



XUNTA DE GALICIA  
CONSELLERÍA DE EDUCACIÓN  
E ORDENACIÓN UNIVERSITARIA

I.E.S. ARMANDO  
COTARELO VALLEDOR 

**DAVID CASTRO VILAS**

Proxecto final do Ciclo Formativo de Técnico Superior en Administración de Sistemas  
Informáticos en Rede. Curso 2020/2021. IES ARMANDO COTARELO VALLEDOR

**Curso 2020–2021**

# **Alta dispoñibilidade con PostgreSQL**



## Índice

---

1 Obxectivos do proxecto.....	4
1.1 Terminoloxía e principios básicos.....	4
1.2 O produto final.....	4
1.3 Motivación persoal.....	5
2 Análise.....	5
2.1 Necesidades.....	5
2.2 Dificultades.....	6
2.3 Organización de tarefas.....	7
3 Deseño.....	8
3.1 WAL Logging: a replicación de PostgreSQL.....	8
3.2 WAL Archiving: backups e PITR.....	9
3.3 Xestión do cluster e failover automático.....	10
3.4 Seleccionando un DCS.....	11
3.5 Balanceo de carga con HAProxy.....	13
3.6 Engadindo un pool de conexións.....	15
3.7 PGBackRest: o repositorio de backups.....	17
3.8 Deseño da infraestrutura.....	18
4 Implementación.....	20
4.1 Sistemas.....	20
4.2 Táboa de direccionamento.....	20
4.3 Creación do cluster de Etcd.....	20
4.4 Configuración do cluster de Patroni.....	22
4.5 Configuración de PGBouncer.....	26
4.6 Configuración de HAProxy Keepalived.....	28
4.7 Montando o repositorio de backups.....	33
4.8 Estimación e planificación da posta en marcha.....	36
5 Soporte e mantemento.....	37
5.1 Distribución dos sistemas.....	37
5.2 Copias de seguridade.....	37
5.3 Plan de continxencias.....	38
5.4 Supervisión e monitorización.....	38
5.5 Actualizacións.....	39
6 Estudo económico.....	40
6.1 DaaS: Database as a Service.....	40

6.2 Amazon Aurora.....	41
6.3 Heroku.....	41
6.4 Cloud SQL.....	41
6.5 Azure SQL.....	42
6.6 Custos da infraestrutura e os servizos contratados.....	42
6.6.1 Hardware.....	42
6.6.2 Persoal.....	44
7 Posibles melloras.....	44
8 Dificultades atopadas.....	45
9 Conclusións.....	47
10 Bibliografía e referencias.....	47
11 Anexos.....	49
11.1 Bootstrap dunha réplica a partires dun backup.....	49

## 1 Obxectivos do proxecto

---

Neste proxecto vou a investigar diferentes ferramentas de código aberto que permitan implementar un servizo de base de datos PostgreSQL con alta dispoñibilidade e elaborar unha infraestrutura personalizada con elas.

### 1.1 Terminoloxía e principios básicos

---

#### Os principios do deseño:

A **clusterización** dun servizo de base de datos é o proceso de combinar varios servidores ou instancias que conectan unha única base de datos. Un único servidor pode non ser suficiente para xestionar un gran volume de peticións ou unha base de datos de gran tamaño.

#### Clusterizando unha base de datos obtemos:

- **Redundancia de datos:** todos os nodos están sincronizados, contendo exactamente os mesmos datos, de maneira que se un falla temos outro perfectamente operativo e sincronizado.
- **Balanceo de carga:** repartir as solicitudes dos clientes entre os distintos nodos permite dar servizo a un maior volume de usuarios e reduce o risco de fallo por sobrecarga ou incapacidade de atender a un gran número de peticións.
- **Alta dispoñibilidade:** que o servizo se execute de maneira ininterrompida e sen fallos o maior tempo posible. A alta dispoñibilidade ten 3 principios fundamentais:
  - A eliminación de SPOFs (Single Points Of Failure). Isto é que ante o fallo dunha parte do sistema este non colapse na súa totalidade.
  - Crossover confiable: un mecanismo que permita realizar un *switchover* automático no caso de fallo. É dicir, que se un nodo falla o sistema sexa capaz de promocionar outro ao seu nivel ou levantar un novo.
  - Detección de fallos: se os puntos 1 e 2 se revisan de maneira proactiva, un usuario nunca debería atoparse ante un fallo total do sistema.

### 1.2 O produto final

---

O propósito deste traballo de investigación é deseñar unha infraestrutura de bases de datos que ofrezca un servizo fiable e ininterrompido, que poida atender un gran volume de peticións e que teña unha resposta ante fallos automatizada o máximo posible.

Unha infraestrutura deste tipo é un produto atractivo para calquera empresa que precise dunha base de datos para calquera dos seus servizos. O tempo que un

servizo está caído pode resultar moi custoso para unha organización polo que a alta dispoñibilidade é algo ao que se tende case de forma natural.

O downtime (*tempo que un servizo está caído*) é caro, pero evitar o downtime é aínda máis caro, polo que ofrecer unha solución que non necesite monitorización ou supervisión constante reduce os custos enormemente.

PostgreSQL é a base de datos favorita dos desenvolvedores de software hoxe en día e son moitas compañías de software que buscan unha solución que non só dea soporte continuo ás súas aplicacións, se non que tamén sexa escalable e o máis autosuficiente posible.

### 1.3 Motivación persoal

---

Fai xa uns meses que me picou a curiosidade e me entraron as ganas de aprender sobre PostgreSQL. Isto xuntouse con que na miña empresa da FCT teñen aplicacións Java con estado, polo que pensei que podía ser interesante investigar distintas maneiras de mellorar a súa infraestrutura actual, avalialas e potencialmente implantalas nun contorno de produción nun futuro a medida que a empresa medra.

Actualmente no caso de que falle o servidor principal, levantar o secundario e promocionalo é un proceso manual. Aínda que non resulte algo complicado, aumenta o downtime potencial do servizo, ademais de que non se está aproveitando ao máximo o hardware xa que só un servidor atende peticións á vez, non se reparte a carga.

## 2 Análise

---

### 2.1 Necesidades

---

#### Entón, por onde comezamos?

Para comezar precisamos varias instancias de PostgreSQL: unha primaria e dúas secundarias (standby). Sería o ideal para que no caso de fallo non quede só unha en execución.

Estas instancias ou nodos deberán sincronizarse periódicamente, en intervalos o máis curtos posible.

Os datos entre os servidores deben ser idénticos para que non haxa problemas ao cambiar entre un ou outro.

Cando un servidor falle, o sistema debería ser capaz de detectar o estado dos outros nodos e promocionar a primario a calquera dos dous, así como de detectar cando o nodo caído volva a estar en funcionamento e reintroducilo no cluster co rol de standby.

## Isto está moi ben...pero que hai da repartición da carga de traballo?

Empregando un balanceador de carga podemos repartir as conexións ou as consultas á base de datos que sexan de só lectura entre os tres nodos, que serán **hot standby**, polo que non estarán esperando un fallo do primario para entrar en funcionamento, se non que estarán sendo empregados en conxunto, non desperdiciando o hardware en ningún momento en comparación co standby tradicional.

As consultas de lectura son as máis frecuentes, polo que é conveniente repartilas, e as de lectura e escritura poden ir ao nodo principal sen problema.

O balanceador de carga deberá ser **escalable** e permitir o **auto failover** igual que o serán os tres nodos de PostgreSQL. Para isto precisamos un balanceador de carga de respaldo polo menos, e algún sistema que monitorice o servizo e no caso de fallo promocioe ao balanceador de carga de respaldo a principal, e que poida reintroducir ao caído cando sexa restaurado.

### Connection Pooling

Se pretendemos que a nosa solución sexa o máis escalable posible e que aproveite ao máximo os servizos de base de datos precisamos un pool de conexións para cada nodo de PostgreSQL.

Un pool de conexións mellora a eficiencia da base de datos optimizando a xestión da carga de traballo. Cun pool de conexións temos un mediador entre un cliente ou aplicación e a base de datos.

O pool contén un número de conexións abertas constantemente coa base de datos e recibe e redirixe as peticións dos clientes permitíndolles empregar temporalmente unha desas conexións.

Como estas conexións xa están establecidas, redúcese a carga que supón crear un novo proceso cada vez que un cliente se conecta con PostgreSQL e permite empregar os recursos que se gastarían en abrir estes procesos en servir máis peticións ou acelerar as existentes.

### Axilidade

Proporcionar un método sinxelo de conectar aplicacións ou clientes á infraestrutura sen quebradeiros de cabeza, e, no caso de que algún sistema falle, reducir o impacto no lado do cliente ou facelo imperceptible.

## 2.2 Dificultades

---

A principal dificultade reside en que moitas das ferramentas necesarias para implementar isto que propoño son novas para min, así como o xestor de base de datos escollido, polo que vai ser un traballo de investigación extenso.



## 2.3 Organización de tarefas

Ao longo destas semanas realizarase unha **investigación exhaustiva** das tecnoloxías e plantexarase unha infraestrutura que acomode as distintas ferramentas escollidas e optimice o traballo.



Unha vez teñamos escollido un stack tecnolóxico, comezarase coa preparación dos equipos e a instalación dos distintos servizos:

- PostgreSQL con replicación de BBDD mediante un xestor de clusters que permita o autofailover.
- O xestor de clusters precisará un DCS, é dicir, un mecanismo que lle facilite a toma de decisións á hora de promocionar un nodo e establecer as réplicas.
- Balanceador de carga para repartir as peticións á base de datos entre as distintas instancias.
- O DCS precisa outras instancias de respaldo, que se configurarán unha vez a versión básica do deseño funcione acorde ao esperado.
- Montarase un respaldo para o balanceador de carga e implementarase un sistema que permita cambiar entre os dous no caso de que falle o principal.

- Para o máximo aproveitamento do servizo de BBDD, instalárase e configurárase un pool de conexións.
- Faranse probas ante situacións críticas para confirmar a eficacia do sistema.
- Implementárase un mecanismo de respaldo para as BBDD e documentáranse métodos de iniciar o cluster a partires dos backups realizados.

### 3 Deseño

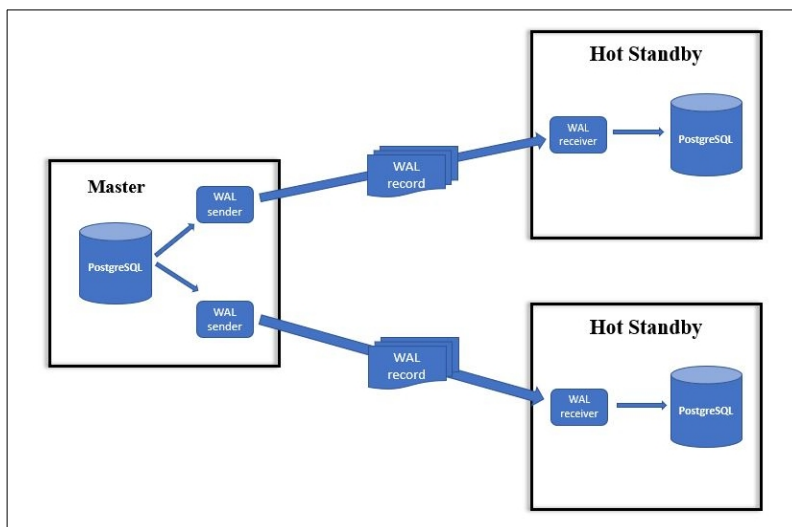
#### 3.1 WAL Logging: a replicación de PostgreSQL

Para manter todas as instancias do servizo de BBDD actualizadas en todo momento empregaremos a **replicación WAL**.

WAL son as siglas de Write-Ahead Logging, o protocolo empregado por PostgreSQL para asegurarse de que todos os cambios que se realizan nunha base de datos se rexistran correctamente na orde na que acontecen.

O WAL axuda no mantemento da integridade de datos facilitando a recuperación dos mesmos no caso de que haxa un fallo ou unha caída do servizo.

Aquí é onde entra en acción a replicación por retransmisión (*streaming replication*):



O servidor que teña o rol de primario retransmitirá os rexistros WAL por segmentos aos outros servidores mediante conexión directa, de maneira que os datos sempre estean sincronizados. A ventaxa da retransmisión destes rexistros é que non se envían cando se acada o límite de capacidade, se non que se fai de forma inmediata, polo que o estado da base de datos será idéntico en todo momento.



### 3.2 WAL Archiving: backups e PITR

Postgres almacena os arquivos WAL no directorio pg\_wal do datadir, pero estes arquivos non están protexidos e poden ser borrados ou sobrescritos polo servidor, polo que dispoñemos da posibilidade de facer unha copia destes arquivos nun directorio distinto.

Para isto temos que habilitar **archive\_mode**, especificar un modo de facer o arquivado mediante **archive\_command** e o nivel de detalle do almacenamento WAL que queremos:

- Minimal: só a información requirida para a recuperación ante unha caída de servizo, non serve para a replicación.
- Replica: a información xusta para facer a replicación.
- Logical: información requirida para a replicación lóxica.



#### Point In Time Recovery (PITR)

A recuperación dun punto no tempo ou PITR permítenos executar únicamente aqueles arquivos WAL necesarios para retornar a base de datos a un momento concreto. Isto é moi útil se a BBDD entra nun estado de erro e se precisa restablecer dende un punto no que se saiba con certeza que non se corrompeu a información.


Para que se poida realizar a PITR precisamos:

- Un backup completo da base de datos
- Arquivamento WAL

### 3.3 Xestión do cluster e failover automático

Entre as alternativas máis destacables hoxe en día atópanse **PAF** (Failover Automático de PostgreSQL), **REPMGR** e **Patroni**.

