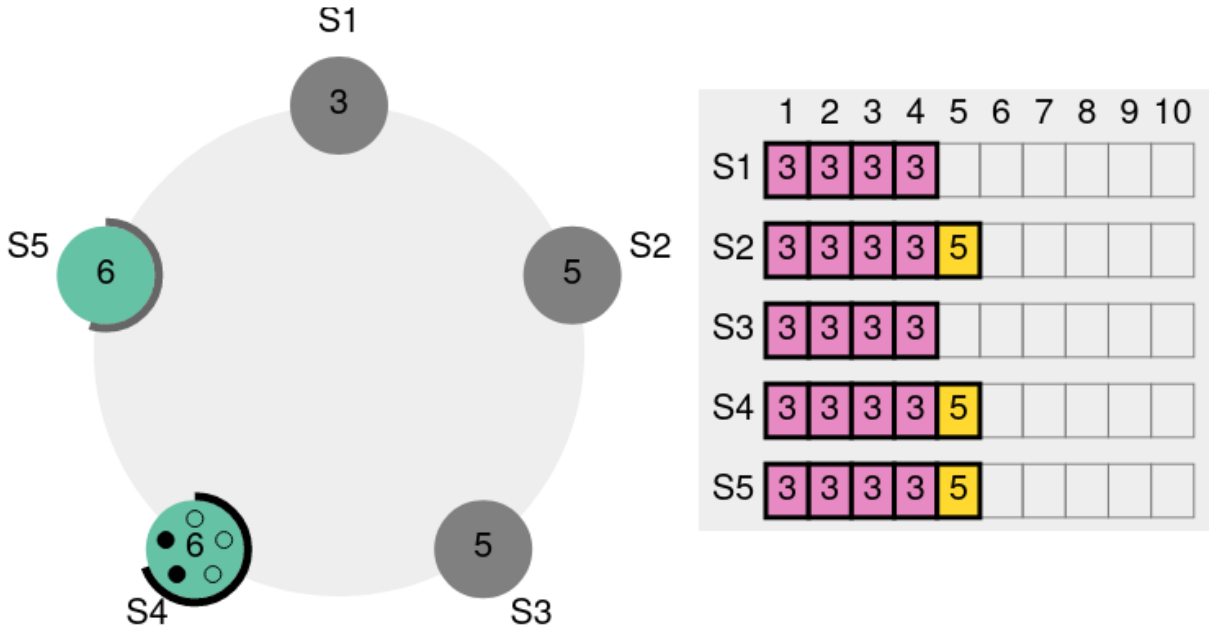


Figure 1/ .23: élection sans quorum etcd - étape 2

Faute de *leader*, il devient impossible de soumettre des écritures. Tout client s'appuyant dessus reçoit une erreur. Il faut attendre le retour en ligne d'un nœud et l'élection d'un nouveau *leader*.

1.10.2 Installation d'etcd sous Debian

> – Installer etcd sur les 3 nœuds `e1`, `e2` et `e3`.

Les paquets etcd sont disponibles dans les dépôts officiels de Debian 12. L'installation consiste simplement à installer les deux paquets `etcd-server` et `etcd-client` :

```
# apt-get install etcd-server etcd-client
```

> – Supprimer le nœud etcd créé sur chaque serveur.

Afin de respecter la politique des paquet Debian, l'installation du paquet `etcd-server` crée automatiquement une instance etcd locale. Nous devons la détruire avant de pouvoir construire notre cluster etcd à trois nœuds.

Sur chaque serveur etcd, exécuter les commandes suivantes :

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

> – Configurer un cluster etcd sur les 3 nœuds `e1`, `e2` et `e3`.

La configuration sous Debian se situe dans le fichier `/etc/default/etcd` où doivent être renseignés les paramètres etcd sous leur forme de variables d'environnements.

Voici le fichier de configuration commenté du nœud `e1`, veillez à adapter **toutes** les adresses IP `10.x.y.z` :

```
# nom du nœud au sein du cluster
ETCD_NAME=e1

# emplacement du répertoire de données
ETCD_DATA_DIR=/var/lib/etcd/acme

# interface et port d'écoute des autres nœuds
ETCD_LISTEN_PEER_URLS=http://10.20.50.249:2380

# interfaces et port d'écoute des clients
ETCD_LISTEN_CLIENT_URLS=http://10.20.50.249:2379,http://127.0.0.1:2379,http://[::1]:2379

# interface et port d'écoute à communiquer aux clients
ETCD_ADVERTISE_CLIENT_URLS=http://10.20.50.249:2379

########################################
## Initialisation du cluster et du nœud
```

```
# création du nœud au sein d'un nouveau cluster
ETCD_INITIAL_CLUSTER_STATE=new

# interface et port à communiquer aux autres nœuds
ETCD_INITIAL_ADVERTISE_PEER_URLS=http://10.20.50.249:2380

# liste des nœuds composant le cluster
ETCD_INITIAL_CLUSTER=e1=http://10.20.50.249:2380,e2=http://10.20.50.147:2380,e3=http://10.20.50.66
```

