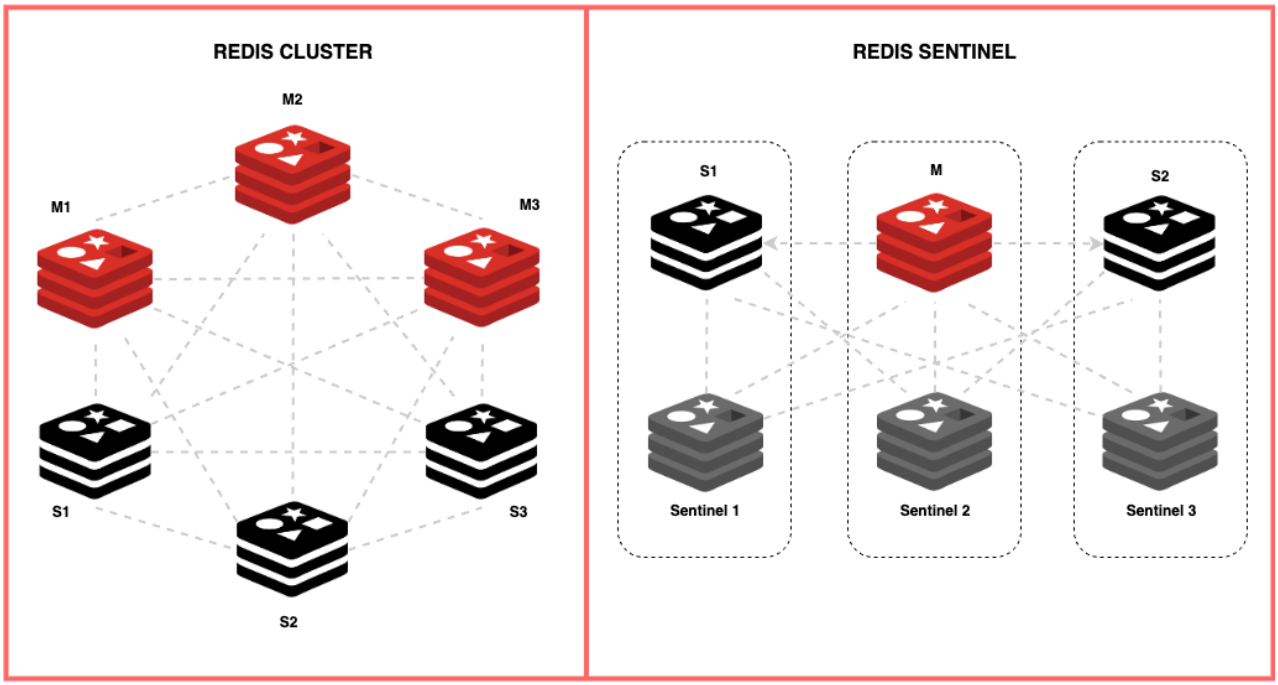
Understanding Redis High Availability: Cluster vs. Sentinel

medium.com/@khandelwal.praful/understanding-redis-high-availability-cluster-vs-sentinel-420ecaac3236

Praful Khandelwal November 26, 2024

Redis is a leading in-memory database that can be used as a key-value store, cache or message broker. As applications grow in complexity, ensuring high availability and fault tolerance becomes crucial. To address these challenges, Redis provides two primary solutions: **Redis Cluster** and **Redis Sentinel**. While both are designed to ensure resilience, they cater to different use cases.



In this blog, we'll explore:

1. What Redis Cluster and Redis Sentinel are
2. Key differences between the two
3. Step-by-step approach to set up each of those modes
4. Pitfalls/Challenges

Redis Cluster

Redis Cluster provides high availability and horizontal scaling. It divides the data into shards, distributed across multiple nodes. Each shard has a primary node and replicas for fault tolerance. This allows Redis Cluster to handle larger datasets and higher write throughput.

Redis Sentinel

Redis Sentinel is a high availability tool that monitors Redis instances and provides automatic failover for a primary-replica architecture. When the primary node becomes unavailable, Sentinel promotes one of the replicas to primary and updates the remaining replicas to sync with the new primary.

Comparative Analysis

Here is a quick comparison of two modes:

Feature	Redis Cluster	Redis Sentinel
Purpose	High availability with built-in horizontal scaling via sharding	High availability and failover management for a single dataset
Primary Node Count	Multiple primaries (with sharded datasets)	One primary per Sentinel group
Horizontal Scaling	Supported (automatic data sharding across nodes)	Limited (read replicas only; no native sharding)
Write Scalability	Distributed across multiple primaries	Single primary limits write capacity
Read Scalability	Supported via replicas across shards	Supported via replicas
Fault Tolerance	Automatic failover within shards	Automatic failover to replicas
Data Partitioning	Automatic (via hash slots)	Manual (handled by the application, if needed)
Operational Complexity	More complex due to sharding and multi-node coordination	Easier to set up and maintain
Quorum for Failover	Minimum majority of nodes needed for failover decisions	Sentinel nodes vote for failover
Use Case	Applications requiring scalability for large datasets	Applications needing high availability with simpler setups

Setup Procedure

We will use Azure Virtual Machines here with Ubuntu OS. Same steps can be followed to set up Redis on Amazon EC2, On-prem Ubuntu devices or any other similar environment.

We will consider minimum 3-node set up for high availability and fault tolerance though you may need more nodes in production depending on your use case. Specially, for Redis Cluster, you may want to go with 6 nodes (3 master + 3 replica). However, we'll try the setup with just 3 nodes (each having 1 master and 1 replica). There are some limitations with this configuration, which I'll discuss later.

It is assumed that 3 Azure VMs are already set up with:

- Public IPs assigned as we would access them from local machine
- Required inbound ports (22-SSH and 6379-Redis) are open
- SSH key downloaded and permissions updated

Also make sure that redis-cli is installed on your local machine to access and operate on redis instances running on Azure VMs. On Mac, it can simply be installed with below command:

```
$ brew install redis-cli
```

SETTING UP AND TESTING REDIS CLUSTER

Let's consider a set of 3 VMs (VM1, VM2, VM3)

SSH into VM1

```
$ ssh -i <key> azureuser@<VM1 IP Address>
```

Update packages

```
$ sudo apt update
```

Install redis server

```
$ sudo apt install redis-server -y
```

Open redis configuration file (created by default during redis server installation)

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

Enable/Update the following options in above configuration file (

```
bind 0.0.0.0port 6379protected-mode nocluster-enabled yescluster-config-file
nodes-6379.confcluster-node-timeout 15000appendonly yes
```

Now, we will use above config file as a reference and create two more config files by copying it — one for master and one for slave.

```
$ sudo  /etc/redis/redis.conf /etc/redis/redis-6380.conf$ sudo
/etc/redis/redis.conf /etc/redis/redis-6381.conf
```

Open first configuration file

```
$ sudo nano /etc/redis/redis-6380.conf
```

Enable/Update the following options in replica configuration file.

```
bind 0.0.0.0port 6380protected-mode nocluster-enabled yescluster-config-file
nodes-6380.confcluster-node-timeout 15000appendonly yes
```

Open second configuration file

```
$ sudo nano /etc/redis/redis-6381.conf
```

Enable/Update the following options in replica configuration file.

```
bind 0.0.0.0port 6381protected-mode nocluster-enabled yescluster-config-file
nodes-6381.confcluster-node-timeout 15000appendonly yes
```