Elaborouse unha táboa comparativa entre estes tres frameworks de alta dispoñibilidade:

PROS E CONTRAS	PAF	REPMGR	PATRONI
AUTOMATIZA A INICIALIZACIÓN E A CONFIGURACIÓN DE POSTGRESQL	✗	✓	✓
XESTIONA OS FALLOS DOS NODOS E ELIXE SUBSTITUTOS	✓	✓	✓
REACCIONA ANTE UN ESCENARIO SPLIT BRAIN*	✓	✗	✓
INCLÚE COMPLETA MONITORIZACIÓN E XESTIÓN ANTE FALLOS DE REDE	✓	✗	✓
PERMITE XESTIONAR UN NODO DENDE CALQUERA OUTRO DO CLUSTER	✓	✗	✓
DETECTA ERROS DE CONFIGURACIÓN NOS NODOS EN STANDBY	✗	✗	✗
REQUIRE ABRIR PORTOS ADICIONAIS NO FIREWALL	PACEMAKER + COMUNICACIÓN UDP PARA COROSYNC	✗	REST API PARA PATRONI, MIN. 2 PORTOS PARA O DCS
SOPORTA CONFIGURACIÓN BASEADA EN NAT	✗	✓	✓
PG_REWIND AUTOMÁTICO (RESINCRONIZACIÓN DE NODOS DIVERXENTES)	✗	✓	✓
NOTIFICACIÓN MEDIANTE SCRIPTS DO USUARIO PARA EVENTOS REXISTRADOS	✓	✓	✓
XESTIONA A RECUPERACIÓN DA SAÚDE DOS NODOS INDIVIDUALMENTE	✓	✗	✓
SOPORTA API REST	✗	✗	✓
SOPORTA INTEGRACIÓN CON HAPROXY	✗	✗	✓
 * EVITA QUE UN NODO QUE PERDEU A CONEXIÓN COS DEMAIS SE RECOÑEZA A SI MESMO COMO LÍDER FORMANDO UN CLUSTER EN SOLITARIO E CORROMPA OS DATOS			

Patroni proporcionanos a flexibilidade de escoller o noso DCS e xestionar a creación dos nodos standby, o cal é unha vantaxe sobre os demais.

A integración con **HAProxy** e as posibilidades de monitorización en conxunto con todas as ferramentas e características que ten, fan que sexa a mellor solución para ofrecer a alta dispoñibilidade con PostgreSQL.

#### Que é Patroni exactamente?

Patroni é ao mesmo tempo un xestor de clusters, unha gran plantilla configurable e un kit de ferramentas de código aberto escrito en Python, que nos permite xestionar clusters de PostgreSQL para ofrecer alta dispoñibilidade.

Patroni non implementa un protocolo propio para manter a consistencia do cluster, emprega no seu lugar un almacén de configuración distribuído (**DCS**), algo que explicaremos con detalle no seguinte apartado. Ademais, non nos limita a un en concreto, temos varios entre os que escoller.

Podemos acceder a diversas funcionalidades de Patroni mediante a liña de comandos empregando a súa utilidade *patronictl* e revisar o estado do cluster

grazas á súa integración nativa con HAProxy, usando as APIs que este proporciona para revisar a saúde dos nodos.

Para implementar Patroni precisamos:

- Python 2.7 ou superior
- DCS co seu módulo de python específico instalado. Nun contorno de probas, o DCS pode estar integrado no mesmo nodo ca Patroni, nun contorno de produción o DCS **debe** ser instalado nun nodo independente, xa que se require da máxima estabilidade e dispoñibilidade de recursos para asegurar o seu correcto funcionamento.
- Para a súa correcta configuración require un arquivo *yaml* coas seguintes opcións de alto nivel presentes:
  - Global/Universal: nome do host (único), nome do cluster (scope), path para almacenar a configuración no DCS (namespace).
  - Log: configuracións de log específicas de patroni.
  - Bootstrap: configuración que se lle indicará ao DCS. Inclúe o método de creación dos nodos en standby, os parámetros de inicio da base de datos, scripts de inicio...
  - PostgreSQL: autenticación, directorio de datos, binarios e configuración, direccións IP, portos...
  - REST API: direccións IP, SSL...
  - Un apartado dependendo do DCS que se empregue:
    - Consul
    - Etcd
    - Exhibitor
    - Kubernetes
    - ZooKeeper
    - Watchdog

Patroni é altamente escalable, podendo engadir ao cluster tantas instancias de PostgreSQL como desexemos, neste caso iniciaremos un cluster de 3 nodos: un primario e dúas réplicas.

### 3.4 Seleccionando un DCS

---

Á hora de montar un cluster con Patroni, as ferramentas máis escollidas polas empresas son Etcd e Zookeeper.

Etcd ofrece as seguintes melloras sobre Zookeeper:

- Reconfiguración dinámica da membresía do cluster.

- Operacións de lectura e escritura estables ante cargas altas.
- Un modelo de datos de control de concurrencia multiversión.
- Monitorización de claves altamente confiable e que nunca descarta eventos sen avisar.
- Arrendamento de primitivas desacoplando conexións de sesións.
- APIs para locks compartidos, seguros e distribuídos.
- Soporte para un amplo rango de linguaxes e frameworks de base.

Debido ás melloras que implementa, a cantidade de documentación, recursos e o feito de que sexa un compoñente de Kubernetes (unha tecnoloxía na que me interesa investigar máis no futuro), escolleuse **Etcd** como o DCS que se empregará en conxunto con Patroni.

Etcd é un almacén clave-valor distribuído e de código aberto que se emprega para manter e administrar información crítica que os sistemas distribuídos precisan para funcionar correctamente: datos de configuración, información de estado e, no caso de Kubernetes, metadatos.

### Que aporta Etcd?

- **Replicación total:** nun cluster de Etcd, todos os nodos teñen acceso total ao almacén de datos.
- **Alta dispoñibilidade:** deseñado para non ter SPOF e tolerar cortes da rede e fallos de hardware.
- **Consistente e confiable:** cada lectura dos datos devolve a información escrita máis recente entre todos os clusters.
- **Rápido:** acadando as 10.000 escrituras por segundo en benchmarks.
- **Seguro:** soportando TLS e certificados SSL.
- **Sinxelo:** calquera aplicación pode acceder a el empregando ferramentas estándar HTTP ou JSON.

Os **requisitos principais** de Etcd son que estea instalado en nodos independentes e que se empreguen discos SSD nos contornos nos que este se implemente, xa que o seu rendemento depende enormemente na velocidade do almacenamento.

Os clusters de Etcd empregan o algoritmo de consenso **RAFT**.

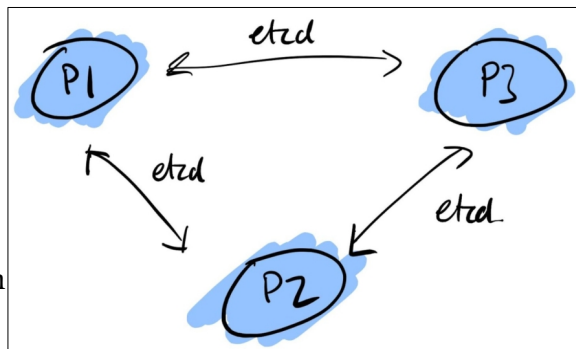
O consenso nun sistema distribuído é un problema fundamental que involucra que múltiples servidores se poñan de acordo nun valor concreto. Unha vez que toman unha decisión nun valor, é final.

Xa que queremos implementar alta dispoñibilidade, non nos chega cun único nodo de Etcd; precisamos respaldo, e, para que os nodos de Etcd cheguen a un acordo sobre cal vai ser o novo líder no caso de fallo ou mantemento do principal, precísase unha maioría.

Para un cluster con  $n$  membros o mínimo de participantes é de  $n/2+1$ , polo que se temos 3 nodos o noso sistema soportará o fallo de 1 nodo e poderá escoller un novo líder.

O número de nodos recomendados para un cluster é de 3, 5 ou 7.

Neste caso, o cluster será de 3 nodos, tantos como instancias de PostgreSQL.



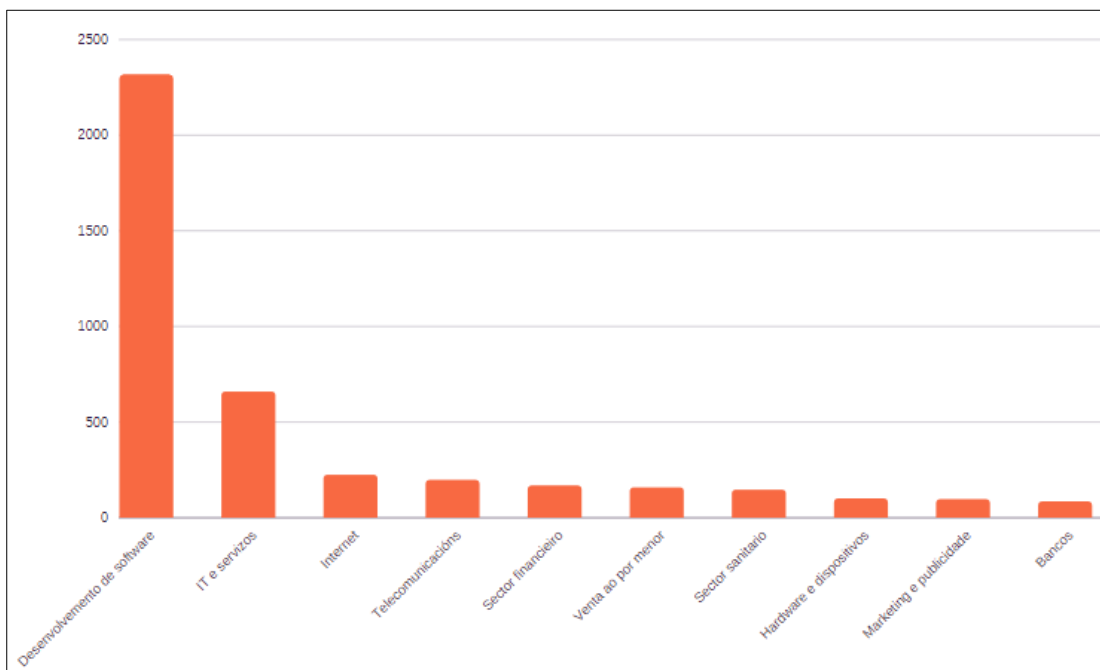
### 3.5 balanceo de carga con HAProxy

Debido a que ofrece failover automático, á súa enorme aceptación na industria, á súa perfecta integración con Patroni, o seu aproveitamento dos recursos (e baixos requisitos) e que proporciona estatísticas en tempo real, optouse directamente por empregar HAProxy como balanceador de carga.

Son moitas as empresas que empregan HAProxy, destacando o propio **IEEE** (Instituto de Enxeñeiros Eléctricos e Electrónicos) ou **Reddit**, o sexto sitio máis visitado do mundo.

A maiores, o seu ámbito de uso é principalmente o sector tecnolóxico.

Na seguinte gráfica pódese ver a súa aceptación segundo o número de empresas:



E como mostra da súa escalabilidade, en Abril deste mesmo ano rexistrouse unha instancia de AWS con HAProxy que acadou unha cifra superior aos dous millóns de peticións HTTP por segundo, todo isto mantendo unha latencia de 560 microsegundos.

### Ligazón ao artigo

Precisamos polo menos dúas instancias de HAProxy na nosa infraestrutura. Xa que os clientes se conectarán ao servizo de base de datos a través del, non nos podemos permitir que haxa un único balanceador de carga, e o respaldo deberá estar preparado para entrar en acción tan pronto como se detecte un fallo do balanceador principal para que as aplicacións ou clientes non se queden sen acceso ao servizo.

HAProxy configurarase para desviar as peticións de lectura á base de datos por un porto determinado, de maneira que se repartan entre os tres nodos e as peticións de lectura e escritura por outro, para que cheguen sempre ao primario.

Para monitorizar HAProxy e proporcionar failover automático empregárase **Keepalived**.

Keepalived é unha ferramenta de software empregada para acadar a alta dispoñibilidade asignando a dous ou máis nodos unha IP virtual e monitorizándoos. No caso de fallo, a IP virtual (a través da cal se conectan os clientes) transferirase dun HAProxy ao outro facendo que o downtime sexa mínimo.

Isto faise mediante o protocolo **VRRP** (Virtual Router Redundancy Protocol), que se asegura de que un dos dous nodos de HAProxy está a funcionar como primario. O nodo de respaldo está á escoita de paquetes multicast dende o primario (que ten maior prioridade). Se o nodo de respaldo deixa de recibir paquetes VRRP por tres veces máis tempo do establecido, asume o rol de primario e asígnase a IP virtual a si mesmo.



No caso de contar con varios nodos de respaldo coa mesma prioridade, aquel coa IP de maior número recibe a IP virtual.

### 3.6 Engadindo un pool de conexións

---

A medida que as aplicacións medran a necesidade dun pool de conexións faise evidente.

Cando unha aplicación envía unha solicitude de conexión a unha base de datos de PostgreSQL esta é recibida polo servizo de BBDD, que observa o límite establecido pola variable *max\_connections* e se bifurca, creando un novo proceso no backend para dar soporte a esta nova conexión. Este proceso do backend permanece activo ata que a conexión é pechada polo cliente ou polo propio PostgreSQL.

Se unha aplicación foi concebida coa base de datos en mente fará un emprego máis efectivo das conexións, reusando as existentes sempre que sexa posible e pechando as que xa non sexan necesarias. Pero este non sempre é o caso, e ademais a medida que unha aplicación aumenta en popularidade e ten unha base de usuarios e unha cantidade de datos maior, a carga de traballo pode saturar o servizo de PostgreSQL.

Hai un límite práctico para o número de conexións que un servidor de bases de datos pode xestionar nun momento concreto. Se se acada este límite o rendemento do servidor pode caer en picado provocando fallos maiores como unha caída do servizo.

Mediante un pool de conexións podemos empregar unha caché (pool) de conexións que se manteñen abertas co servidor de bases de datos e que son reutilizadas polas distintas peticións dos clientes evitando ter que abrir novos procesos no backend constantemente.

Isto leva a un mellor aproveitamento dos recursos, aumentando o número de transaccións que se poden levar a cabo e aumentando a velocidade á que se fan.