- Démarrez le cluster etcd

Une fois les trois fichiers de configuration établis, nous pouvons démarrer le cluster en exécutant la commande suivante sur tous les serveurs etcd :

```
# systemctl start etcd
```

La commande suivante doit désormais fonctionner sur n'importe quel nœud :

```
$ etcdctl -wjson endpoint status | jq
```


```
[
  {
    "Endpoint": "127.0.0.1:2379",
    "Status": {
      "header": {
        "cluster_id": 11628162814576028000,
        "member_id": 13786016400334385000,
        "revision": 95,
        "raft_term": 24
      },
      "version": "3.4.23",
      "dbSize": 61440,
      "leader": 2266733444126347800,
      "raftIndex": 112,
      "raftTerm": 24,
      "raftAppliedIndex": 112,
      "dbSizeInUse": 53248
    }
  }
]
```


1.10.3 Installation d'etcd sous Rocky Linux 9

- Installer etcd sur les 3 nœuds `e1`, `e2` et `e3`.

Les paquets etcd sont disponibles depuis les dépôts PGDG pour les distributions Red Hat 9 & dérivées (comme ici Rocky Linux 9). Il nous faut donc au préalable configurer ce dépôt :

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

Il est désormais possible d'installer le paquet `etcd` en activant le dépôt `pgdg-rhel9-extras` :

```
# dnf --enablerepo=pgdg-rhel9-extras install -y etcd
```

> – Configurer un cluster etcd sur les 3 nœuds `e1`, `e2` et `e3`.

La configuration sur cette distribution se situe dans le fichier `/etc/etcd/etcd.conf` où sont renseignés les paramètres etcd sous leur forme de variables d'environnements.

Ci-après les différents paramètres à modifier pour le nœud `e1`, veillez à adapter **toutes** les adresses IP `10.x.y.z` :

```
ETCD_NAME=e1
ETCD_DATA_DIR=/var/lib/etcd/acme
ETCD_LISTEN_PEER_URLS=http://10.20.50.249:2380
ETCD_LISTEN_CLIENT_URLS=http://10.20.50.249:2379,http://127.0.0.1:2379,http://[::1]:2379
ETCD_ADVERTISE_CLIENT_URLS=http://10.20.50.249:2379
ETCD_INITIAL_CLUSTER_STATE=new
ETCD_INITIAL_ADVERTISE_PEER_URLS=http://10.20.50.249:2380
ETCD_INITIAL_CLUSTER=e1=http://10.20.50.249:2380,e2=http://10.20.50.147:2380,e3=http://10.20.50.66
```

> – Activez et démarrez le cluster etcd

Une fois les trois fichiers de configuration établis, nous pouvons démarrer le cluster en exécutant la commande suivante sur tous les serveurs etcd :

```
# systemctl start etcd
```

Afin qu'etcd démarre automatiquement avec le serveur, il est nécessaire d'activer le service :

```
# systemctl enable etcd
```

Une fois les trois fichiers de configuration établis, la commande suivante doit fonctionner sur n'importe quel nœud :

```
$ etcdctl -wjson endpoint status | jq
```


```
[
  {
    "Endpoint": "127.0.0.1:2379",
    "Status": {
      "header": {
        "cluster_id": 11628162814576028000,
        "member_id": 13786016400334385000,
        "revision": 95,
        "raft_term": 24
      },
      "version": "3.4.23",
      "dbSize": 61440,
      "leader": 2266733444126347800,
      "raftIndex": 112,
      "raftTerm": 24,
      "raftAppliedIndex": 112,
      "dbSizeInUse": 53248
    }
  }
]
```

1.10.4 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`.

```
$ etcdctl put foo bar
OK
```

- Récupérer cette valeur depuis un autre nœud.

```
$ etcdctl get foo
foo
bar
```

- Modifier la valeur à `baz`.

```
$ etcdctl put foo baz
OK
```

- Créer un répertoire `food` contenant les clés/valeurs `poisson: bar` et `vin: blanc`.

```
$ etcdctl put food/poisson bar
OK

$ etcdctl put food/vin blanc
OK

$ etcdctl get --keys-only --prefix food/
food/poisson

food/vin
```

- Récupérer toutes les clefs du répertoire `food` en exigeant une réponse d'un quorum.

```
$ etcdctl get --consistency="l" --keys-only --prefix food/
 /food/poisson

 /food/vin
```

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>`.