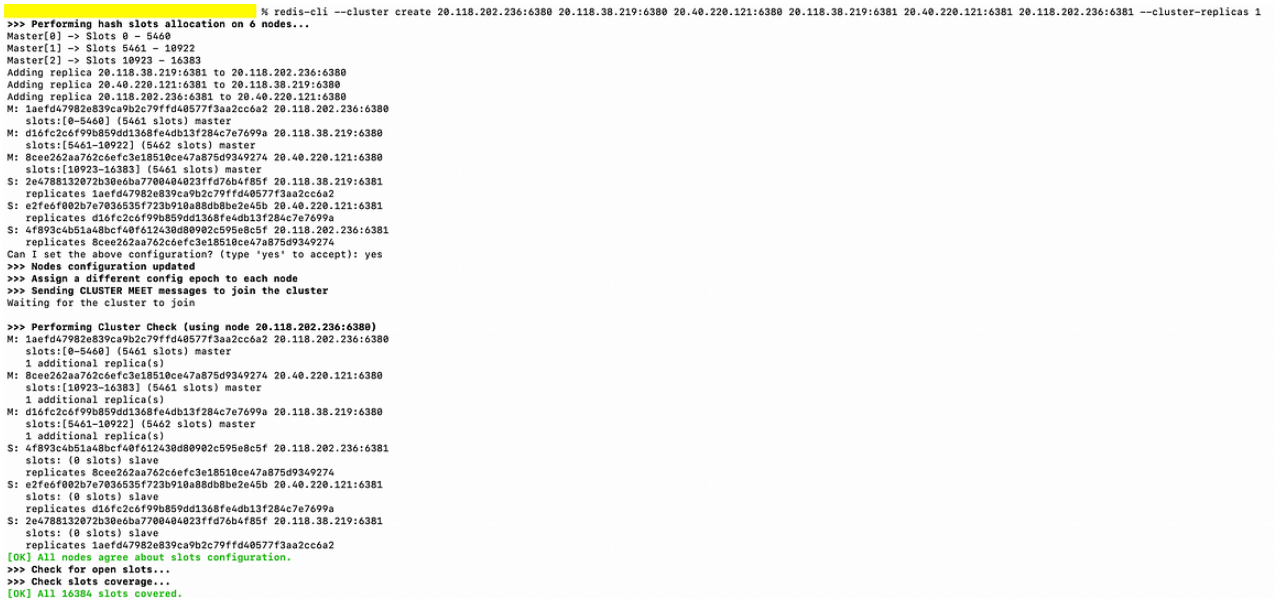
Start the two redis instances

```
$ sudo redis-server /etc/redis/redis-6380.conf$ sudo redis-server
/etc/redis/redis-6381.conf
```

- Repeat exact same steps on other 2 VMs (VM2 and VM3)
- Now our 3 VMs are ready with 2 redis instances running on each. We can add them to the cluster. For this we will use installed on our local machine. By default, Redis Cluster doesn’t automatically ensure that replicas are placed on different nodes from their masters. We can handle that manually by carefully ordering the IP addresses and ports during cluster creation as follows. We’ll use the redis instances running on 6380 ports as masters and the ones running on 6381 ports as slaves ()

```
$ redis-cli --cluster create <VM1 Public IP Address>:6380 <VM2 Public IP
Address>:6380 <VM3 Public IP Address>:6380 <VM2 Public IP Address>:6381 <VM3
Public IP Address>:6381 <VM1 Public IP Address>:6381 --cluster-replicas 1
```

If all goes well, it should display the output as follows (showing the masters and slaves along with slots):



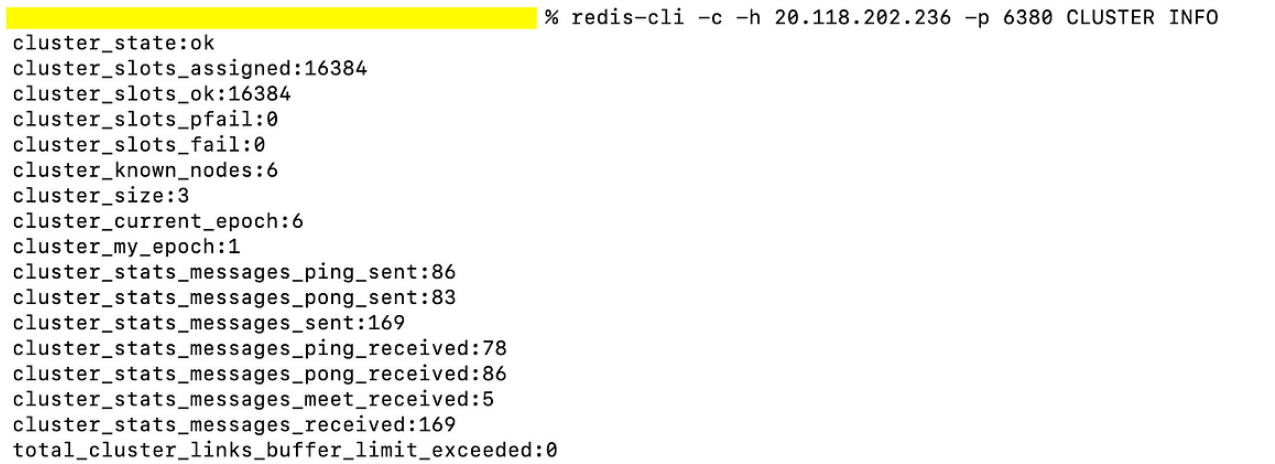
Above ordering during cluster creation has ensured that replica for master 1 is placed on node 2, replica for master 2 is placed on node 3 and replica for master 3 is placed on node 1.

This set up works well until any of the node leaves the cluster and joins back. If a node leaves the cluster (owing to failure) and joins back, the above equation is disturbed as the replica would have been promoted to master while node was down and after restoration there may be 2 masters on same node and 2 replicas on another node.

List cluster info

```
$ redis-cli -c -h <Public IP Address of any VM> -p 6380 CLUSTER INFO
```

It will list the cluster info as follows:



List cluster nodes

```
$ redis-cli -c -h <Public IP Address of any VM> -p 6380 CLUSTER NODES
```

It will list the nodes information as follows. As you can see, there are 3 masters and 3 slaves distributed across 3 nodes.

```
% redis-cli -c -h 20.118.202.236 -p 6380 CLUSTER NODES
8cee262aa762c6efc3e18510ce47a875d9349274 20.40.220.121:6380@16380 master - 0 1732645986569 3 connected 10923-16383
d16fc2c6f99b859dd1368fe4db13f284c7e7699a 20.118.38.219:6380@16380 master - 0 1732645984000 2 connected 5461-10922
4f893c4b51a48bcf40f612430d80902c595e8c5f 20.118.202.236:6381@16381 slave 8cee262aa762c6efc3e18510ce47a875d9349274 0 1732645984562 3 connected
e2fe6f002b7e7036535f723b910a88db8be2e45b 20.40.220.121:6381@16381 slave d16fc2c6f99b859dd1368fe4db13f284c7e7699a 0 1732645986000 2 connected
1aefd47982e839ca9b2c79ffd40577f3aa2cc6a2 10.0.0.10:6380@16380 myself,master - 0 1732645985000 1 connected 0-5460
2e4788132072b30e6ba7700404023ffd76b4f85f 20.118.38.219:6381@16381 slave 1aefd47982e839ca9b2c79ffd40577f3aa2cc6a2 0 1732645985000 1 connected
```

List cluster slots

```
$ redis-cli -c -h <Public IP Address of any VM> -p 6380 CLUSTER SLOTS
```