Realizouse unha comparación entre PgPool-II e PGBouncer revisando as súas funcionalidades para decidir cal se integraría co resto da pía tecnolóxica:

	PGBOUNCER	PGPOOL-II
CONSUMO DE RECURSOS	Moi lixeiro (emprega un só proceso), garantizando un efecto mínimo na memoria do sistema, ata cando ten que dar soporte a BBDD de gran tamaño	N conexións paralelas bifurcan N procesos fillos (32 por defecto)
CANDO SE REUTILIZAN CONEXIÓNS?	Define un pool por combinación de usuario+base de datos, compartido entre todos os clientes, polo que todos dispoñen dunha conexión parte do pool	Un proceso por proceso fillo. Non se pode controlar a que proceso fillo se conecta un cliente. O cliente non sempre se beneficia do pool.
MODOS DE CONNECTION POOLING	Por sesión, transacción e sentenza, gran flexibilidade	Sesión. A eficacia depende do comportamento do cliente.
ALTA DISPOÑIBILIDADE	Non soportado - PGBouncer recomenda HAProxy para realizar o balanceo de carga	Mediante procesos de monitorización integrados
BALANCEO DE CARGA	Non soportado - PGBouncer recomenda HAProxy para realizar o balanceo de carga	Balanceo de carga automático
SOPORTE MULTI-CLUSTER	Unha instancia podería dar servizo a varios clusters (só en escenarios específicos)	Non ten
CONTROL DE CONEXIÓNS	Permite limitar o número de conexións por pool, por base de datos, por usuario e por cliente	Permite limitar o número de conexións total
COLA DE CONEXIÓNS	Soportado a nivel de aplicación (mantena o propio PGBouncer)	Soportado a nivel de kernel
REPLICACIÓN LÓXICA	Non soportado a través de PGBouncer	Soportado. Envía as consultas de escritura a todos os nodos. Xeralmente non recomendado
ADMINISTRACIÓN	Base de datos virtual con estadísticas	Interface de administración, inclúe GUI

Neste caso optouse por PGBouncer por ser unha solución moito máis lixeira en canto ao consumo de recursos e flexible ca PGPool-II. Dado que a replicación\*, o balanceo de carga e o autofailover xa están cubertos por ferramentas dedicadas e máis eficaces descritas en apartados anteriores, PGBouncer é o ideal.

*\*PGPOOL-II non é fiable porque as sentencias SELECT poderían modificar datos se fan chamadas a funcións VOLATILE (unha función deste tipo pode facer de todo, ata cambios na base de datos)*

Outro factor de peso para decantarse por PGBouncer é poder escoller o modo de pooling, xa que dependendo da aplicación pode ser máis interesante un ou outro.

#### Métodos de pooling con PGBouncer:

- O **pooling transaccional** beneficia a aplicacións que manteñan sesións en idle. Con este modo PGBouncer non precisa manter sesións abertas e en idle, pode sinxelamente coller unha do pool cando se inicie unha transacción. Esas sesións en idle só consumen unha conexión de

PGBouncer, no lugar dunha sesión real de PostgreSQL co backend gastando memoria e sen sincronizar nada.

- O **pooling de sesións** é útil se se van a realizar consultas preparadas ou se se proporcionan funcionalidades que funcionan a nivel de sesión.
- O **pooling de sentencias** retorna a conexión cada vez que se realiza unha consulta, polo que as transaccións con varias sentencias non funcionarán correctamente.

Configurarase PGBouncer diante de cada unha das instancias de PostgreSQL (no mesmo nodo) e para que proporcione pooling transaccional.

### 3.7 PGBackRest: o repositorio de backups

---

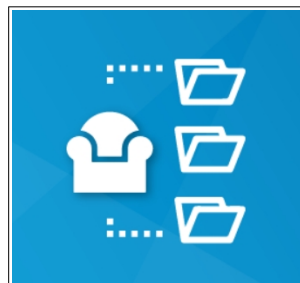
**PGBackRest** é unha das ferramentas de código aberto máis populares para PostgreSQL grazas á súa eficacia tratando con grandes volumes de datos e polos seus elaborados mecanismos de validación de backups.

Para cada copia de seguridade realízase un checksum a cada arquivo no momento da realización e no momento da restauración.

PgBackRest permite facer copias completas, incrementais e diferenciais e facilita a súa limpeza con mecanismos como por exemplo a posibilidade de limitar o número de copias completas que se manteñen á vez.

Coa funcionalidade *delta restore* podemos restaurar bases de datos de gran tamaño sen ter que restablecer o cluster previamente.

Unha **stanza** é a definición de todos os parámetros requiridos para a replicación dun cluster de bases de datos. Na stanza defínese onde se atopan os nodos do cluster, os parámetros de conexión coa BBDD, como se fan os backups, opcións de arquivamento...



Un servidor de PostgreSQL conta coa súa propia stanza, mentras que os servidores de backup contan cunha stanza por cada cluster ao que respaldan.

Un **repositorio** é un sitio onde pgbackrest almacena os WALs e as copias de seguridade, que ademais pode estar protexido mediante cifrado simétrico.

Unha razón de peso á hora de escoller esta ferramenta é que se integra perfectamente con Patroni, **permitindo lanzar un cluster a partires dun backup**, configurando a [sección de bootstrap de Patroni](#).

Outra razón é que PGBackRest **detecta** cal é o nodo primario cando se dispón a facer unha copia de seguridade, polo que no caso de que se producise un failover por caída do servizo, realizaríanse as copias sobre o nodo promocionado automaticamente.

### 3.8 Deseño da infraestrutura

---

En resumo, teríamos un cluster de 3 nodos con Etcd, o DCS que axudará a Patroni a elixir o servidor primario, promocionar un secundario no caso de fallo e reintroducir no cluster os nodos recuperados.

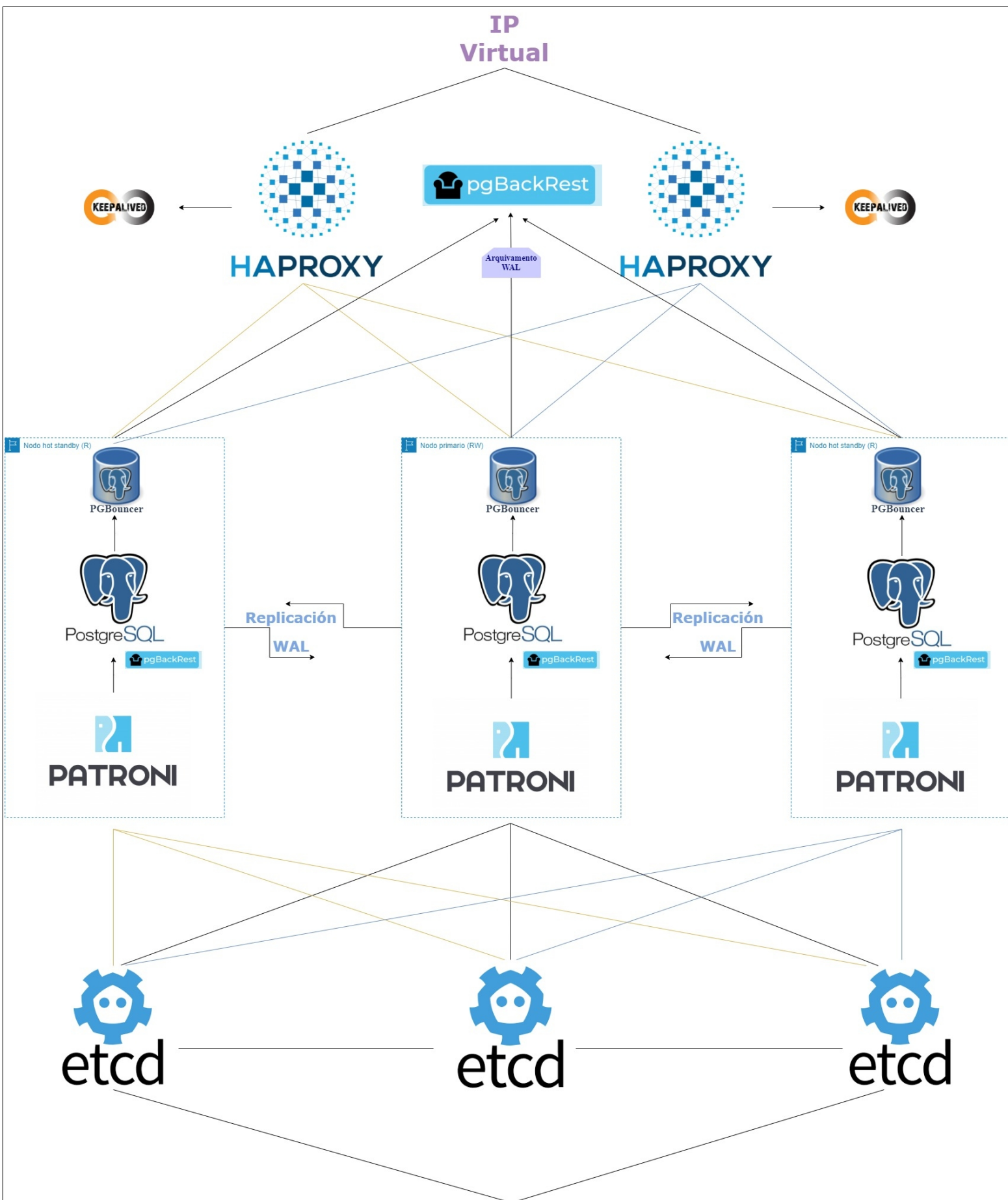
Tres nodos con Patroni + PostgreSQL + PGBouncer + PGBackRest, un primario que ofrecerá lectura e escritura e dous secundarios (*hot standby*) que replicarán os seus datos mediante WAL e ofrecerán servizo de lectura.

Un nodo que funcionará de repositorio central con PGBackRest que empregará o arquivado de WAL para permitir restablecer o cluster ou desplegar novos clusters a partir de backups en caso de necesidade.

Dous nodos con HAProxy como balanceador de carga, que redirixirán peticións de lectura a todos os nodos e as de lectura e escritura ao primario. O servizo estará monitorizado por Keepalived, o cal se encargará de revisar o estado de HAProxy en todo momento e transferir a VIP dun nodo a outro.

Un cliente ou aplicación, teóricamente conectaríase a través da IP virtual e un porto concreto ao balanceador de carga (principal nese momento), que reenviará a súa petición ao nodo primario de Patroni e ao porto no que escoita PGBouncer, para que este conecte co servizo de PostgreSQL.

**A continuación elaborouse un esquema do deseño final da infraestrutura.**





## 4 Implementación

### 4.1 Sistemas

Para a elaboración do contorno de probas empregaremos 9 máquinas con Debian 10.6 sen GUI conectadas nunha rede interna. As especificacións delas son:

- CPU single core 2GHz ou superior
- 512MB RAM
- 30GB de espazo en disco

### 4.2 Táboa de direccionamento

Equipo	Interface	Dirección IPv4	Máscara	Gateway
postgres1	enp0s3	192.168.100.1	255.255.255.0	N/D
postgres2	enp0s3	192.168.100.2	255.255.255.0	N/D
postgres3	enp0s3	192.168.100.3	255.255.255.0	N/D
haproxy1	enp0s3	192.168.100.5	255.255.255.0	N/D
haproxy2	enp0s3	192.168.100.6	255.255.255.0	N/D
etcd1	enp0s3	192.168.100.4	255.255.255.0	N/D
etcd2	enp0s3	192.168.100.7	255.255.255.0	N/D
etcd3	enp0s3	192.168.100.8	255.255.255.0	N/D
pgbackrestRepo	enp0s3	192.168.100.9	255.255.255.0	N/D

### 4.3 Creación do cluster de Etcd

O primeiro paso é instalar etcd en cada un dos nodos:

```
$ apt-get install etcd
```

A continuación editaremos a configuración do servizo en cada nodo.





Exemplo da configuración en **etcd1**:

```
$ nano /etc/default/etcd
```

-----

#a IP e o porto no que o servizo aceptará as peticións entrantes dos outros membros do cluster:

```
ETCD_LISTEN_PEER_URLS="http://192.168.100.4:2380"
```

#Listado de direccións para recibir peticións dos clientes:

```
ETCD_LISTEN_CLIENT_URLS="http://localhost:2379,http://192.168.100.4:2379"
```

#Dirección IP deste membro para comunicalo co resto do cluster:

```
ETCD_INITIAL_ADVERTISE_PEER_URLS="http://192.168.100.4:2380"
```

#Configuración inicial do cluster para o arranque (manter este orde en todos os nodos, sen espazos):

```
ETCD_INITIAL_CLUSTER="etcd1=http://192.168.100.4:2380,etcd2=http://192.168.100.7:2380,etcd3=http://192.168.100.8:2380"
```

```
ETCD_INITIAL_CLUSTER_STATE="new"
```

#Token inicial do cluster:

```
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"
```

#Dirección IP deste membro para comunicarlle aos clientes:

```
ETCD_ADVERTISE_CLIENT_URLS="http://192.168.100.4:2379"
```

-----

O directorio de datos de etcd debe ser accesible por systemd:

```
$ sudo mkdir -p /var/lib/etcd
```

```
$ sudo chown -R root:$(whoami) /var/lib/etcd
```

```
$ sudo chmod -R a+rw /var/lib/etcd
```

Habilitamos e arrancamos o servizo:

```
$ systemctl enable etcd
```

```
$ systemctl start etcd
```

Configuramos a variable **ETCDCTL\_API** para obter funcionalidades extendidas para a ferramenta de consola de comandos:

```
$ export ETCDCTL_API=3
```

Revisamos o estado do cluster con:

```
$ etcdctl cluster-health
```

```
dcastro@etcd1:/usr/local$ etcdctl cluster-health
member 1022f5f84b872003 is healthy: got healthy result from http://192.168.100.8:2379
member 6d9f6395a49be653 is healthy: got healthy result from http://192.168.100.4:2379
member e5c4776e80d3063e is healthy: got healthy result from http://192.168.100.7:2379
cluster is healthy
dcastro@etcd1:/usr/local$ _
```

Para ver cal é o líder actual:

```
$ etcdctl member list
```

```
dcastro@etcd1:/usr/local$ etcdctl member list
1022f5f84b872003: name=etcd3 peerURLs=http://192.168.100.8:2380 clientURLs=http://192.168.100.8:2379
isLeader=false
6d9f6395a49be653: name=etcd1 peerURLs=http://192.168.100.4:2380 clientURLs=http://192.168.100.4:2379
isLeader=true
e5c4776e80d3063e: name=etcd2 peerURLs=http://192.168.100.7:2380 clientURLs=http://192.168.100.7:2379
isLeader=false
dcastro@etcd1:/usr/local$ _
```

Para transferir o rol de mestre a outro nodo de forma manual:

```
$ etcdctl move-leader (id-membro)
```

Para engadir un novo membro:

```
$ etcdctl member add
```

Se o cluster se iniciou incorrectamente e os nodos non se recoñecen entre si, hai que borrar o directorio `/var/lib/etcd` para que se poida reiniciar o recoñecemento dende cero.