Dans notre correctif, `e1` est actuellement *leader*. Dans une fenêtre sur `e2` et `e3`, laisser défiler le journal :

```
# journalctl -fu etcd
```

Depuis une session dans le serveur hôte, interroger `e2` ou `e3` continuellement à propos de la santé de l'agrégat avec par exemple :

```
$ watch -n1 "curl -s -XGET http://10.20.50.3:2379/health | jq"
```

```
{
  "health": "true",
}
```

Depuis le serveur hôte, cette commande interrompt brutalement la machine virtuelle `e1` (mais sans la détruire) :

```
# virsh destroy e1
Domain 'e1' destroyed
```

Il est aussi possible de suspendre la machine avec la commande suivante :

```
# virsh suspend e1
Domain 'e1' suspended
```

La commande inverse est alors `virsh resume e1`.
Notez que la machine est *suspendue*, donc encore présente, mais plus du tout exécutée par l'hyperviseur. Cette présence inactive peut parfois des créer des situations déstabilisantes pour les autres machines virtuelles ayant toujours des connexions TCP en suspend. Aussi, après son réveil, sauf présence de `chrony` ou `ntpsec`, l'horloge de cette machine virtuelle nécessite une intervention manuelle afin de la resynchroniser.

Dans les traces de **e3**, nous trouvons :

```
INFO: 49fc71338f77c1c4 is starting a new election at term 3
INFO: 49fc71338f77c1c4 became candidate at term 4
INFO: 49fc71338f77c1c4 received MsgVoteResp from 49fc71338f77c1c4 at term 4
INFO: 49fc71338f77c1c4 [logterm: 3, index: 9] sent MsgVote request to
 ↪  97e570ef03022438 at term 4
INFO: 49fc71338f77c1c4 [logterm: 3, index: 9] sent MsgVote request to
 ↪  be3b2f45686f7694 at term 4
INFO: raft.node: 49fc71338f77c1c4 lost leader be3b2f45686f7694 at term 4
```

L'identifiant `49fc71338f77c1c4` est celui de `e3` lui même. Ces traces nous indiquent :

- le déclenchement de l'élection avec un nouveau mandat n°4 ;
- le vote de `e3` pour lui même ;
- les demandes de vote vers `e1` et `e2`.
- la perte du leader `e1` (identifiant `be3b2f45686f7694`) durant le mandat n°4

Suivent alors les messages suivants:

```
INFO: 49fc71338f77c1c4 received MsgVoteResp from 97e570ef03022438 at term 4
INFO: 49fc71338f77c1c4 has received 2 MsgVoteResp votes and 0 vote rejections
INFO: 49fc71338f77c1c4 became leader at term 4
INFO: raft.node: 49fc71338f77c1c4 elected leader 49fc71338f77c1c4 at term 4
```

Seul `e2` (ici `97e570ef03022438`) répond au vote, mais cette réponse suffit à atteindre le quorum (de 2 sur 3) et valider l'élection de `e3` comme nouveau *leader*.

En interrogeant l'état de **e3**, nous confirmons bien qu'il est *leader* :

```
$ etcdctl -wtable endpoint status
+----------------+------------------+---------+---------+-----------+-[…]
|    ENDPOINT    |        ID        | VERSION | DB SIZE | IS LEADER | […]
+----------------+------------------+---------+---------+-----------+-[…]
| 127.0.0.1:2379 | 49fc71338f77c1c4 |  3.4.23 |   20 kB |      true | […]
+----------------+------------------+---------+---------+-----------+-[…]
```

> – Geler le nouveau *leader* de la même manière et voir les traces du nœud restant.

L'état de l'agrégat tombe en erreur suite à cette perte du quorum :

```
$ curl -s -XGET http://10.20.50.47:2379/health | jq
```

```
{
  "health": "false"
}
```

Dans les traces de `e2`, nous constatons qu'il tente en boucle une élection, envoie des messages mais faute de réponse, n'obtient jamais d'accord pour l'élection :

```
15:36:31 INFO: 97e570ef03022438 is starting a new election at term 4
15:36:31 INFO: 97e570ef03022438 became candidate at term 5
15:36:31 INFO: 97e570ef03022438 received MsgVoteResp from 97e570ef03022438 at term 5
15:36:31 INFO: 97e570ef03022438 [logterm: 4, index: 1192] sent MsgVote request to
 ↪  49fc71338f77c1c4 at term 5
15:36:31 INFO: 97e570ef03022438 [logterm: 4, index: 1192] sent MsgVote request to
 ↪  be3b2f45686f7694 at term 5
15:36:31 INFO: raft.node: 97e570ef03022438 lost leader 49fc71338f77c1c4 at term 5
15:36:33 INFO: 97e570ef03022438 is starting a new election at term 5
15:36:33 INFO: 97e570ef03022438 became candidate at term 6
15:36:33 INFO: 97e570ef03022438 received MsgVoteResp from 97e570ef03022438 at term 6
15:36:33 INFO: 97e570ef03022438 [logterm: 4, index: 1192] sent MsgVote request to
 ↪  49fc71338f77c1c4 at term 6
15:36:33 INFO: 97e570ef03022438 [logterm: 4, index: 1192] sent MsgVote request to
 ↪  be3b2f45686f7694 at term 6
15:36:34 INFO: 97e570ef03022438 is starting a new election at term 6
15:36:34 INFO: 97e570ef03022438 became candidate at term 7
15:36:34 INFO: 97e570ef03022438 received MsgVoteResp from 97e570ef03022438 at term 7
15:36:34 INFO: 97e570ef03022438 [logterm: 4, index: 1192] sent MsgVote request to
 ↪  49fc71338f77c1c4 at term 7
15:36:34 INFO: 97e570ef03022438 [logterm: 4, index: 1192] sent MsgVote request to
 ↪  be3b2f45686f7694 at term 7
```