It will list slots information as follows (*total 16384 slots distributed across 3 nodes — this is the basis of sharding)*

```
% redis-cli -c -h 20.118.202.236 -p 6380 CLUSTER SLOTS
1) 1) (integer) 0
   2) (integer) 5460
   3) 1) "10.0.0.10"
      2) (integer) 6380
      3) "1aefd47982e839ca9b2c79ffd40577f3aa2cc6a2"
      4) (empty array)
   4) 1) "20.118.38.219"
      2) (integer) 6381
      3) "2e4788132072b30e6ba7700404023ffd76b4f85f"
      4) (empty array)
2) 1) (integer) 5461
   2) (integer) 10922
   3) 1) "20.118.38.219"
      2) (integer) 6380
      3) "d16fc2c6f99b859dd1368fe4db13f284c7e7699a"
      4) (empty array)
   4) 1) "20.40.220.121"
      2) (integer) 6381
      3) "e2fe6f002b7e7036535f723b910a88db8be2e45b"
      4) (empty array)
3) 1) (integer) 10923
   2) (integer) 16383
   3) 1) "20.40.220.121"
      2) (integer) 6380
      3) "8cee262aa762c6efc3e18510ce47a875d9349274"
      4) (empty array)
   4) 1) "20.118.202.236"
      2) (integer) 6381
      3) "4f893c4b51a48bcf40f612430d80902c595e8c5f"
      4) (empty array)
```

Test the set up by connecting to any of the redis master instances, creating key value pair and checking that key value pair shows up in its replica as well. Please note here that keys are hashed and automatically created in appropriate master based on hashed value. This is how the load is distributed across shards.

Connect to any of the master nodes

```
$ redis-cli -c -h <Public IP Address of any VM> -p 6380
```

Try setting a key value pair. It may be redirected to appropriate master node/slot based on hashing

```
% redis-cli -c -h 20.118.38.219 -p 6380
[20.118.38.219:6380> set k1 v1
 -> Redirected to slot [12706] located at 20.118.202.236:6381
 OK
```

Please note my master-slave configuration has changed since my machines were shut down. This is the problem I was referring earlier that when the nodes fail/shut down/leave the cluster and join back, the cluster configuration may change leading to redistribution of masters and slaves. Now one node has 2 masters and other one has 2 slaves.. This can be fixed by reconfiguring the cluster but may need to go through certain steps (not covered here).

```
% redis-cli -c -h 20.118.202.236 -p 6380 CLUSTER NODES
4f893c4b51a48bcf40f612430d80902c595e8c5f 20.118.202.236:6381@16381 master - 0 1732651381000 7 connected 10923-16383
8cee262aa762c6efc3e18510ce47a875d9349274 20.40.220.121:6380@16380 slave 4f893c4b51a48bcf40f612430d80902c595e8c5f 0 1732651381000 7 connected
2e4788132072b30e6ba7700404023ffd76b4f85f 20.118.38.219:6381@16381 slave 1aefd47982e839ca9b2c79ffd40577f3aa2cc6a2 0 1732651381653 1 connected
1aefd47982e839ca9b2c79ffd40577f3aa2cc6a2 10.0.0.10:6380@16380 myself,master - 0 1732651380000 1 connected 0-5460
d16fc2c6f99b859dd1368fe4db13f284c7e7699a 20.118.38.219:6380@16380 master - 0 1732651381000 2 connected 5461-10922
e2fe6f002b7e7036535f723b910a88db8be2e45b 20.40.220.121:6381@16381 slave d16fc2c6f99b859dd1368fe4db13f284c7e7699a 0 1732651382656 2 connected
```

Anyways, let's continue and try retrieving above key from the master instance where its created

```
% redis-cli -c -h 20.118.202.236 -p 6381
[20.118.202.236:6381> get k1
 "v1"
```

Also, let's try to retrieve the same key from replica of above master. It will return the value by redirecting to appropriate master which holds the key

```
% redis-cli -c -h  20.40.220.121 -p 6380
[20.40.220.121:6380> get k1
 -> Redirected to slot [12706] located at 20.118.202.236:6381
 "v1"
```

SETTING UP AND TESTING REDIS SENTINEL

Let's consider new set of VMs here (VM4, VM5, VM6)

SSH into VM4

```
$ ssh -i <key> azureuser@<VM4 IP Address>
```

Update packages

```
$ sudo apt update
```

Install redis server

```
$ sudo apt install redis-server -y
```

Open redis configuration file (created by default during redis server installation)

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

Enable/Update the following options in above configuration file

```
bind 0.0.0.0
```

Start redis service

```
$ sudo service redis-server restart
```

SSH into VM5

```
$ ssh -i <key> azureuser@<VM5 IP Address>
```

Update packages

```
$ sudo apt update
```

Install redis server

```
$ sudo apt install redis-server -y
```

Open redis configuration file (created by default during redis server installation)

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

Enable/Update the following options in above configuration file

```
bind 0.0.0.0protected-mode noreplicaof <VM4 (Master) Private IP Address> 6379
```

Start redis service

```
$ sudo service redis-server restart
```

Repeat exact same steps on VM6 to create second replica

Connect to master node

```
$ redis-cli -c -h <VM4 (Master) Public IP Address  -p 6379
```

Try adding key-value pair

```
<IP Address of any VM>:6379> SET k1 v1
```

It should return “OK status

Try retrieving the key ‘k1"

```
<IP Address of any VM>:6379> GET k1
```

It should return value “v1”

Now connect to any of the slave nodes

```
$ redis-cli -c -h <VM5 (Slave) Public IP Address  -p 6379
```

Try retrieving the key ‘k1"

```
<IP Address of any VM>:6379> GET k1
```

It should return value “v1” as the key-value pair has been replicated on slave

Now try adding key-value pair to slave node

```
<IP Address of any VM>:6379> SET k1 v1
```

It should give an error — “(error) READONLY You can’t write against a read only replica.”

Configure Sentinel on each of the VMs to manage the Master-Replica set up such that if master dies, one of the replicas is promoted as master.

SSH into VM

```
$ ssh -i <key> azureuser@<VM IP Address>
```

Create sentinel configuration file

```
$ sudo nano /etc/redis/sentinel.conf
```

Add the following to sentinel configuration file

```
daemonize yesport 26379bind 0.0.0.0supervised systemdpidfile "/run/redis/redis-
sentinel.pid"logfile "/var/log/redis/sentinel.log"sentinel monitor mymaster <VM4
(Master) Private IP Address> 6379 2sentinel down-after-milliseconds mymaster
5000sentinel failover-timeout mymaster 60000sentinel parallel-syncs mymaster 1
```

Change ownership of configuration file to redis

```
$ sudo  redis:redis /etc/redis/sentinel.conf
```

Create systemd file to automatically start sentinel service on reboot

```
$ sudo nano /etc/systemd/system/redis-sentinel.service
```

Add below configuration to above file

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

```
[Unit]Description=Redis SentinelAfter=network.target
```

```
[Service]User=redisGroup=redisType=notifyExecStart=/usr/bin/redis-server
/etc/redis/sentinel.conf --sentinelExecStop=/usr/bin/redis-cli
shutdownRestart=always
```

Reload the daemon

```
$ sudo systemctl daemon-reload
```

Start sentinel service

```
$ sudo service redis-sentinel start
```

Make sure sentinel is set up and started on each of the VMs (master and replicas).