#### 4.4 Configuración do cluster de Patroni

---

En cada nodo debemos realizar o mesmo procedemento. Comezaremos coa instalación de PostgreSQL e os paquetes de python, que precisamos xa que é a linguaxe na que está escrito Patroni:

```
$ sudo apt-get -y install postgresql-11 postgresql-server-dev-11
python python-pip
```

É importante verificar que non se estea a executar PostgreSQL:

```
$ sudo systemctl stop postgresql
```

Patroni emprega ferramentas incluídas en PostgreSQL, polo que faremos un enlace simbólico para que poida atopalas:

```
$ sudo ln -s /usr/lib/postgresql/11/bin/* /usr/bin/
```

Comprobamos que temos a última versión do paquete setuptools e instalamos psycopg2:

```
$ sudo -H pip install --upgrade setuptools && sudo -H pip install psycopg2
```

Instalamos patroni e python-etcd:

```
$ sudo -H pip install patroni python-etcd
```

Instalamos pgbackrest:

```
$ sudo apt install pgbackrest
```

Creamos e editamos o arquivo de configuración de Patroni (nodo **postgres1**):

```
$ sudo nano /etc/patroni.yml
```

-----

**Scope** especifica o nome do cluster, **namespace** é o path dentro do almacén de configuración onde se gardará a información do cluster e **name** é o nome do host (debe ser único no cluster).

```
scope: postgres
namespace: /db/
name: postgres1
```

Configuramos a API REST de Patroni, que é o mecanismo que o propio Patroni emprega para realizar os failovers.

A API REST é empregada tamén por HAProxy para realizar as revisións de saúde dos nodos mediante peticións HTTP.

```
restapi:
  listen: 192.168.100.1:8008
  connect_address: 192.168.100.1:8008
```

No apartado de `etcd` indicamos a IP e o porto no que escoita cada un dos nodos do cluster e a continuación comezamos a sección de **bootstrap**, onde no apartado “**dcs**” se especifican parámetros como *ttr* (*tempo que ten que pasar para que se inicialice o failover automático*), *retry\_timeout* (especifica que os fallos de rede que duren menos do especificado non provocarán que Patroni lle quite ao nodo primario o liderazgo).

```
etcd:
  hosts: 192.168.100.4:2379,192.168.100.7:2379,192.168.100.8:2379

bootstrap:
  dcs:
    ttl: 30
    loop_wait: 10
    retry_timeout: 10
    maximum_lag_on_failover: 1048576
    postgresql:
      use_pg_rewind: true

  initdb:
    - encoding: UTF8
    - data-checksums
```

Habilitamos o uso de `pg_rewind` (ferramenta de resincronización dos nodos do cluster).

No apartado **initdb** especificamos parámetros de arranque de postgres como a codificación.

Por último e para rematar co apartado de **bootstrap**, configuramos os parámetros do arquivo `pg_hba` de postgres, que é o arquivo donde se habilita a autenticación de clientes ao servidor de postgres e é o que se empregará para que se repliquen as bases de datos mediante a retransmisión WAL.

```
pg_hba:
- host replication replicator 127.0.0.1/32 md5
- host replication replicator 192.168.100.1/0 md5
- host replication replicator 192.168.100.2/0 md5
- host replication replicator 192.168.100.3/0 md5
- host all all 0.0.0.0/0 md5

users:
  admin:
    password: admin
    options:
      - createrole
      - createdb
```

Agora imos a configurar os parámetros de arranque de PostgreSQL, especificando a IP e o porto no que escoitará, o directorio de datos, as credenciais do superusuario e do usuario **replicator** (indicado no apartado de `pg_hba`), **que se empregará para a replicación**.

Habilitamos o arquivado de WAL (necesario para PgBackRest máis adiante) e establecemos o comando co nome da stanza.

Ademais, no apartado de **tags** especificaremos detalles como o failover e o balanceo de carga.

```
postgresql:
  listen: 192.168.100.1:5432
  connect_address: 192.168.100.1:5432
  data_dir: /data/patroni
  pgpass: /tmp/patroni
  authentication:
    replication:
      username: replicator
      password: abc123.
    superuser:
      username: postgres
      password: superdbadm
  parameters:
    unix_socket_directories: '.'
    archive_mode: "on"
    archive_command: 'pgbackrest --stanza=produccion archive-push %p'
  tags:
    nofailover: false
    noloadbalance: false
    clonefrom: false
    nosync: false
```

A configuración do arquivo yaml de patroni é idéntico en **postgres2** e **postgres3**, cambiando as IP según corresponda e o nome do nodo.

Creamos o directorio de datos de patroni:

```
$ sudo mkdir -p /data/patroni
```

Facemos que postgres sexa o dono e poida acceder ao directorio:

```
$ sudo chown postgres:postgres /data/patroni
```

Aseguramos os permisos para que só poida acceder o usuario Postgres:

```
$ sudo chmod 700 /data/patroni
```

Agora só nos falta crear o daemon de Patroni, polo que faremos un script de systemd:

```
$ sudo nano /etc/systemd/system/patroni.service
```

Configuramos o script para que se execute cando a rede estea operativa, especificamos o directorio de patroni e o arquivo yaml coa configuración de arranque e o usuario que o inicia (postgres)

```
[Unit]
Description=PostgreSQL HA Cluster
After=syslog.target network.target

[Service]
Type=simple
User=postgres
Group=postgres
ExecStart=/usr/local/bin/patroni /etc/patroni.yml
KillMode=process
TimeoutSec=30
Restart=no

[Install]
WantedBy=multi-user.target
```

## 4.5 Configuración de PGBouncer

En cada nodo con Patroni e PostgreSQL, configuraremos PGBouncer. A continuación o setup no nodo **postgres1**:

```
$ sudo nano /etc/pgbouncer/pgbouncer.ini
```

o noso cluster parte dunha base de datos chamada aplicación que ten como dono a un usuario co mesmo nome, polo que hai que indicarlle a pgbouncer que esas conexións se fagan a través del:

```
[databases]
aplicacion = host=192.168.100.1 user=aplicacion dbname=aplicacion
```

Establecemos os parámetros de escoita do servizo:

```
; IP address or * which means all IPs
listen_addr = *
listen_port = 6432
```

Indicamos o tipo de autentificación e o arquivo que contén as credenciais:

```
; any, trust, plain, crypt, md5, cert, hba, pam
auth_type = md5
auth_file = /etc/pgbouncer/userlist.txt
```

```
; comma-separated list of users, who are allowed to change settings
admin_users = postgres, aplicacion
```

Configuramos o pooling transaccional:

```
; When server connection is released back to pool:
; session      - after client disconnects
; transaction  - after transaction finishes
; statement    - after statement finishes
pool_mode = transaction
```

Establecemos o tamaño do pool e o máximo de conexións:

```
; total number of clients that can connect
max_client_conn = 300

; default pool size. 20 is good number when transaction pooling
; is in use, in session pooling it needs to be the number of
; max clients you want to handle at any moment
default_pool_size = 35
```

O contido do arquivo coas credenciais de usuario:

```
$ sudo nano /etc/pgbouncer/userlist.txt
```

```
"aplicacion" "abc123."
```

*nun contorno de produción sería recomendable empregar a autentificación mediante un contrasinal encriptado ou con certificados dixitais.*



Unha vez estean configurados os tres nodos de patroni podemos arrancar o servizo:

```
$ sudo systemctl start patroni
```

```
$ sudo systemctl start pgbouncer
```

Mediante a ferramenta de consola de patroni podemos revisar o estado do cluster:

```
postgres@postgres1:/data/patroni$ patronictl -c /etc/patroni.yml list
+ Cluster: postgres (6952525831333962433) -----+-----+-----+
| Member   | Host           | Role   | State   | TL | Lag in MB |
+-----+-----+-----+-----+-----+-----+
| postgres1 | 192.168.100.1 | Leader | running | 4  |           |
| postgres2 | 192.168.100.2 | Replica | running | 4  | 0.0        |
| postgres3 | 192.168.100.3 | Replica | running | 2  | 0.0        |
+-----+-----+-----+-----+-----+-----+
postgres@postgres1:/data/patroni$
```

Revisamos o estado de Patroni no primario:

```
root@postgres1:~# systemctl status patroni
● patroni.service - PostgreSQL HA Cluster
   Loaded: loaded (/etc/systemd/system/patroni.service; disabled; vendor preset: enabled)
   Active: active (running) since Sun 2021-06-06 17:00:25 CEST; 1min 2s ago
     Main PID: 564 (patroni)
        Tasks: 16 (limit: 545)
       Memory: 143.0M
      CGroup: /system.slice/patroni.service
              └─564 /usr/bin/python /usr/local/bin/patroni /etc/patroni.yml
                └─583 postgres -D /data/patroni --config-file=/data/patroni/postgresql.conf --listen_addr
                  └─586 postgres: postgres: checkpoint
                    └─587 postgres: postgres: background writer
                      └─588 postgres: postgres: stats collector
                        └─591 postgres: postgres: postgres 192.168.100.1(34810) idle
                          └─602 postgres: postgres: walwriter
                            └─603 postgres: postgres: autovacuum launcher
                              └─604 postgres: postgres: archiver last was 000000010000000000000007.partial
                                └─605 postgres: postgres: logical replication launcher
                                  └─640 postgres: postgres: walsender replicator 192.168.100.2(34082) streaming 0/7000250
                                    └─643 postgres: postgres: walsender replicator 192.168.100.3(50974) streaming 0/7000250
```

```
Jun 06 17:00:39 postgres1 patroni[564]: 2021-06-06 17:00:39,332 INFO: no action. I am the leader
```

Revisamos os standby:

```
root@postgres2:~# systemctl status patroni
● patroni.service - PostgreSQL HA Cluster
   Loaded: loaded (/etc/systemd/system/patroni.service; disabled; vendor preset: enabled)
   Active: active (running) since Sun 2021-06-06 17:00:31 CEST; 50s ago
     Main PID: 564 (patroni)
        Tasks: 12 (limit: 545)
       Memory: 131.3M
      CGroup: /system.slice/patroni.service
              └─564 /usr/bin/python /usr/local/bin/patroni /etc/patroni.yml
                └─581 postgres -D /data/patroni --config-file=/data/patroni/postgresql.conf --listen_addr
                  └─583 postgres: postgres: startup recovering 000000020000000000000007
                    └─584 postgres: postgres: checkpoint
                      └─585 postgres: postgres: background writer
                        └─586 postgres: postgres: stats collector
                          └─594 postgres: postgres: postgres 192.168.100.2(49336) idle
                            └─610 postgres: postgres: walreceiver streaming 0/7000250
```

```
Jun 06 17:00:59 postgres2 patroni[564]: 2021-06-06 17:00:59,374 INFO: no action. I am a secondary
```

## 4.6 Configuración de HAProxy Keepalived

Esta configuración é a do nodo **haproxy1**, a configuración do secundario é idéntica **salvo excepcións que en todo caso estarán especificadas**.

Instalamos HAProxy, Keepalived e paquetes adicionais:

```
$ sudo apt-get install keepalived psmisc haproxy
```

Configuramos HAProxy:

```
$ sudo nano /etc/haproxy/haproxy.cfg
```

Establecemos o límite de conexións que aceptará HAProxy, isto dependerá dos recursos dos que dispoñamos.

Indicámoslle coa directiva *log global* que habilite o logging para todos os proxies que se configuren.

Configuramos HAProxy para que funcione como un proxy na capa de transporte. Indicámoslle o tempo de espera para que se estableza unha conexión co backend e con *timeout server* medimos a inactividade do mesmo.

```
maxconn 100

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

Cando traballamos con HAProxy na capa 4, é preciso especificar un timeout idéntico para o cliente e o servidor porque HAProxy non sabe cal dos dous debe estar remitindo comunicacións.

O balanceador de carga terá dispoñible un panel de monitorización que será accesible a través da(s) súa(s) IP(s) e o porto 7000.

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

Configuramos a sección que conectará aos clientes co servidor de PostgreSQL primario para peticións de lectura e escritura. Como dirección poñeremos a IP virtual que asignaremos máis adiante mediante Keepalived e como porto o 5000.

HAProxy empregará peticións HTTP (**option httpchk**) para revisar o estado e amosalo no panel.

Para cada nodo especificaremos que revise o estado dos nodos con Patroni especificándolle o porto da **API REST** de Patroni.

No caso de que un nodo falle HAProxy redirixirá as peticións ao nodo promocionado, o tempo será de 3 segundos de cambio e 2 segundos para que o novo nodo apareza como “levantado”.

Hai que especificar as IP dos nodos de Patroni/PostgreSQL xunto co porto de PGBouncer e o máximo de conexións que imos permitir para cada un deles, o cal, unha vez máis depende dos recursos dos que dispoñamos.

No listen (parte superior) é recomendable poñer un nome descriptivo para dar máis claridade ao panel de administración.