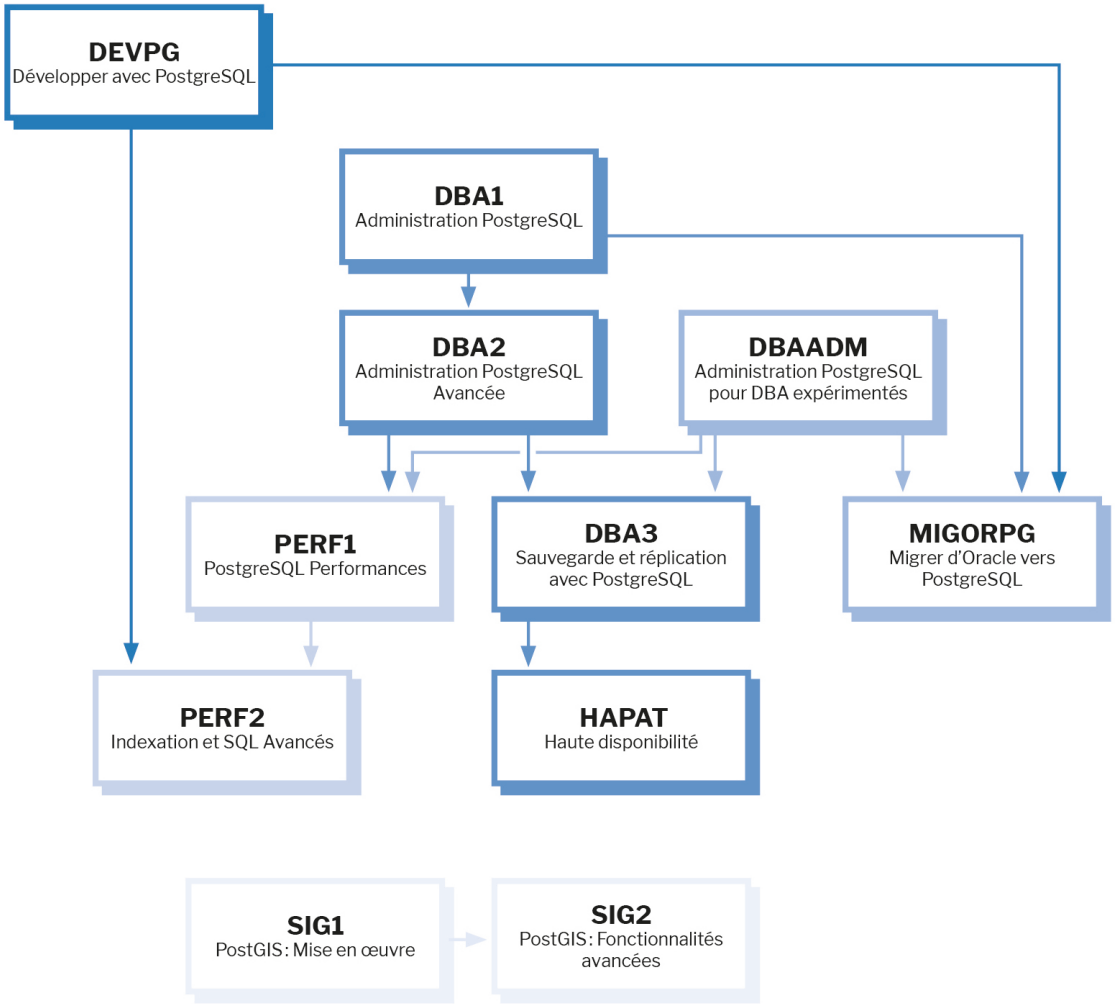
La situation revient à la normale dès qu'un des deux autres revient en ligne avec par exemple la commande `virsh start e3`.

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.