Connect to any of the nodes over sentinel port

```
$ redis-cli -c -h <Public IP Address of any VM> -p 26379
```

List master nodes monitored by sentinel

```
<IP Address of any VM>:26379> SENTINEL masters
```

It will list the master nodes information as follows (*note the highlighted sections*):

```
[prafulkhandelwal@macus-ee210296 Downloads % redis-cli -c -h  135.233.112.151 -p 26379
[135.233.112.151:26379> SENTINEL masters
 1)  1) "name"
     2) "mymaster"
     3) "ip"
     4) "10.0.0.8"
     5) "port"
     6) "6379"
     7) "runid"
     8) "bfb3b67073037e7701351ffa8e4dfa1cb95df403"
     9) "flags"
    10) "master"
    11) "link-pending-commands"
    12) "0"
    13) "link-refcount"
    14) "1"
    15) "last-ping-sent"
    16) "0"
    17) "last-ok-ping-reply"
    18) "283"
    19) "last-ping-reply"
    20) "283"
    21) "down-after-milliseconds"
    22) "5000"
    23) "info-refresh"
    24) "2073"
    25) "role-reported"
    26) "master"
    27) "role-reported-time"
    28) "172939"
    29) "config-epoch"
    30) "3"
    31) "num-slaves"
    32) "2"
    33) "num-other-sentinels"
    34) "2"
    35) "quorum"
    36) "2"
    37) "failover-timeout"
    38) "60000"
    39) "parallel-syncs"
    40) "1"
```

List slave nodes (replicas) for above master

```
<IP Address of any VM>:26379> SENTINEL slaves mymaster
```

It will list the slave nodes information as follows (*note the highlighted sections*):

```
[135.233.112.151:26379> SENTINEL slaves mymaster
 1)  1) "name"
     2) "10.0.0.4:6379"
     3) "ip"
     4) "10.0.0.4"
     5) "port"
     6) "6379"
     7) "runid"
     8) "df3516ddd06c011d20b845238aaf188798e9d916"
     9) "flags"
    10) "slave"
    11) "link-pending-commands"
    12) "0"
    13) "link-refcount"
    14) "1"
    15) "last-ping-sent"
    16) "0"
    17) "last-ok-ping-reply"
    18) "882"
    19) "last-ping-reply"
    20) "882"
    21) "down-after-milliseconds"
    22) "5000"
    23) "info-refresh"
    24) "8802"
    25) "role-reported"
    26) "slave"
    27) "role-reported-time"
    28) "361307"
    29) "master-link-down-time"
    30) "0"
    31) "master-link-status"
    32) "ok"
    33) "master-host"
    34) "10.0.0.8"
    35) "master-port"
    36) "6379"
    37) "slave-priority"
    38) "100"
    39) "slave-repl-offset"
    40) "3352744"
    41) "replica-announced"
    42) "1"
 2)  1) "name"
     2) "10.0.0.12:6379"
     3) "ip"
     4) "10.0.0.12"
     5) "port"
     6) "6379"
     7) "runid"
     8) "cacd3673f48972eb238c3a8d7fd00f969e069740"
     9) "flags"
    10) "slave"
    11) "link-pending-commands"
    12) "0"
    13) "link-refcount"
    14) "1"
    15) "last-ping-sent"
    16) "0"
    17) "last-ok-ping-reply"
    18) "545"
    19) "last-ping-reply"
    20) "545"
    21) "down-after-milliseconds"
```

- Now, shutdown the master node (one with private IP address 10.0.0.8). Sentinel should promote one of the slaves as master
- List master nodes again

```
<IP Address of any VM>:26379> SENTINEL masters
```

Note the IP address of master. It changed from 10.0.0.8 to 10.0.0.12, which means one of the slaves got promoted to master

```
[135.233.112.151:26379> SENTINEL masters
 1)  1) "name"
     2) "mymaster"
     3) "ip"
     4) "10.0.0.12"
     5) "port"
     6) "6379"
     7) "runid"
     8) "cacd3673f48972eb238c3a8d7fd00f969e069740"
     9) "flags"
    10) "master"
    11) "link-pending-commands"
    12) "0"
    13) "link-refcount"
    14) "1"
    15) "last-ping-sent"
    16) "0"
    17) "last-ok-ping-reply"
    18) "773"
    19) "last-ping-reply"
    20) "773"
    21) "down-after-milliseconds"
    22) "5000"
    23) "info-refresh"
    24) "8197"
    25) "role-reported"
    26) "master"
    27) "role-reported-time"
    28) "78506"
    29) "config-epoch"
    30) "4"
    31) "num-slaves"
    32) "2"
    33) "num-other-sentinels"
    34) "2"
    35) "quorum"
    36) "2"
    37) "failover-timeout"
    38) "60000"
    39) "parallel-syncs"
    40) "1"
```

List the slave nodes

```
<IP Address of any VM>:26379> SENTINEL slaves mymaster
```

Note down the first slave node. It is the previous master (10.0.0.8), which is now down and has acquired slave role

```
[135.233.112.151:26379> SENTINEL slaves mymaster
1)  1) "name"
    2) "10.0.0.8:6379"
    3) "ip"
    4) "10.0.0.8"
    5) "port"
    6) "6379"
    7) "runid"
    8) ""
    9) "flags"
   10) "s_down,slave"
   11) "link-pending-commands"
   12) "66"
   13) "link-refcount"
   14) "1"
   15) "last-ping-sent"
   16) "238900"
   17) "last-ok-ping-reply"
   18) "238900"
   19) "last-ping-reply"
   20) "238900"
   21) "s-down-time"
   22) "233825"
   23) "down-after-milliseconds"
   24) "5000"
   25) "info-refresh"
   26) "0"
   27) "role-reported"
   28) "slave"
   29) "role-reported-time"
   30) "238900"
   31) "master-link-down-time"
   32) "0"
   33) "master-link-status"
   34) "err"
   35) "master-host"
   36) "?"
   37) "master-port"
   38) "0"
   39) "slave-priority"
   40) "100"
   41) "slave-repl-offset"
   42) "0"
   43) "replica-announced"
   44) "1"
2)  1) "name"
    2) "10.0.0.4:6379"
    3) "ip"
    4) "10.0.0.4"
    5) "port"
    6) "6379"
    7) "runid"
    8) "df3516ddd06c011d20b845238aaf188798e9d916"
    9) "flags"
   10) "slave"
   11) "link-pending-commands"
   12) "0"
   13) "link-refcount"
   14) "1"
   15) "last-ping-sent"
   16) "0"
   17) "last-ok-ping-reply"
   18) "171"
   19) "last-ping-reply"
```

Now start the node with private IP address 10.0.0.8 and list the slaves again.

```
<IP Address of any VM>:26379> SENTINEL slaves mymaster
```

You can notice that the slave (10.0.0.8) is restored.

```
[135.233.112.151:26379> SENTINEL slaves mymaster
1)  1) "name"
    2) "10.0.0.8:6379"
    3) "ip"
    4) "10.0.0.8"
    5) "port"
    6) "6379"
    7) "runid"
    8) "795333e78a29b3f22202a22b68013de745eec644"
    9) "flags"
   10) "slave"
   11) "link-pending-commands"
   12) "0"
   13) "link-refcount"
   14) "1"
   15) "last-ping-sent"
   16) "0"
   17) "last-ok-ping-reply"
   18) "335"
   19) "last-ping-reply"
   20) "335"
   21) "down-after-milliseconds"
   22) "5000"
   23) "info-refresh"
   24) "5595"
   25) "role-reported"
   26) "slave"
   27) "role-reported-time"
   28) "15616"
   29) "master-link-down-time"
   30) "0"
   31) "master-link-status"
   32) "ok"
   33) "master-host"
   34) "10.0.0.12"
   35) "master-port"
   36) "6379"
   37) "slave-priority"
   38) "100"
   39) "slave-repl-offset"
   40) "3477949"
   41) "replica-announced"
   42) "1"
2)  1) "name"
    2) "10.0.0.4:6379"
    3) "ip"
    4) "10.0.0.4"
    5) "port"
    6) "6379"
    7) "runid"
    8) "df3516ddd06c011d20b845238aaf188798e9d916"
    9) "flags"
   10) "slave"
   11) "link-pending-commands"
   12) "0"
   13) "link-refcount"
   14) "1"
   15) "last-ping-sent"
   16) "0"
   17) "last-ok-ping-reply"
   18) "804"
   19) "last-ping-reply"
   20) "804"
   21) "down-after-milliseconds"
```