```
listen nodo-activo
bind 192.168.100.56:5000
option httpchk OPTIONS/master
http-check expect status 200
default-server inter 3s fall 3 rise 2 on-marked-down shutdown-sessions
server postgresql_192.168.100.1_6432 192.168.100.1:6432 maxconn 100 check port 8008
server postgresql_192.168.100.2_6432 192.168.100.2:6432 maxconn 100 check port 8008
server postgresql_192.168.100.3_6432 192.168.100.3:6432 maxconn 100 check port 8008
```

O parámetro **on-marked-down shutdown-sessions** fará que HAProxy peche as conexións co backend existentes cando deixe de funcionar, xa que nos interesa que se reconecte o máis rápido posible cun nodo funcional.

Agora configuramos o mesmo para as conexións aos nodos en standby (só lectura), que serán tamén a través da IP virtual e polo porto 5001 e se repartirán facendo un **Round Robin** entre todos eles.

```
listen nodos-replica-standby
bind 192.168.100.56:5001
option httpchk OPTIONS/replica
http-check expect status 200
default-server inter 3s fall 3 rise 2 on-marked-down shutdown-sessions
server postgresql_192.168.100.1_6432 192.168.100.1:6432 maxconn 100 check port 8008
server postgresql_192.168.100.2_6432 192.168.100.2:6432 maxconn 100 check port 8008
server postgresql_192.168.100.3_6432 192.168.100.3:6432 maxconn 100 check port 8008
```

---

Antes de configurar Keepalived precisamos habilitar en Linux a posibilidade de asociar direccións IP non locais ao equipo, xa que a primeira vez que arranquemos HAProxy co servizo Keepalived no nodo de respaldo, este detectará que a IP xa está asociada ao mestre e o servizo entrará nun estado de erro.

Para isto hai que editar o arquivo `/etc/sysctl.conf`:

```
net.ipv4.ip_nonlocal_bind = 1
```

```
net.ipv6.ip_nonlocal_bind = 1
```

E recargar a configuración con:

```
$/usr/bin/sysctl -p
```

Crear un usuario do sistema chamado `keepalived_script`:

```
$ useradd keepalived_script
```

Configuramos KeepAlived:

```
$ sudo nano /etc/keepalived/keepalived.conf
```

O primeiro será habilitar `script_security` para que un usuario non root poida executar os scripts que empregará Keepalived para revisar o servizo.

```
global_defs {  
    enable_script_security  
}
```

Agora especificamos o script que Keepalived executará e o intervalo no que o fará, para revisar o estado de HAProxy.

No caso de que Keepalived detecte que o HAProxy mestre se caíu, aumentará a súa prioridade en 4 puntos no secundario e reduciráa no mestre, polo que o secundario promocionará e transferirase a IP virtual.

```
vrrp_script chk_haproxy { #necesita keepalived  
    script "/etc/keepalived/revisar_haproxy.sh"  
    interval 2          #revisa o servizo cada 2s  
    weight 4 #o peso modifica o valor da prioridade (+4 éxito, -4 erro)  
}
```

O último paso é **definir a instancia do protocolo VRRP** que se executará na interface de rede de cada nodo xunto coa **IP virtual** que se vai asociar, o nome identificativo do **script** que revisa o servizo e a prioridade base para cada un e o estado do que parten.

Hai que **especificar o método de autenticación de VRRP** e unha **ID para o router virtual**.

Este é o único apartado que difire entre o principal e o respaldo.

No mestre escribimos:

```
vrrp_instance VI_1 {
    interface enp0s3
    state MASTER #este srv é o haproxy principal
    priority 101 #a prioridade do master ten que ser superior á do slave
    virtual_router_id 51
    authentication {
        auth_type PASS
        auth_pass abc123.
    }
    virtual_ipaddress {
        192.168.100.56
    }
    track_script {
        chk_haproxy
    }
}
```

No secundario:

```
vrrp_instance VI_1 {
    interface enp0s3
    state BACKUP #este srv é o haproxy de respaldo
    priority 100 #a prioridade do master ten que ser superior á do slave
    virtual_router_id 51
    authentication {
        auth_type PASS
        auth_pass abc123.
    }
    virtual_ipaddress {
        192.168.100.56
    }
    track_script {
        chk_haproxy
    }
}
```

Xa só queda crear un script dentro do mesmo directorio que o arquivo de configuración. O script é idéntico en ambos nodos :

```
GNU nano 3.2 /etc/keepalived/revisar_haproxy.sh

#!/bin/bash

STATUS=$(systemctl is-active haproxy.service)

if [ "${STATUS}" = "active" ]
then
    exit 0 #exit code 0 = éxito
else
    exit 1 #exit code 1 = erro
fi
```

O único que fai é revisar o estado do servizo e devolver un exit code, que é o que precisa Keepalived para aumentar ou reducir a prioridade do nodo.

Arrancamos HAProxy e Keepalived nos dous nodos, comezando polo mestre.

```
$ systemctl enable haproxy && systemctl start haproxy
```

```
$ systemctl enable keepalived && systemctl start keepalived
```

Revisamos o estado de keepalived e vemos como concordan os roles e as prioridades efectivas no primario e no respaldo:

```
root@haproxy1:~# systemctl status keepalived.service
● keepalived.service - Keepalive Daemon (LVS and VRRP)
   Loaded: loaded (/lib/systemd/system/keepalived.service; enabled; vendor preset: enabled)
   Active: active (running) since Wed 2021-06-16 19:54:15 CEST; 3h 0min ago
 Main PID: 514 (keepalived)
    Tasks: 2 (limit: 545)
   Memory: 11.0M
   CGroup: /system.slice/keepalived.service
           └─514 /usr/sbin/keepalived --dont-fork
             └─535 /usr/sbin/keepalived --dont-fork

jun 16 19:54:15 haproxy1 Keepalived[514]: Opening file '/etc/keepalived/keepalived.conf'.
jun 16 19:54:15 haproxy1 Keepalived[514]: Starting VRRP child process, pid=535
jun 16 19:54:15 haproxy1 Keepalived_vrrp[535]: Registering Kernel netlink reflector
jun 16 19:54:15 haproxy1 Keepalived_vrrp[535]: Registering Kernel netlink command channel
jun 16 19:54:15 haproxy1 Keepalived_vrrp[535]: Opening file '/etc/keepalived/keepalived.conf'.
jun 16 19:54:15 haproxy1 Keepalived_vrrp[535]: Registering gratuitous ARP shared channel
jun 16 19:54:15 haproxy1 Keepalived_vrrp[535]: (VI_1) Entering BACKUP STATE (init)
jun 16 19:54:15 haproxy1 Keepalived_vrrp[535]: VRRP_Script(chk_haproxy) succeeded
jun 16 19:54:15 haproxy1 Keepalived_vrrp[535]: (VI_1) Changing effective priority from 101 to 105
jun 16 19:54:19 haproxy1 Keepalived_vrrp[535]: (VI_1) Entering MASTER STATE
root@haproxy1:~#
```

```
root@haproxy2:~# systemctl status keepalived
● keepalived.service - Keepalive Daemon (LVS and VRRP)
   Loaded: loaded (/lib/systemd/system/keepalived.service; enabled; vendor preset: enabled)
   Active: active (running) since Wed 2021-06-16 19:54:18 CEST; 3h 2min ago
 Main PID: 514 (keepalived)
    Tasks: 2 (limit: 545)
   Memory: 11.0M
   CGroup: /system.slice/keepalived.service
           └─514 /usr/sbin/keepalived --dont-fork
             └─535 /usr/sbin/keepalived --dont-fork

jun 16 19:54:18 haproxy2 Keepalived[514]: Command line: '/usr/sbin/keepalived' '--dont-fork'
jun 16 19:54:18 haproxy2 Keepalived[514]: Opening file '/etc/keepalived/keepalived.conf'.
jun 16 19:54:18 haproxy2 Keepalived[514]: Starting VRRP child process, pid=535
jun 16 19:54:18 haproxy2 Keepalived_vrrp[535]: Registering Kernel netlink reflector
jun 16 19:54:18 haproxy2 Keepalived_vrrp[535]: Registering Kernel netlink command channel
jun 16 19:54:18 haproxy2 Keepalived_vrrp[535]: Opening file '/etc/keepalived/keepalived.conf'.
jun 16 19:54:18 haproxy2 Keepalived_vrrp[535]: Registering gratuitous ARP shared channel
jun 16 19:54:18 haproxy2 Keepalived_vrrp[535]: (VI_1) Entering BACKUP STATE (init)
jun 16 19:54:18 haproxy2 Keepalived_vrrp[535]: VRRP_Script(chk_haproxy) succeeded
jun 16 19:54:18 haproxy2 Keepalived_vrrp[535]: (VI_1) Changing effective priority from 100 to 104
root@haproxy2:~# _
```

Por suposto o HAProxy debe estar executándose correctamente:

```
root@haproxy1:~# systemctl status haproxy
● haproxy.service - HAProxy Load Balancer
   Loaded: loaded (/lib/systemd/system/haproxy.service; enabled; vendor preset: enabled)
   Active: active (running) since Wed 2021-06-16 19:54:15 CEST; 3h 3min ago
     Docs: man:haproxy(1)
           file:/usr/share/doc/haproxy/configuration.txt.gz
  Process: 510 ExecStartPre=/usr/sbin/haproxy -f $CONFIG -c -q $EXTRA_OPTS (code=exited, status=0/SUCCESS)
 Main PID: 522 (haproxy)
    Tasks: 2 (limit: 545)
   Memory: 5.1M
   CGroup: /system.slice/haproxy.service
           └─522 /usr/sbin/haproxy -Ws -f /etc/haproxy/haproxy.cfg -p /run/haproxy.pid
             └─530 /usr/sbin/haproxy -Ws -f /etc/haproxy/haproxy.cfg -p /run/haproxy.pid

jun 16 19:54:15 haproxy1 systemd[1]: Starting HAProxy Load Balancer...
jun 16 19:54:15 haproxy1 systemd[1]: Started HAProxy Load Balancer.
lines 1-15/15 (END)
```



Se accedemos dende o navegador ao panel de administración  
(192.168.100.56:7000) podemos ver o estado do cluster tal e como o configuramos:

HAProxy version 1.8.19-1+deb10u3, released 2020/08/01

Statistics Report for pid 531

> General process information

pid = 531 (process #1, nproc = 1, nthread = 1)

uptime = 0s (idle=0s)

system limits: memmax = unlimited; ulimit-n = 219

maxsock = 219; maxconn = 100; maxpipes = 0

current conns = 1; current pipes = 0/0; conn rate = 1/sec

Running tasks: 1/11; idle = 100 %

active UP

active UP, going down

active DOWN, going up

active or backup DOWN

active or backup DOWN for maintenance (MAINT)

active or backup SOFT STOPPED for maintenance

Note: "NOLB"/"DRAIN" = UP with load-balancing disabled.

backup UP

backup UP, going down

backup DOWN, going up

not checked

Display option:

• Scope :

• Hide DOWN servers

• Refresh now

• CSV export

External resources:

• Primary file

• Updates (v1.8)

• Online manual

state

	Queue			Session rate			Sessions			Bytes			Denied			Errors			Warnings			Status	LastChk	Server				
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	Ln	Tot	Ln	Tot	Ln	Tot	Ln	Tot	Ln	Tot	Wght			Act	Bck	Chk	Dwn	Dwntme
Frontend	1	1	-	1	1	2 000	1	0	0	0	0	0	0	0	0	0	0	0	0	0	OPEN							
Backend	0	0	0	0	0	0	200	0	0	0	0	0	0	0	0	0	0	0	0	0	4m7s UP		0	0	0	0	0	

node-active

Queue		Session rate		Sessions		Bytes		Denied		Errors		Warnings		Status		Server	
	Cur	Max	Limit	Cur	Max	Limit	Total	Ln	Tot	Ln	Tot	Ln	Tot	Ln	Tot	Ln	Tot
Frontend	0	0	-	0	0	0	2 000	0	0	0	0	0	0	0	0	0	0
postgres_192.168.100.1_6432	0	0	-	0	0	0	100	0	0	0	0	0	0	0	0	0	0
postgres_192.168.100.2_6432	0	0	-	0	0	0	100	0	0	0	0	0	0	0	0	0	0
postgres_192.168.100.3_6432	0	0	-	0	0	0	100	0	0	0	0	0	0	0	0	0	0
Backend	0	0	0	0	0	0	200	0	0	0	0	0	0	0	0	0	0

node-replica-standby

Queue		Session rate		Sessions		Bytes		Denied		Errors		Warnings		Status		Server	
	Cur	Max	Limit	Cur	Max	Limit	Total	Ln	Tot	Ln	Tot	Ln	Tot	Ln	Tot	Ln	Tot
Frontend	0	0	-	0	0	0	2 000	0	0	0	0	0	0	0	0	0	0
postgres_192.168.100.1_6432	0	0	-	0	0	0	100	0	0	0	0	0	0	0	0	0	0
postgres_192.168.100.2_6432	0	0	-	0	0	0	100	0	0	0	0	0	0	0	0	0	0
postgres_192.168.100.3_6432	0	0	-	0	0	0	100	0	0	0	0	0	0	0	0	0	0
Backend	0	0	0	0	0	0	200	0	0	0	0	0	0	0	0	0	0

## 4.7 Montando o repositorio de backups

Instalamos pgbackrest en **pgbackrestRepo**:

```
$ sudo apt install pgbackrest
```

Imos a empregar, por comodidade o usuario postgres para realizar os backups, polo que o usuario debe existir tamén no nodo **pgbackrestRepo** e teremos que crealo.

Ademais, debemos ter en cada un dos nodos configurado o acceso **SSH con clave pública para o usuario postgres**. Isto é necesario para que pgbackrest poida conectarse ao nodo que sexa primario nese momento e realice os backups, se SSH pide contrasinal a conexión falla.

Agora hai que establecer os permisos para o usuario postgres da seguinte maneira (necesario nos nodos de postgres (1,2 e 3 tamén):

```
$ chown postgres:postgres /var/log/pgbackrest -R
```

```
$ chown postgres:postgres /etc/pgbackrest.conf
```

#o seguinte pode ser necesario se falla a creación da stanza:

```
$ chown postgres:postgres /tmp/pgbackrest -R
```

```
$ chmod 750 /tmp/pgbackrest -R
```

Agora imos a xerar a passphrase para encriptar o repositorio:

```
$ openssl rand -base64 48
```

A continuación configuramos pgbackrest.conf en **pgbackrestRepo**, indicando o nome da nosa **stanza**, onde se encontran as instancias de postgres, o directorio do repositorio, a passphrase e o cifrado que emprega e, neste caso tamén establecemos un máximo de backups totais que se manterán:

```
[produccion]
pg1-path=/data/patroni
pg1-port=5432
pg1-host=192.168.100.1
pg1-socket-path=/data/patroni
[produccion]
pg2-path=/data/patroni
pg2-port=5432
pg2-host=192.168.100.2
pg2-socket-path=/data/patroni
[produccion]
pg3-path=/data/patroni
pg3-port=5432
pg3-host=192.168.100.3
pg3-socket-path=/data/patroni

[global]
repo1-path=/var/lib/pgbackrest
repo1-retention-full=3
repo-cipher-pass=TyDo7kHi0y11QQ/g2kSUxcU9xFCiaY2iFjgDX7F6JzyrbUmvCu21JcU898tgJKQ8
repo-cipher-type=aes-256-cbc
start-fast=y
```

Nos nodos de PostgreSQL hai que configurar o arquivo pgbackrest.conf da seguinte maneira, para indicarlles onde se atopa o repositorio:

```
[produccion]
pg1-path=/data/patroni
pg1-socket-path=/data/patroni
pg1-port=5432

[global]
log-level-file=detail
repo1-host=192.168.100.9
repo1-host-user=postgres
```

Agora imos a crear a stanza no nodo repositorio:

**\$ pgbackrest --stanza=produccion --log-level-console=info stanza-create**

```
postgres@pgbackrestRepo:~$ sh crear_stanza
2021-06-06 15:02:41.279 P00 INFO: stanza-create command begin 2.10: --log-level-console=info --pg1-
-host=192.168.100.1 --pg2-host=192.168.100.2 --pg3-host=192.168.100.3 --pg1-path=/data/patroni --pg2-
-path=/data/patroni --pg3-path=/data/patroni --pg1-port=5432 --pg2-port=5432 --pg3-port=5432 --pg1-s
ocket-path=/data/patroni --pg2-socket-path=/data/patroni --pg3-socket-path=/data/patroni --repo1-cip
her-pass=<redacted> --repo1-cipher-type=aes-256-cbc --repo1-path=/var/lib/pgbackrest --stanza=produc
cion
2021-06-06 15:02:44.420 P00 INFO: stanza-create command end: completed successfully (3142ms)
postgres@pgbackrestRepo:~$
```

Comprobamos que se creou correctamente executando o seguinte dende calquera dos nodos de postgres:

**\$ pgbackrest --stanza=produccion --log-level-console=info check**

```
postgres@postgres1:/root$ pgbackrest --stanza=produccion --log-level-console=info check
2021-06-04 20:03:41.598 P00 INFO: check command begin 2.10: --log-level-console=info --log-level-f
ile=detail --pg1-path=/data/patroni --pg1-port=5432 --pg1-socket-path=/data/patroni --repo1-host=192
.168.100.9 --repo1-host-user=postgres --stanza=produccion
2021-06-04 20:03:45.521 P00 INFO: WAL segment 00000004000000000000000007 successfully stored in the
archive at '/var/lib/pgbackrest/archive/produccion/11-1/0000000400000000/000000040000000000000007-d2
ba696cb13e892d6e0a0e97edfbed8d66de91d7.gz'
2021-06-04 20:03:45.529 P00 INFO: check command end: completed successfully (3931ms)
postgres@postgres1:/root$ _
```

*\*Os comandos anteriores deben ser executados como o usuario postgres xa que empregan SSH*

Comprobamos que só arquiva os WAL do primario executando o mesmo comando en calquera dos standby:

```
postgres@postgres3:/root$ pgbackrest --stanza=produccion --log-level-console=info check
2021-06-04 20:05:30.771 P00 INFO: check command begin 2.10: --log-level-console=info --log-level-file=detail --pg1-path=/data/patroni --pg1-port=5432 --pg1-socket-path=/data/patroni --repo1-host=192.168.100.9 --repo1-host-user=postgres --stanza=produccion
2021-06-04 20:05:32.713 P00 INFO: switch wal cannot be performed on the standby, all other checks passed successfully
2021-06-04 20:05:32.720 P00 INFO: check command end: completed successfully (1950ms)
postgres@postgres3:/root$ _
```

Realizamos un backup completo para comprobar:

**\$ pgbackrest --stanza=produccion --log-level-console=info backup**

```
2021-06-05 20:29:02.978 P01 INFO: backup file 192.168.100.1:/data/patroni/base/1/1417 (0B, 100%)
2021-06-05 20:29:02.987 P01 INFO: backup file 192.168.100.1:/data/patroni/base/1/12968 (0B, 100%)
2021-06-05 20:29:02.995 P01 INFO: backup file 192.168.100.1:/data/patroni/base/1/12966 (0B, 100%)
2021-06-05 20:29:02.004 P01 INFO: backup file 192.168.100.1:/data/patroni/base/1/12963 (0B, 100%)
2021-06-05 20:29:03.011 P01 INFO: backup file 192.168.100.1:/data/patroni/base/1/12958 (0B, 100%)
2021-06-05 20:29:03.021 P01 INFO: backup file 192.168.100.1:/data/patroni/base/1/12953 (0B, 100%)
2021-06-05 20:29:03.029 P01 INFO: backup file 192.168.100.1:/data/patroni/base/1/12948 (0B, 100%)
2021-06-05 20:29:03.038 P01 INFO: backup file 192.168.100.1:/data/patroni/base/1/12943 (0B, 100%)
2021-06-05 20:29:03.058 P01 INFO: backup file 192.168.100.1:/data/patroni/base/1/12938 (0B, 100%)
2021-06-05 20:29:03.116 P00 INFO: full backup size = 30.2MB
2021-06-05 20:29:03.118 P00 INFO: execute non-exclusive pg_stop_backup() and wait for all WAL segments to archive
2021-06-05 20:29:03.226 P00 INFO: backup stop archive = 00000004000000000000000009, lsn = 0/9002138
2021-06-05 20:29:04.846 P00 INFO: new backup label = 20210605-202840F
2021-06-05 20:29:04.909 P00 INFO: backup command end: completed successfully (24556ms)
2021-06-05 20:29:04.911 P00 INFO: expire command begin
2021-06-05 20:29:04.931 P00 INFO: full backup total < 3 - using oldest full backup for 11-1 archive retention
2021-06-05 20:29:04.946 P00 INFO: expire command end: completed successfully (35ms)
postgres@pgbackrestRepo:~$ _
```

Programamos unha tarefa de backups no crontab do usuario postgres en **pgbackrestRepo**. Farase un backup completo todos os domingos (*recordar que só se gardarán 3 porque así se especificou na configuración*) e un backup diferencial de luns a sábado.

```
GNU nano 3.2 /tmp/crontab.nShPgu/crontab
#m h dom mon dow command
30 05 * * 0 pgbackrest --type=full stanza=produccion backup
30 05 * 1-6 0 pgbackrest --type=diff stanza=produccion backup
```

Para restaurar un backup hai que deter patroni e executar como usuario postgres:

**\$ pgbackrest --stanza=produccion --delta restore**

```
postgres@postgres1:/root$ pgbackrest --stanza=produccion --log-level-console=info --delta restore
2021-06-06 17:09:44.298 P00 INFO: restore command begin 2.10: --delta --log-level-console=info --log-level-file=detail --pg1-path=/data/patroni --repo1-host=192.168.100.9 --repo1-host-user=postgres --stanza=produccion
2021-06-06 17:09:45.222 P00 INFO: restore backup set 20210606-165343F
2021-06-06 17:09:46.072 P00 INFO: remove invalid files/paths/links from /data/patroni
2021-06-06 17:09:49.438 P01 INFO: restore file /data/patroni/global/pg_control.pgbackrest.tmp (8KB, 88%) checksum dcb85fbc1e52a51a6a91d71b6d2b754e7e053c17
2021-06-06 17:09:51.562 P00 INFO: write /data/patroni/recovery.conf
2021-06-06 17:09:51.569 P00 INFO: restore global/pg_control (performed last to ensure aborted restores cannot be started)
2021-06-06 17:09:51.580 P00 INFO: restore command end: completed successfully (7283ms)
postgres@postgres1:/root$
```

*Se se fai PITR con pgBackRest en Patroni débese facer configurando o método de bootstrap no arquivo de patroni e reiniciando o cluster:*

```
bootstrap:
  dcs:
    ttl: 30
    loop_wait: 10
    retry_timeout: 10
    maximum_lag_on_failover: 1048576
    postgresql:
      use_pg_rewind: true

  method: pgbackrest
  pgbackrest:
    command: /var/lib/postgres/restaurar.sh
    keep_existing_recovery_conf: False
    no_params: False
  recovery_conf:
    recovery_target: immediate
    recovery_target_action: pause
    restore_command: pgbackrest --stanza=produccion archive-get %f %p_
```

```
GNU nano 3.2 /var/lib/postgresql/restaurar.sh Modificado
#!/bin/sh
mkdir -p /data/patroni
pgbackrest --stanza=produccion --log-level-console=info --delta \
--type=time "--target=2021-06-06 16:04:30.320599+01" --target-action=promote restore \
/
```

## 4.8 Estimación e planificación da posta en marcha

A posta en marcha fariase sistema a sistema e no mesmo orde no que aquí se indicou.

Dependendo da infraestrutura anterior da empresa pode variar o tempo en gran medida, no caso de partir dun sistema de bases de datos existente, que é o máis probable, contamos co método de bootstrap de Patroni que nos permitirá arrancar o cluster directamente a partir dos backups que haxa, sen importar o método co que se fixeran, polo que se aceleraría o proceso.

Se partimos de cero, a estimación é de **2 a 3 días completos de traballo**, o primeiro para a configuración e o arranque da infraestrutura e o segundo para a comprobación do funcionamento e a fortificación dos sistemas.

## 5 Soporte e mantemento

---

### 5.1 Distribución dos sistemas

---

Manter todos os sistemas ou toda a información nun mesmo CPD pode resultar nunha catástrofe absoluta, poderíamos dicir que o nivel de seguridade da información é proporcional á distancia entre os Centros de Proceso de Datos.

Outro risco de ter todos os sistemas centralizados é que se a zona na que se atopa o CPD perde a comunicación quedaríamos sen acceso aos datos, aínda que non os perdamos, supón unha interrupción do servizo.

É por iso que os servidores se distribuirán xeográficamente coa distancia suficiente para que a caída dun CPD non afecte a todos os sistemas.

### 5.2 Copias de seguridade

---

Aínda que no último apartado do noso modelo técnico xa se establece unha política de copias de seguridade, é preciso elaborar unha planificación máis completa:

- Trasladar os backups do nodo que funciona como repositorio a outro servidor/**segundo repositorio** para ter unha copia e se falla o repositorio principal seguir podendo acceder a eles.
- **Programar copias de seguridade que se manteñan offline**, para que no caso de que a seguridade da organización se vexa comprometida se conserve a integridade da información.
- **Restauración e comprobación de backups diaria e automática**. Isto podería realizarse nun contorno de preproducción, facilitaría aos desenvolvedores o seu traballo xa que ter unha copia actualizada da base de datos para traballar nas novas versións das aplicacións é esencial e proporcionaríanos un teste consistente.
- **Se non se comproban os backups, non hai backups.**
- **Exemplo planificación de backups:**
  - Gardar un backup completo cada primeiro de mes.
  - Gardar un backup completo ao final de cada ano.
  - Realizar backups completos unha vez por semana, almacenando os últimos 3.
  - Realizar un backup diferencial o resto de días da semana.

No caso de fallo total do sistema e de que a información se vise corrompida contamos co sistema **PITR** para volver a un punto no que sabemos con certeza que a información non se corrompeu e a posibilidade de configurar o **bootstrap** de

Patroni para iniciar o cluster enteiro a partir da copia de seguridade que sexa convinte.

Coa **retransmisión de WAL** temos ademais, unha copia sincronizada constantemente entre os tres servidores de bases de datos polo que se falla un sistema, o que tome o relevo estará ao día.

**PGBackRest** seguirá podendo realizar backups aínda que falle o nodo primario, xa que é capaz de detectar cal é o novo líder e facer as copias de seguridade dende el.

COPIAS LÓXICAS	COPIAS FÍSICAS
Máis sinxelas de configurar	Mellor adaptadas para BBDD de gran tamaño
Facilitan a migración de Postgres a novas versións	Permiten a Point In Time Recovery
Permiten facer backup dunha soa BBDD	

As copias que se están a facer son físicas, xa que isto está orientado a BBDD de gran tamaño, pero sería recomendable realizar tamén unha copia lóxica da base de datos frecuentemente, que pode ser útil nunha emerxencia.

### 5.3 Plan de continxencias

Nesta infraestrutura xa contamos con:

- Failover/switchover automático.
- Copias de seguridade automáticas e replicación constante.
- Alta dispoñibilidade.
- Balanceo de carga.
- Un sistema distribuído.

Polo que, están cubertos a gran maioría de escenarios e o sistema reaccionará de maneira automática para recuperarse cando detecte un problema.

### 5.4 Supervisión e monitorización

Contamos con diversas ferramentas para a revisión dos sistemas:

- **etcdctl**: a ferramenta de consola de Etcd que nos permite ver o estado do cluster e os membros, así como promocionar ou engadir novos e que é accesible dende múltiples puntos.

- **patronictl**: unha ferramenta similar á de etcd, que tamén nos permite revisar a saúde do cluster, reintroducir membros, reiniciar o cluster...e é accesible dende calquera nodo.
- **O panel de administración de haproxy**, que podemos acceder dende calquera sistema final a través dun navegador e nos ofrece estadísticas varias e o estado e rol de cada servidor de bases de datos nese momento. Dende este panel tamén podemos ver as transicións entre os servidores paso a paso. Este panel ademais está altamente dispoñible, xa que nos podemos conectar a través da IP virtual e se un balanceador de carga fallase ofreceríanos o servizo o outro.

## 5.5 Actualizacións

---

Non se recomenda facer unha actualización xeral (por ex. *apt update*) xa que pode levar moito tempo e introducir un novo kernel obrigándonos a reiniciar. Actualizaremos só o esencial.

Para actualizar o cluster hai que seguir este procedemento:

- Pausar PGBouncer para suspender conexións futuras de clientes a ese nodo.
- Nos nodos standby de patroni:
  - Esperar a que as consultas actuais se completen.
  - Crear unha copia de seguridade do cluster por se fose necesario facer **rollback** (retornar a un punto anterior).
  - Desactivar o failover automático de Patroni para que entre en modo mantemento:

```
$ patronictl -c patroni.yml pause
```

- Nos nodos standby deter o servidor de Postgres:

```
$ pg_ctl -D /var/lib/pgsql/data stop -m fast
```

- Executamos a actualización de **PostgreSQL 11**
- Actualizamos individualmente pgbouncer, haproxy, pgbackrest e keepalived
- Comprobamos o correcto funcionamento.
- Despausamos o failover:

```
$ patronictl -c patroni.yml pause
```

- No nodo primario:
  - Realizamos un switchover manual a calquera dos nodos standby actualizados:

```
$ patronictl -c patroni.yml switchover
```



- Seguimos os mesmos pasos ca nos standby
- Devolvémolle o rol de primario facendo de novo o switchover.
- Ao actualizar PostgreSQL a unha nova versión hai que executar o comando **stanza-upgrade**.
- Para actualizar Etcd procedemos membro a membro, comezando por facer un snapshot da información actual en caso de que haxa que volver atrás:

```
$ etcdctl snapshot save recuperar.db
```

```
$ etcdctl -write-out=table snapshot status  
recuperar.db
```

*Etcd permite que membros dun mesmo cluster teñan distintas versións sempre e cando sexan menores, é dicir, non podemos actualizar un nodo da versión 2.3 á 3.0. Debemos ir unha por unha.*

- Despregamos a actualización nun nodo e revisamos que siga funcionando correctamente co seguinte comando:

```
$ etcdctl endpoint health  
--endpoints=192.168.100.4:2379,  
192.168.100.7:2379,192.168.100.8:2379
```

- Se non hai ningún problema continuamos actualizando os demais membros do cluster.

## 6 Estudo económico

### 6.1 DaaS: Database as a Service

Unha **base de datos administrada** é un servizo de computación na nube que nos permite pagar a un proveedor polo acceso a unha base de datos. Este proveedor é o encargado de darlle soporte, actualizala, escalala e facer copias de seguridade.

Podemos escoller o motor da base de datos, o tamaño do cluster...

Este tipo de servizo **simplifica moito o traballo pero limita a flexibilidade**, xa que o enfoque adoita ser ofrecer unha base de datos máis rudimentaria e que se acomode a un gran volume de clientes.

Isto pode ser un problema cando precisamos unhas características ou extensións específicas e dependerá do noso software en gran medida.

Outra desvantaxe das bases de datos administradas é o **prezo**, xa que estamos pagando pola súa xestión. **O custo pode aumentar moito** se temos un tráfico moi elevado ou se traballamos cun gran volume de datos.

Algo a ter moi en conta tamén é o SLA (**Services Level Agreement**), é dicir, o acordo ao que chegas co proveedor ao aceptar o seu servizo, o cal pode afectar no fluxo de traballo da organización e ademais faite moi dependente dun terceiro.

Por último, nunha base de datos administrada **non controlamos onde se almacenan os datos nin como se protexen**, o cal pode ser incompatible coas políticas da organización.

Unha base de datos administrada é sen duda unha opción a ter en conta xa que facilita enormemente o mantemento dun pilar fundamental, pero as facilidades veñen acompañadas dun aumento de prezo e funcionalidades limitadas.

## 6.2 Amazon Aurora

A arquitectura de Amazon Aurora proporciona características de alta dispoñibilidade e mantén a información segura almacenándoa nas chamadas Zonas de Dispoñibilidade, accesibles aínda que as instancias de base de datos estean caídas.

Aurora permite crear ata 15 réplicas de só lectura nun cluster cun só servidor primario de lectura e escritura.

O servizo de Amazon conta tamén con failover automático mediante a promoción dunha das instancias de só lectura ou a creación dunha nova instancia primaria. Estas instancias poden organizarse en orde de prioridade.

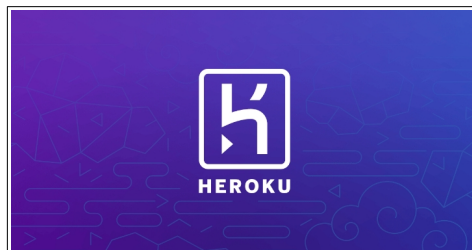
Aurora realiza a recuperación ante erros de maneira asíncrona empregando fíos paralelos para acelerar a recuperación da base de datos.



## 6.3 Heroku

Heroku foi o primeiro proveedor en impulsar o emprego de PostgreSQL no lugar de MySQL para o desenvolvemento de aplicacións, é facilmente escalable tanto vertical como horizontalmente.

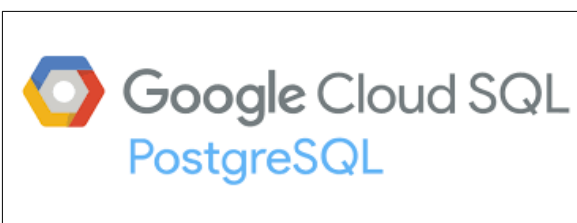
Esta **PaaS** (Plataforma como servizo) é máis sinxela que AWS, sendo tamén máis apropiada para pequenas e medianas empresas mentras que Aurora está enfocado a negocios de maior tamaño.



## 6.4 Cloud SQL

Outra solución completamente xestionada e escalable.

As instancias de Cloud SQL configuradas para ofrecer alta dispoñibilidade reciben o nome de *instancias rexionais* e ubícanse nunha zona primaria e unaha secundaria dentro da rexión configurada.



Dentro dunha instancia rexional, a configuración componse dunha instancia primaria e unha instancia en espera.

Cloud SQL realiza replicación síncrona cos discos persistentes de cada zona, replicando todas as operacións de escritura realizadas na instancia principal nos discos de ambas zonas antes de que se confirme unha transacción.

Se hai un fallo nunha instancia ou zona, o disco persistente trasládase á instancia en espera, que se converte nunha instancia principal, á que se redirixen os usuarios. Isto chámase *conmutación por erro*.

As réplicas de lectura non poden ter alta dispoñibilidade. Durante unha interrupción nunha zona detense o tráfico de lectura cara as réplicas.

## 6.5 Azure SQL

---

Os servizos de tier crítico de Azure integran os recursos computacionais e o almacenamento (SSD conectado localmente) nun único nodo. A alta dispoñibilidade ofrécena replicando todo en nodos adicionais creando un cluster de tres a catro nodos.

Os datos replicanse constantemente aos nodos de respaldo aos que Azure Service Fabric fará o failover automático no caso de fallo. As conexións dos clientes son redirixidas automaticamente.



Azure Premium permite redixir conexións de só lectura a **unha** das réplicas.

## 6.6 Custos da infraestrutura e os servizos contratados

---

### 6.6.1 Hardware

Contrataranse varios servidores dedicados en OVH co sistema operativo Debian 10.

Á hora de montar a infraestrutura nun entorno real e para aproveitar ben os recursos, integrárase o repositorio de **pgbackrest** dentro do cluster de **etcd**. Para etcd requirimos almacenamento SSD pero para os backups cun RAID de varios discos duros será suficiente.

O mínimo recomendado para etcd son 16GB de RAM e un procesador de catro núcleos. Hai que ter en conta que este cluster pode administrar unha enorme cantidade de instancias de Patroni futuras.

Tres servidores **Advance-3** para etcd e pgbackrest, a un coste de 109,24€ por servidor, por un total de 327,72€ mensuais.

*Entre as opcións de ancho de banda quizais sería recomendable coller a que nos da máis garantías.*

465,79 € + IVA/mes  
Instalación gratuita (- 99,99 €)

Contratar

Alemania (Fráncfort - FRA) 10 d

Cambiar datacenter

Procesador :

AMD Epyc 7371 - 16c/ 32t - 3.1GHz/ 3.8GHz

Memoria :

128 GB DDR4 ECC 2400 MHz +0,00 €

256 GB DDR4 ECC 2400 MHz +60,00€ +57,00 €

512 GB DDR4 ECC 2400 MHz +180,00€ +171,00 €

Almacenamiento :

3 x 3,84 TB SSD NVMe Soft RAID +136,80 €

Ancho de banda público :

2Gb/s, ilimitado y garantizado +120,00 €

Ancho de banda privado :

2Gb/s, ilimitado +0,00 €

109,24 € + IVA/mes  
Instalación gratuita (- 79,99 €)

Contratar

Francia (Roubaix - RBX) 30 d

Cambiar datacenter

Procesador :

Intel Xeon-D 2141I - 8c/ 16t - 2.2GHz/ 3GHz

Memoria :

32 GB DDR4 ECC 2133 MHz +0,00 €

64 GB DDR4 ECC 2133 MHz +15,00€ +14,25 €

128 GB DDR4 ECC 2133 MHz +45,00€ +42,75 €

Almacenamiento :

2 x 500 GB SSD NVMe Soft RAID + 2 x 4 TB HDD SATA Soft RAID +28,50 €

Ancho de banda público :

1Gb/s, ilimitado +0,00 €

1Gb/s, ilimitado +0,00 €

1Gb/s, ilimitado y garantizado +70,00 €

Ancho de banda privado :

100Mb/s, ilimitado y garantizado +0,00 €

Tres servidores **Infra-3** de base de datos para soportar múltiples instancias de Patroni e Postgres. Que ademáis integrarán PGBouncer. Precisan un ancho de banda considerable e un hardware bastante máis potente ca os nodos de etcd. O almacenamento será NVMe para que o acceso á información sexa o máis rápido posible.

Tres servidores a 465,79€ mensuais, por un total de 1397,37€.

Para os balanceadores de carga precisamos unha boa cantidade de memoria RAM e ancho de banda garantizado. Optamos por dous servidores **Advance-2**. O seu custo é de 141,24€ mensuais cada un, por un total de 282,48€.

141,24 € + IVA/mes  
Instalación gratuita (- 49,99 €)

Contratar

Polonia (Varsovia - WAW) 10 d

Cambiar datacenter

Procesador :

Intel Xeon-E 2136 - 6c/ 12t - 3.3GHz/ 4.5GHz

Memoria :

☒ 32 GB DDR4 ECC 2666 MHz +0,00 €  
☐ 64 GB DDR4 ECC 2666 MHz +15,00 € +14,25 €  
☐ 128 GB DDR4 ECC 2666 MHz +45,00 € +42,75 €

Almacenamiento :

2 x 500 GB SSD NVMe Soft RAID +0,00 €

Ancho de banda público :

1Gb/s, ilimitado y garantizado +70,00 €

Ancho de banda privado :

100Mb/s, ilimitado y garantizado +0,00 €

Distribuciones

Ver distribuciones disponibles

O custo total de manter os servizos rondaría os 2000 euros mensuais.

### 6.6.2 Persoal

Esta infraestrutura ten un nivel de automatización que permitiría que fose controlada por un só administrador de bases de datos, posto que conta cun salario medio duns 37.000€ anuais.

Dependendo do tamaño do cluster, pode ser suficiente cun enxeñeiro de DevOps, que ronda os 30.000€ anuais.

## 7 Posibles melloras

Plantéxanse as seguintes melloras e recomendacións de cara ao futuro e á implantación nun contorno de produción:

- Ao configurar a autenticación de usuarios nas diversas ferramentas, especialmente Patroni e PGBouncer empregáronse contrasinais en claro por comodidade, nun entorno de produción sería convinte empregar encriptación.
- Etcld soporta TLS automático para a autenticación dos membros do cluster. Sería convinte xerar certificados dixitais e habilitalo.

- Actualmente PGBouncer non conta con failover automático nin monitorización específica. Aínda que é moi improbable que falle únicamente PGBouncer e o resto do nodo siga en pé, se o noso obxectivo é acadar a alta dispoñibilidade sería convinte ter un mecanismo para respaldalo.

Unha solución podería ser mediante un script configurado no crontab para que se revise o servizo constantemente e se intente recuperar se falla:

```
GNU nano 3.2 /root/scripts/pgbouncer_failover.sh Modificado
#!/bin/bash
STATUSPGB=$(systemctl is-active pgbouncer.service)
LOG="/var/log/pgbouncer_failover.log"
case $STATUSPGB in
    inactive)
        #primeiro intentará recuperar o servizo pgbouncer
        systemctl restart pgbouncer && sleep 5
        STATUSPGB=$(systemctl is-active pgbouncer.service)
        #se falla detén patroni para que promocióne outro nodo e non se empregue este
        if [ "${STATUSPGB}" = "inactive" ]
        then
            systemctl stop patroni
            echo "$(date) ----- Fallo ao reiniciar PGbouncer en $HOSTNAME" >> $LOG
        else
            echo "$(date) ----- Recuperouse PGbouncer en $HOSTNAME" >> $LOG
        fi
    active)
        exit 0
esac
```

## 8 Dificultades atopadas

Durante o desenvolvemento do proxecto solventáronse os seguintes problemas:

- Ao intentar levantar o HAProxy de respaldo unha vez configurado Keepalived o servizo entraba nun estado de erro. Isto debíase a que a IP virtual xa estaba asociada na interface do nodo HAProxy primario, polo que ao iniciar o respaldo, este detectaba que tiña asociada unha IP que non era local ao equipo.  
Hai que habilitar a asociación de IPs non locais en Linux para solucionalo (*ver apartado de Implementación*).
- Se arrancas de novo o cluster de etcd sobre un cluster de patroni totalmente distinto ao que había inicialmente a información almacenada polo DCS non coincide e pode xerarse un escenario split-brain no que cada nodo intenta formar o seu propio cluster e seguir unha liña temporal diferente.

- É recomendable que o repositorio de pgBackRest non estea situado na mesma máquina que funciona como servidor primario da base de datos. De facelo así hai que empregar unha configuración específica (*indicando a pgBackRest que a BBDD é local á máquina*), eu optei por configurar un nodo repositorio.
- Aínda que indicase a Patroni na configuración que habilitase o arquivado de WAL, este non o habilitaba polo que non se podía empregar pgBackRest correctamente.

A solución foi reinicializar o cluster de Patroni:

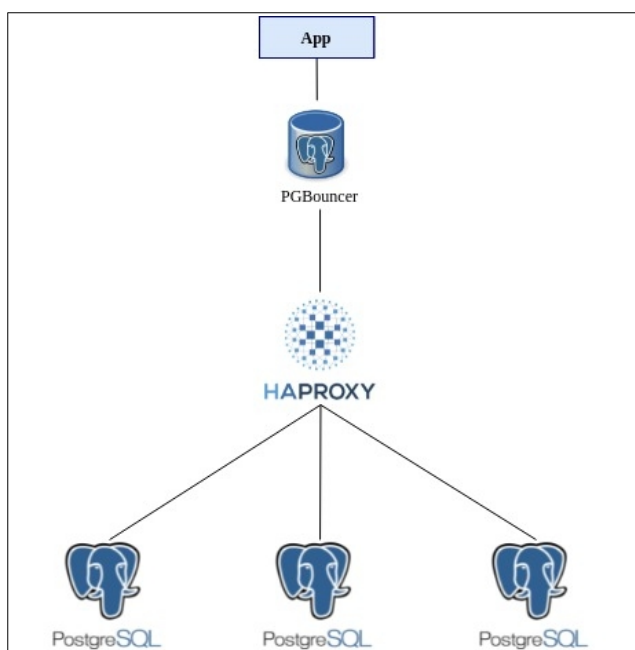
```
$ patronictl -c /etc/patroni.yml reinit
```

- Dilema coa localización de PGBouncer:

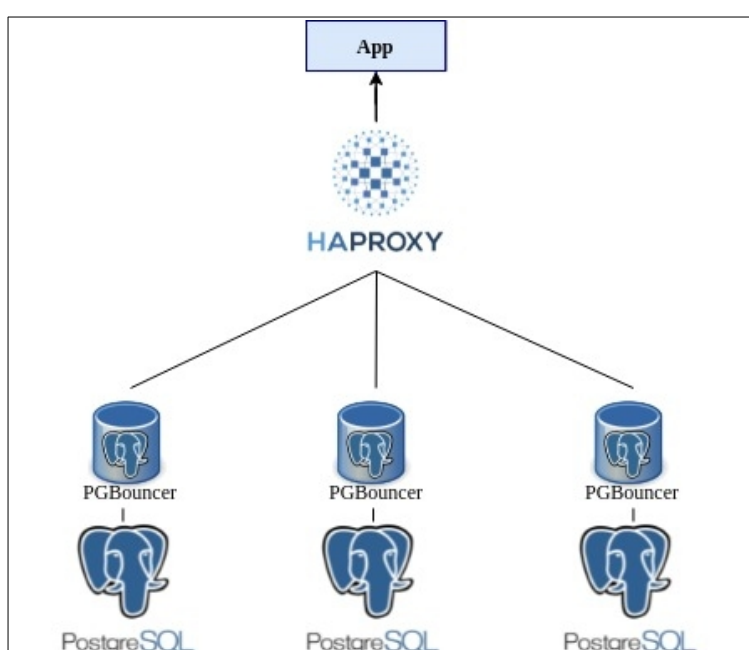
á hora de posicionar o pool de conexións non tiña moi claro onde debía poñelo e atopei que hai quen conecta as aplicacións a unha única instancia de PGBouncer que conecta con HAProxy para que faga balanceo de carga das bases de datos.

A alternativa é o que se aplicou no proxecto: ter unha instancia de PGBouncer en cada servidor de base de datos e que a aplicación se conectase a través do HAProxy.

1º



2º





Unha app tarda moito menos en conectase á base de datos a través de PGBouncer debido a que os mecanismos de PostgreSQL para verificar credenciais cada vez que se solicita unha conexión son moi lentos.

Ademais, HAProxy redirixe a conexión entre os servidores, o cal resulta nun cambio de MAC na conexión coa BBDD. Se PGBouncer está situado diante de HAProxy as conexións no pool son invalidadas reiteradamente por ese cambio de MAC.

É por isto que se decidiu optar polo segundo achegamento.

## 9 Conclusións

Este proxecto ensinome moito sobre PostgreSQL e a importancia que ten para o mantemento dun sistema de gran tamaño a automatización de procedementos e a verificación de que estes se realizan correctamente.

A tranquilidade que ofrece saber que tes un sistema ben mantido e configurado, capaz de restablecerse só e que nun futuro poidas escalar con facilidade non ten prezo, ou mellor dito si o ten, pero merece a pena se se trata dunha gran infraestrutura.

Non podo ignorar a frustración que sentín algúns días nos que tras varias horas intentando facer que algunha das partes funcionase, esta seguía fallando; non polo feito de ter que pasar máis tempo investigando, se non porque o meu tempo era bastante limitado e ter que dedicarlle unhas horas a un problema e non acadar unha resolución significaba ademais non poder avanzar noutros apartados importantes.

Poñerme a traballar no proxecto ao principio custaba; non saber por onde comezar ou que ferramentas empregar era abafante, pero unha vez estivo todo definido podía pasarme horas e horas xogando co software e era moi gratificante ver como as distintas pezas comezaban a encaixar e todo avanzaba.

É por isto polo que a experiencia, que comezou cargada de frustración rematou sendo moi instructiva e positiva.

## 10 Bibliografía e referencias

Todos os esquemas e diagramas deste proxecto foron realizados en [draw.io](https://draw.io), salvo os das páxinas 9 e 14.

As táboas e as gráficas fixéronse en [canva](https://canva.com).

Consultouse o github de distintas ferramentas para resolver problemas e documentarse:

- [Patroni](#)

- [Manual de configuración](#)
- [Etcd](#)
- [pgBackRest](#)
  - [Documentación por EDB](#)
- [HAProxy](#)
  - [Manual de configuración](#)
- [Keepalived](#)
  - [Guía de configuración por RedHat](#)
- [PGBouncer](#)
  - [Modos de pooling](#)

Documentación de PostgreSQL:

- [Streaming replication](#)
- [Hot standby](#)
- [Continuous archiving and PITR](#)

Artigos:

- [Escalando PostgreSQL](#)
- [Frameworks de alta disponibilidad](#)
- [Clusters de Etcd con Patroni](#)
- [Protocolo VRRP](#)
- [Algoritmo de consenso RAFT](#)
- [Split-brain](#)
- [Alta disponibilidad con Patroni](#)
- [Alta disponibilidad con Repmgr](#)
- [Failover automático de PostgreSQL](#)
- [Necesidade dun pool de conexións](#)

Libros:

- [Pro Linux High Availability Clustering](#) - Sander Van Gut, 2014

Guías:

- [Activar asociación de IPs non locais](#)
- [Patroni: replicas e bootstrap](#)

## 11 Anexos

### 11.1 Bootstrap dunha réplica a partires dun backup

No caso de que queiramos ampliar o tamaño do cluster podemos configurar Patroni para arrancar réplicas directamente a partires dun backup. Grazas á flexibilidade da configuración de **Patroni** e á ferramenta **pgBackRest**.

Para facelo, débese modificar o ficheiro de configuración de Patroni no nodo que se queira introducir ao cluster:

```
postgresql:
  create_replica_methods:
    - pgbackrest
  pgbackrest:
    command: pgbackrest --pg1-path=/data/patroni --stanza=produccion --delta restore
    keep_data: True
    no_params: True
  listen: 192.168.100.3:5432
  connect_address: 192.168.100.3:5432
  data_dir: /data/patroni
  pgpass: /tmp/patroni
  authentication:
    replication:
      username: replicator
      password: abc123.
    superuser:
      username: postgres
      password: superbadm
  parameters:
    unix_socket_directories: '.'
    archive_mode: "on"
    archive_command: 'pgbackrest --stanza=produccion archive-push %p'
```

Agora simplemente queda iniciar o servizo e revisar os logs e o estado do mesmo. Rediriximos a saída do comando de arrancar o servizo e revisamos a información:

```
GNU nano 3.2      patroni_restore.log

• patroni.service - PostgreSQL HA Cluster
  Loaded: loaded (/etc/systemd/system/patroni.service; disabled; vendor preset: enabled)
  Active: active (running) since Wed 2021-06-16 22:16:12 CEST; 6s ago
  Main PID: 698 (patroni)
  Tasks: 12 (limit: 545)
  Memory: 119.0M
  CGroup: /system.slice/patroni.service
          └─698 /usr/bin/python /usr/local/bin/patroni /etc/patroni.yml
             707 pgbackrest --pg1-path=/data/patroni --stanza=produccion --delta restore
             708 sh -c ssh -o LogLevel=error -o Compression=no -o PasswordAuthentication=no postgres
             709 ssh -o LogLevel=error -o Compression=no -o PasswordAuthentication=no postgres@192.168.100.4
             714 pgbackrest --command=restore --host-id=1 --log-level-file=off --pg1-path=/data/pat
             715 sh -c ssh -o LogLevel=error -o Compression=no -o PasswordAuthentication=no postgres
             716 ssh -o LogLevel=error -o Compression=no -o PasswordAuthentication=no postgres@192.168.100.4

jun 16 22:16:12 postgres3 systemd[1]: Started PostgreSQL HA Cluster.
jun 16 22:16:13 postgres3 patroni[698]: 2021-06-16 22:16:13,206 INFO: Selected new etcd server
http://192.168.100.4:2379
jun 16 22:16:13 postgres3 patroni[698]: 2021-06-16 22:16:13,212
INFO: No PostgreSQL configuration items changed, nothing to reload.
jun 16 22:16:13 postgres3 patroni[698]: 2021-06-16 22:16:13,219
INFO: Lock owner: postgres1; I am postgres3
jun 16 22:16:13 postgres3 patroni[698]: 2021-06-16 22:16:13,224
INFO: trying to bootstrap from leader 'postgres1'
jun 16 22:16:14 postgres3 patroni[698]:
WARN: --delta or --force specified but unable to find 'PG_VERSION' or 'backup.manifest'
in '/data/patroni' to confirm that this is a valid $PGDATA directory. --delta and --force
have been disabled and if any files exist in the destination directories the restore
will be aborted.
```

Vemos que efectivamente se está executando a restauración e nos da un aviso de que o directorio de Patroni debe estar baleiro, no noso caso non é preocupante xa que estamos a introducir unha nova réplica. Se fose un nodo que estivese sendo reintroducido sinxelamente borramos /data/patroni e asegurámonos de que únicamente o usuario postgres teña acceso rwx (700).

Esperamos un pouco e volvemos a mirar os logs:

```
root@postgres3:~# systemctl status patroni.service
● patroni.service - PostgreSQL HA Cluster
   Loaded: loaded (/etc/systemd/system/patroni.service; disabled; vendor preset: enabled)
   Active: active (running) since Wed 2021-06-16 22:16:12 CEST; 19min ago
     Main PID: 698 (patroni)
        Tasks: 12 (limit: 545)
       Memory: 208.7M
      CGroup: /system.slice/patroni.service
              └─ 698 /usr/bin/python /usr/local/bin/patroni /etc/patroni.yml
                 1119 postgres -D /data/patroni --config-file=/data/patroni/postgresql.conf --listen_ad
                 1121 postgres: postgres: startup   recovering 000000020000000000000000
                 1122 postgres: postgres: walreceiver streaming 0/9000140
                 1123 postgres: postgres: checkpointer
                 1124 postgres: postgres: background writer
                 1125 postgres: postgres: stats collector
                 1130 postgres: postgres: postgres postgres 192.168.100.3(57870) idle

jun 16 22:35:25 postgres3 patroni[698]: 2021-06-16 22:35:25,238 INFO: no action. i am a secondary a
jun 16 22:35:35 postgres3 patroni[698]: 2021-06-16 22:35:35,227 INFO: Lock owner: postgres1; I am po
jun 16 22:35:35 postgres3 patroni[698]: 2021-06-16 22:35:35,228 INFO: does not have lock
jun 16 22:35:35 postgres3 patroni[698]: 2021-06-16 22:35:35,233 INFO: no action. i am a secondary a
jun 16 22:35:45 postgres3 patroni[698]: 2021-06-16 22:35:45,233 INFO: Lock owner: postgres1; I am po
jun 16 22:35:45 postgres3 patroni[698]: 2021-06-16 22:35:45,234 INFO: does not have lock
jun 16 22:35:45 postgres3 patroni[698]: 2021-06-16 22:35:45,239 INFO: no action. i am a secondary a
jun 16 22:35:55 postgres3 patroni[698]: 2021-06-16 22:35:55,229 INFO: Lock owner: postgres1; I am po
```

Finalmente podemos revisar o estado do cluster, da mesma maneira que se amosou no apartado 4:

```
root@postgres3:~# patronictl -c /etc/patroni.yml list
+ Cluster: postgres (6970693010785563236) -----+
| Member   | Host           | Role    | State  | TL | Lag in MB |
+-----+-----+-----+-----+-----+-----+
| postgres1 | 192.168.100.1 | Leader  | running | 2 |          |
| postgres2 | 192.168.100.2 | Replica | running | 2 |         0.0 |
| postgres3 | 192.168.100.3 | Replica | running | 2 |         0.0 |
+-----+-----+-----+-----+-----+-----+

```

Podemos confirmar polo tanto que todo se realizou correctamente.