Pitfalls / Challenges

As you go through above setup, you may encounter some challenges initially:

- If you are setting up Redis cluster with 3 machines/nodes, with 1 master and 1 replica on each node, you may want to ensure that any master and its replica do not reside on same node to ensure high availaiblty and fault tolerence. However, when any node goes down and joins back the cluster, the cluster configuration changes as one of the slaves got promoted to master when the node went down. This may result in one node having 2 masters now and another one having 2 slaves. To avoid this, 6-node set up is recommended — 3 for masters and 3 for slaves

- Make sure the redis service/instances are running over configured ports on all the nodes. Best is to set them up as daemon services running in background such that they can survive reboot. In same cases, I didn’t do that and had to manually restart redis instances after any reboot when I could not connect from my local machine
- For POC sake, I relaxed the security by binding to 0.0.0.0, However, this poses a significant security risk as binding Redis 0.0.0.0 means that Redis will accept incoming connections from any IP address, essentially making it accessible from any network interface on the host machine. So, you got to be careful with that
- As I used very small VMs for this POC, I often encountered memory related errors. You may want to choose appropriate configuration for your needs depending upon what all processes you want to run on those machines
- Its best to not touch the default configuration file generated by Redis and rather create and use the copies so that we can always refer back the original configuration, if needed, after making the modifications

Rethinking Netflix's Edge Load Balancing

netflixtechblog.com/netflix-edge-load-balancing-695308b5548c

Netflix Technology Blog September 28, 2018

Learn about Netflix's world class engineering efforts, company culture, product developments and more.

By , Netflix Cloud Gateway

We briefly touched on some of the load balancing improvements we've recently been making in our Open Sourcing Zuul 2 post. In this post, we'll go into more detail on the whys, hows and results of that work.

On the Netflix Cloud Gateway team we are always working to help systems reduce errors, gain higher availability, and improve Netflix's resilience to failures. We do this because even a low rate of errors at our scale of over a million requests per second can degrade the experience for our members, so every little bit helps.

So we set out to take our learnings from Zuul, plus those from other teams, and improve our load balancing implementation to further reduce errors caused by *overloaded servers*.

Background

In Zuul, we've historically used the Ribbon load balancer with a round-robin algorithm and some filtering mechanisms for blacklisting servers that have a high connection failure rate.

Over the years we've gone through a couple of improvements and customizations designed to send less traffic to servers that have recently launched, to attempt to avoid overloading them. These have made significant improvements, but for some particularly troublesome origin clusters, we would still see much higher load-related error rates than desirable.

If all servers in a cluster are overloaded, then there's little improvement we can make in choosing one server over another, but we often see situations where **only a subset of servers are overloaded**. For example:

- Cold servers post-startup (during red-black deployments and auto-scaling events).
- Servers temporarily slowing/blocking due to staggered dynamic property/script/data updates or large GC events.
- Bad server hardware. We'll often see some servers permanently running slower than others, whether due to noisy neighbors or different hardware.

Guiding Principles

It can be useful to have some principles in mind when starting on a project, to help guide us with the flood of small and large decisions we need to make daily while designing software. Here are some that were used on this project.

Work within the constraints of the existing load balancer framework

We had coupled our previous load-balancing customizations to the Zuul codebase, which had precluded us from sharing these with other teams at Netflix. So this time we made a decision to accept the constraints and additional investment needed, and to design with reuse in mind from the get-go. This makes adoption in other systems more straightforward, and reduces the chances of reinventing-the-wheel.

Apply learnings from others

Try to build on top of the ideas andimplementations of others. For example, the choice-of-2 and probation algorithms which were previously trialled at Netflix in other IPC stacks.

Avoid Distributed State

Prefer local decision-making to avoid the resiliency concerns, complexities and lag of coordinating state across a cluster.

Avoid Client-Side Configuration and Manual Tuning

Our operational experience over the years with Zuul has demonstrated that situating parts of a services' configuration into a client service not owned by the same team … causes problems.

One problem is that these client-side configurations tend to either get out of sync with the changing reality of the server-side, *or* introduce coupling of change-management between services owned by different teams.

For example, the EC2 instance type used for Service X is upgraded, causing fewer nodes to be needed for that cluster. So now the "maximum number of connections per host" client-side configuration in Service Y should be increased to reflect the new increased capacity. Should the client-side change be made first, or the server-side change, or both at the same time? More likely than not, the setting gets forgotten about altogether and causes even more problems.

When possible, instead of configuring static thresholds, use adaptive mechanisms that change based on the current traffic, performance and environment.

When static thresholds *are* required, rather than have service teams coordinate threshold configurations out to each client, have the *services* communicate this at runtime to avoid the problems of pushing changes across team boundaries.

Load-Balancing Approach

An overarching idea was that while the best source of data for a servers' *latency* is the client-side view, the best source of data on a servers' *utilization* is from the server itself. And that combining these 2 sources of data should give us the most effective load-balancing.

We used a combination of mechanisms that complemented each other, most of which have been developed and used by others previously, though possibly not combined in this manner before.

- A to choose between servers.
- Primarily balance based on the of a servers' utilization.
- Secondarily balance based on the of its utilization.
- and based mechanisms for avoiding overloading newly launched servers.
- of collected server stats to zero over time.

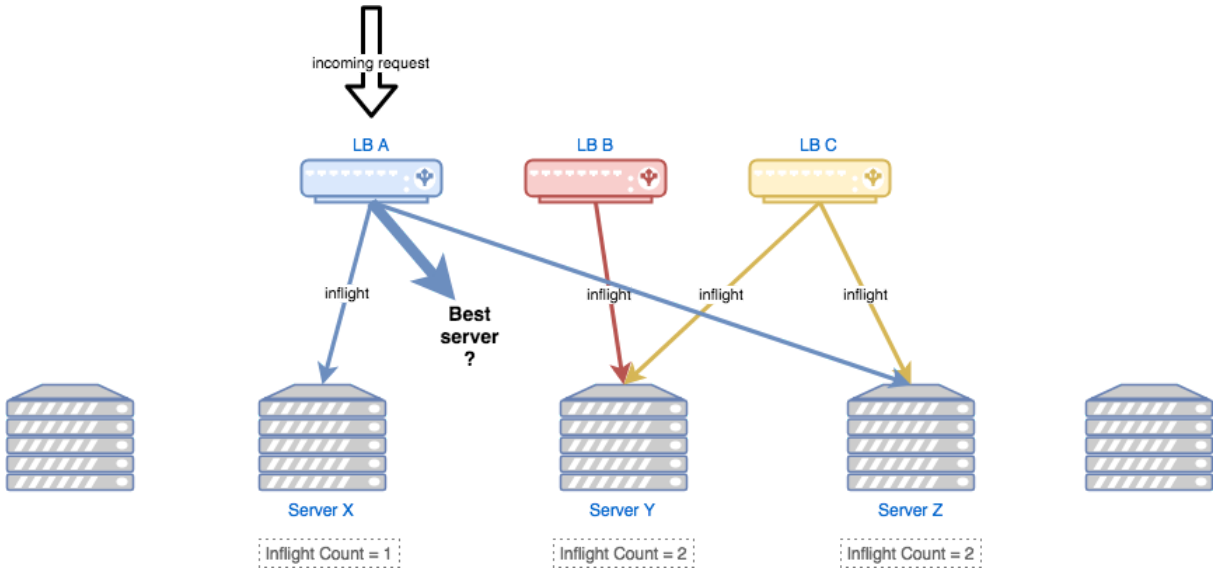
Combining Join-the-Shortest-Queue with Server-Reported Utilization

We chose to buttress the *Join-the-shortest-queue(JSQ)* algorithm that is commonly used, with a 2nd algorithm based on the *server-reported utilization* to try and give us the best of both.

Problems with JSQ Join-the-shortest-queue works very well for a *single* load-balancer, but has a significant problem if used in isolation across a *cluster* of load balancers. The problem is that the load balancers will tend to *herd* and all choose the *same* low-utilized servers at the same time, thus overloading them, and then moving on to the next least-utilized and overloading that, and onward …

This can be resolved by using JSQ in combination with the choice-of-2 algorithm. This mostly removes the herding problem, and works well except for some deficiencies around each load balancer not having a complete picture of *server utilization*.

JSQ is generally implemented by counting the number of in-use connections to a server from only the *local* load balancer, but when there are 10's-100's of load balancer nodes, the local view can be misleading.



A single load balancers' view can be very different from the actual reality

For example in this diagram, load balancer A has 1 inflight request to server X and 1 to server Z, but none to server Y. So when it receives a new request and has to choose which server is least utilized — from the data it has available to it — it will choose server Y. This is not the correct choice though — server Y is actually the *most* utilized, as two other load balancers currently have inflight requests to it, but load balancer A has no way of knowing that.

This illustrates how a single load balancers' viewpoint can be entirely different from the actual reality.

Another problem we experience with relying *only* on the client-side view, is that for large clusters — particularly when combined with low traffic — a load balancer often has only a few in-use connections to a subset of the servers in a pool of hundreds. So when it's choosing which server is least-loaded, it can often just be choosing between zero and zero — ie. it has no data on the utilization of either of the servers it's choosing between, so has to just guess randomly.

One solution to this problem could be to share the state of each load balancers' inflight counts with all other load balancers … but then you have a distributed state problem to solve.

We generally employ distributed mutable state only as a last resort, as the value gained needs to outweigh the substantial costs involved:

- Operational overhead and complexity it adds to tasks like deployments and canarying.
- Resiliency risks related to the blast radius of data corruptions (ie. bad data on 1% of load balancers is an annoyance, bad data on 100% of them is an outage).
- The cost of either implementing a P2P distributed state system between the load balancers, or the cost of operating a separate database with the performance and resiliency credentials needed to handle this massive read and write traffic.

An alternative simpler solution — and one that we've chosen — is to instead rely on the **servers reporting to each load balancer** how utilized they are …

Server-Reported Utilization Using each server's viewpoint on their utilization has the advantage that it provides the aggregate of all load balancers that are using that server, and therefore avoids the JSQ problem of an incomplete picture.

There were 2 ways we could have implemented this — either:

1. poll for each servers' current utilization using health-check endpoints.
2. track responses from the servers annotated with their current utilization data.

We chose the 2nd option, as it was simple, allowed for frequent updating of this data, and avoided the additional load placed on servers of having N load balancers poll M servers every few seconds.

An impact of this *passive* strategy is that the more frequently a load balancer sends a request to one server, the more up-to-date it's view of that servers' utilization. So the higher the RPS, the higher the effectiveness of the load-balancing. But then conversely, the lower the RPS, the less effective the load-balancing.

This hasn't been an issue for us, but it is likely that for services that receive a low RPS through one particular load balancer (while receiving high RPS through another separate one), actively polling health-checks could be more effective. The tipping point would be where the load balancer is sending a lower RPS to each server than the polling frequency used for the health-checks.

Server Implementation We implemented this on the server side by simply tracking the *inflight request count*, converting it to a percentage of the configured max for that server, and writing that out as a HTTP response header:

```
X-Netflix.server.utilization: <current-utilization>
```

The optional target-utilization can be specified by a server to indicate what percentage utilization they are intending to operate at under normal conditions. This is then used by the load balancer for some coarse-grained filtering done as described later.

We experimented a little with using metrics other than the inflight count, such as operating system reported cpu utilization and load average, but found them to cause oscillations seemingly due to the lag induced by them being based on rolling averages. Thus, we decided to just stick with the relatively simple implementation of just counting inflight requests for now.

Choice-of-2 Algorithm Instead of Round-Robin

As we wanted to be able to choose between servers by comparing their statistics, the existing simple round-robin implementation was going to have to go.

An alternative within Ribbon that we tried was JSQ combined with a ServerListSubsetFilter to reduce the *herding* problem of distributed-JSQ. This gave reasonable results, but the resulting request distribution across target servers was still much too wide.

So we instead applied some earlier learnings from another team at Netflix and implemented the **Choice-of-2** algorithm. This has the advantage of being simple to implement, keeps cpu cost on the load balancer low, and gives good request distribution.

Choose based on Combination of Factors

To choose between servers, we compare them on 3 separate factors:

1. : rolling percentage of connection-related errors for that server.
2. : most recent score provided by that server.
3. : current number of inflight requests to that server from this load balancer.

These 3 factors are used to assign scores to each server, and then the aggregate scores are compared to choose the winner.

Using multiple factors like this does make the implementation more complicated, but it hedges against edge-case problems that can occur from relying solely on one factor.

For example, if one server starts failing and rejecting all requests, it's reported utilization will be much lower — due to rejecting requests being faster than accepting them — and if that was the only factor used, then all the load balancers would start sending *more* requests to that bad server. The client-health factor mitigates against that scenario.

Filtering

When randomly choosing the 2 servers to compare, we filter out any servers that are above conservatively configured thresholds for utilization and health.

This filtering is done on each request to avoid the staleness problems of filtering only periodically. To avoid causing high cpu load on the load balancer, we only do a *best-effort* by making N attempts to find a randomly chosen viable server, and then falling-back to non-filtered servers if necessary.

Filtering like this helps significantly when a large proportion of the pool of servers have persistent problems. As in that scenario, randomly choosing 2 servers will frequently result in 2 *bad* servers being chosen for comparison, even though there were many *good* servers available.

This does though have the disadvantage of depending on statically configured thresholds, which we've tried to avoid. The results of testing though, convinced us this was worthwhile adding, even with some general-purpose (ie. non service-specific) thresholds used.

Probation

For any server that the load-balancer hasn't yet received a response from, we only allow one inflight request at a time. We filter out these *in-probation* servers until a response is received from them.

This helps to avoid overloading newly launched servers with a flood of requests, before giving them a chance to indicate how utilized they are.

Server-Age Based Warmup

We use server age to progressively ramp up traffic to newly launched servers over the course of their first 90 seconds.

This is another useful mechanism like *probation* that adds some caution about overloading servers in the sometimes delicate post-launch state.

Stats Decay

To ensure that servers don't get permanently blacklisted, we apply a decay rate to all stats collected for use in the load-balancing (currently a linear decay over 30 secs). So for example, if a server's error-rate goes up to 80% and we stop sending it traffic, then the value we use will decay down to zero over 30 seconds (ie. it will be 40% after 15 secs).

Operational Impacts

Wider Request Distribution

A negative impact of moving away from using round-robin for load-balancing, is that where before we had a very tight distribution of requests across servers in a cluster, we now get a larger delta between servers.

Using the choice-of-2 algorithm has helped to mitigate this a lot (compared to JSQ across all or a subset of servers in a cluster), but it's not been possible to avoid it entirely.

So this does need to be taken into account on the operational side, particularly for *canary analysis* where we typically compare *absolute values* of request counts, error rates, cpu, etc.

Slower Servers Receive Less Traffic

Obviously this is the intended effect, but for teams used to round-robin where traffic is apportioned equally, this has some knock-on effects on the operational side.

As the traffic distribution across origin servers is now dependent on their utilization, if some servers are running a different build that is more or less efficient, then they will receive more or less traffic. So:

- When clusters are being red-black deployed, if the new server-group has a performance regression, then the proportion of traffic to that group will be less than 50%.
- The same effect can be seen with canaries — the baseline may receive a different volume of traffic than the canary cluster. So when looking at metrics, its best to look at the combination of RPS and CPU (as for example RPS may be lower in the canary, while CPU is the same).
- Less effective — we typically have automation that monitors for outlier servers (typically a VM thats slow immediately from launch due to some hardware issue) within a cluster and terminates them. When those outliers receive less traffic due to the load-balancing, this detection is more difficult.

Rolling Dynamic Data Updates

Moving from round-robin to this new load balancer has the useful effect of working very well in conjunction with staged rollouts of dynamic data and properties.

Our best practice is to deploy data updates one region(datacenter) at a time to limit the blast radius of unanticipated problems.

Even without any problems caused by the data update itself, the act of a server *applying* an update can cause a brief load spike (typically GC related). If this spike occurs simultaneously on all servers in a cluster, it can lead to a large spike in load-shedding and errors propagating upstream. In this scenario there's little a load balancer can do to help, as *all* servers are experiencing the high load.

A solution though — when used in combination with an adaptive load balancer like this — is to do *rolling* data updates across the servers in a cluster. If only a small a proportion of the servers are applying the update at once, then the load balancer can briefly reduce traffic to them, as long as there are enough other servers in the fleet to take the diverted traffic.

Synthetic Load-Testing Results

We used synthetic load-testing scenarios extensively while developing, testing and tuning the different aspects of this load-balancer. These were very useful in verifying the effectiveness with real clusters and networks, as a reproducible step above unit-tests, but without yet using real user traffic.

More details of this testing are listed later in the *appendix*, but summarizing the main points:

- The new load balancer, with all features enabled, gave compared to the round-robin implementation.
- There was a (a 3x reduction compared to the round-robin implementation).
- The addition of value, providing one order of magnitude of the error reductions, and the majority of the latency reductions.

Implementation	Total Failed Request Count	Avg Latency Kept Below (ms)
New w/ all features enabled	5	375
New w/ server-utilization disabled	123	1200
Original implementation	14,700	1400

Comparison of the Results

Impact on Real Production Traffic

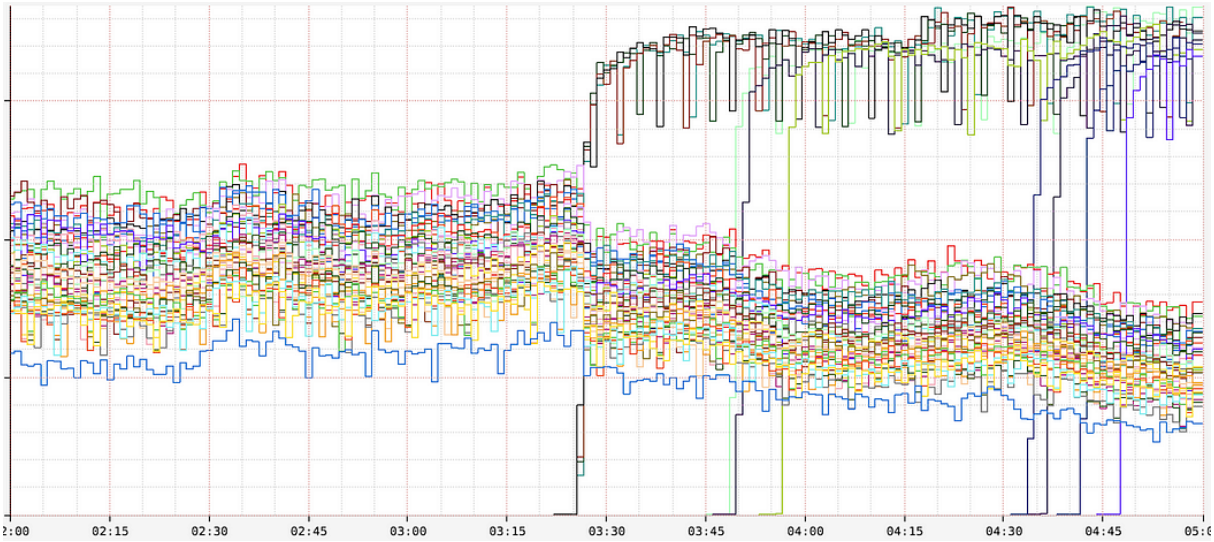
We've found the new load balancer to be very effective at distributing as much traffic to each origin server as that server can handle. This has the valuable effect of routing around both intermittently *and* persistently degraded servers, without any manual intervention, which has avoided significant production problems from causing engineers to be woken in the middle of the night.

It's difficult to illustrate the impact of this during normal operation, but it can be seen during production incidents, and for some services even during their normal steady-state operation.

During Incidents

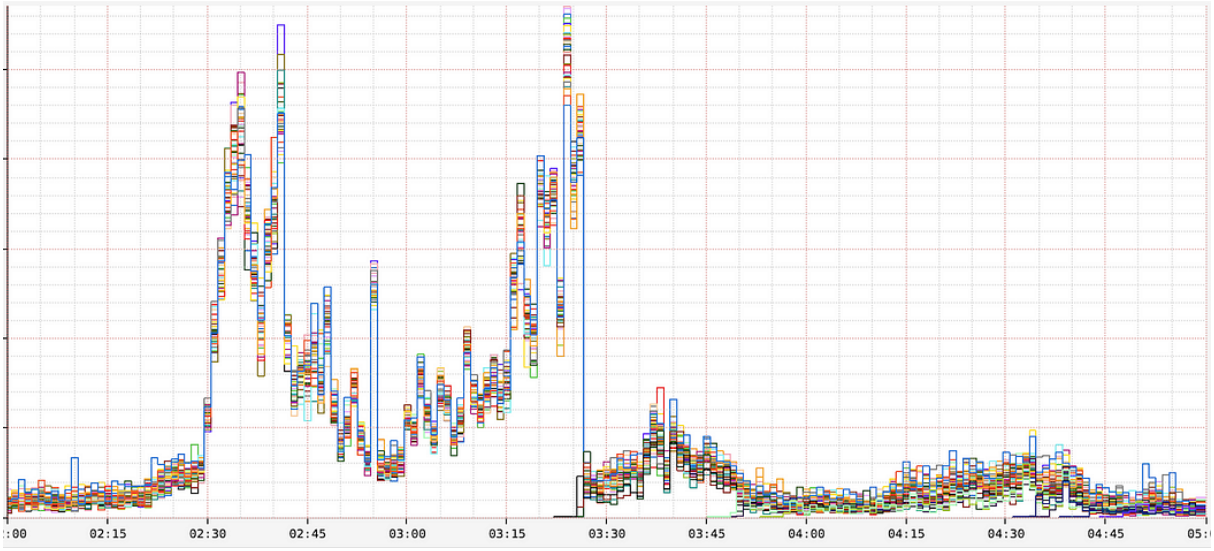
A recent incident involved a bug in a service that was causing more and more server threads to block over time — ie. from the point a server launched, a few threads would block each hour until it eventually started to reach its maximum and shed load.

In the below chart of RPS per server, you can see that before 3am there was a wide distribution across the servers. This was due to the servers with higher numbers of blocked threads being sent less traffic by the load balancers. Then after 3:25am autoscaling started launching more servers, each of which received roughly double the RPS of the existing servers — as they did not yet have any blocked threads and therefore could successfully handle more traffic.



Now if we look at this chart of the rate of errors per server over the same time range, you can see that the distribution of errors across all the servers throughout the incident was fairly evenly distributed, even though we know that some servers had much less capacity than others. This indicates that the load balancers were working effectively, and that all servers were being pushed slightly past their effective capacity, due to too little overall available capacity in the cluster.

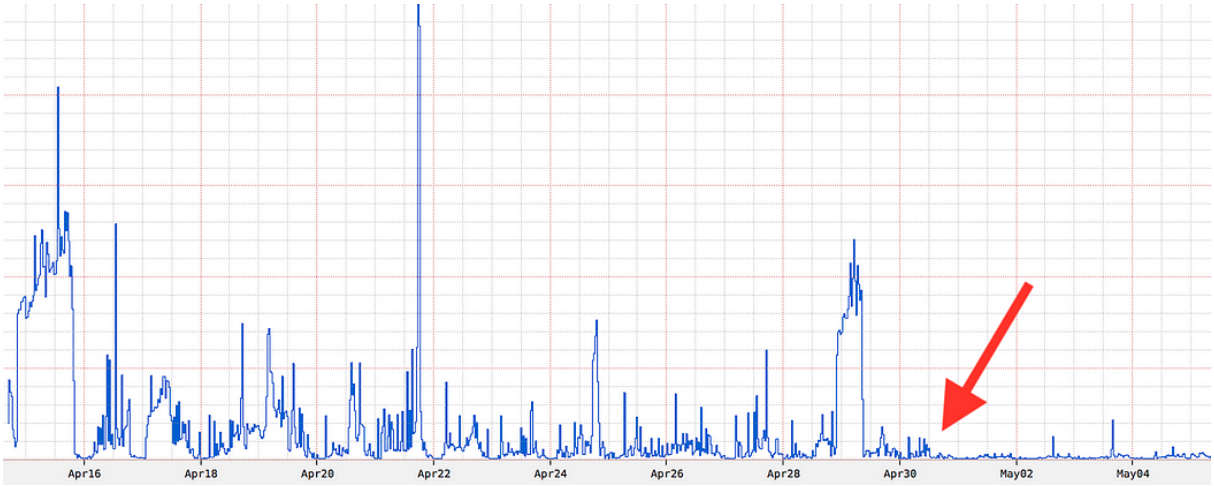
Then when autoscaling launched new servers, they were sent as much traffic as they could handle, up to the point that they were erroring at the same low level as the rest of the cluster.



So in summary, the load-balancing was very effective at distributing traffic to the servers, but in this case not enough new servers were launched to bring the overall error level all the way back down to zero.

Steady-State

We have also seen a significant reduction in just the steady-state noise in some services of servers having blips of load-shedding due to GC events for a few seconds. That can be seen here where the errors reduced substantially after the new load balancer was enabled:



Alerting Gaps

An unanticipated impact has been to highlight some gaps in our automated alerts. Some existing alerts based on error rates from services, which would previously have fired when progressive problems were only affecting a small subset of a cluster, now fire much later or not at all, as error rates are kept lower. This has meant that teams have *not* been notified of sometimes large problems affecting their clusters. The solution has been to plug these gaps by adding additional alerts on deviations in utilization metrics rather than just error metrics.

Conclusion

This post isn't intended as a plug for Zuul — although it is a great system — but more to share and add another datapoint to the wealth of interesting approaches out there in the proxying/service-mesh/load-balancing community. Zuul is a great system to test, implement, and improve these kinds of load balancing schemes; running them with the demands and scale of Netflix gives us the capabilities to prove and improve these approaches.

There are many different approaches that can be taken to improve load-balancing, and this one has worked well for us, producing significant reductions in load-related error rates and much improved balancing of real-world load on servers.

As with any software system though — you should make decisions based on your own organizations' constraints and goals, and try to avoid chasing the gnat of perfection.

If this kind of work is interesting to you feel free to reach out to me or us here on the Cloud Gateway team at Netflix.

Appendix — Results of Synthetic Load-Testing

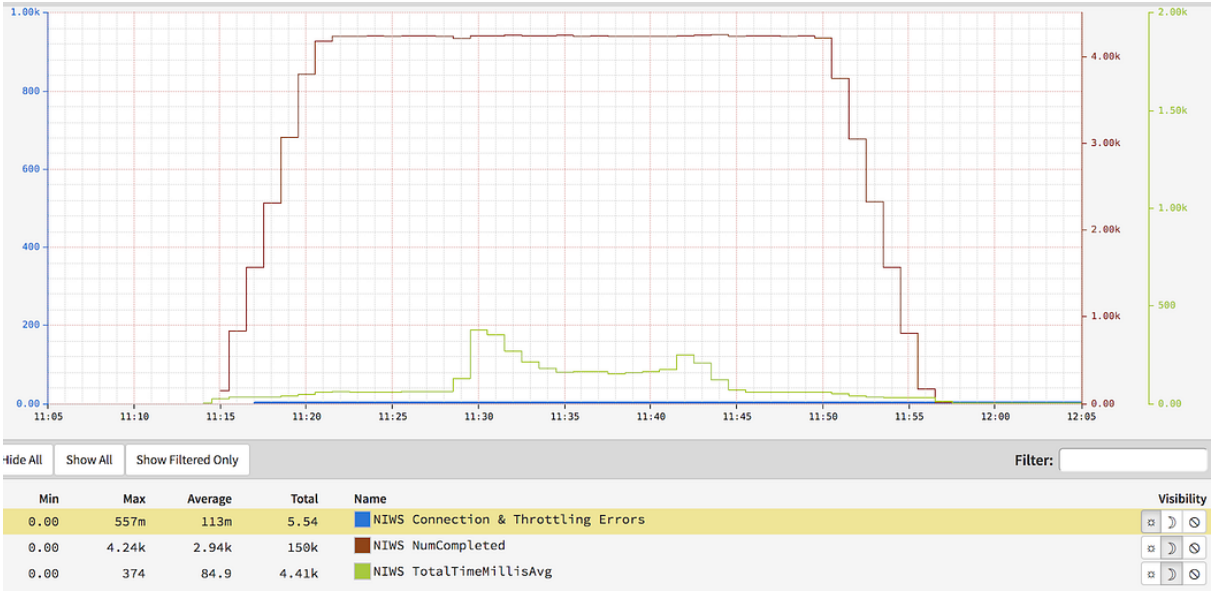
Test Scenario

This load test scenario reproduces the situation where a small origin cluster is going through a red-black deployment, and the newly deployed cluster is having cold startup problems or has some kind of performance regression. This was simulated by artificially injecting additional latency and cpu load for each request on the newly deployed servers.

The test ramps-up to 4000 RPS sent to a large Zuul cluster (200 nodes) which in turn proxies to a small Origin cluster (20 instances), and after a few mins, enabling the 2nd origin cluster (another 20 instances).

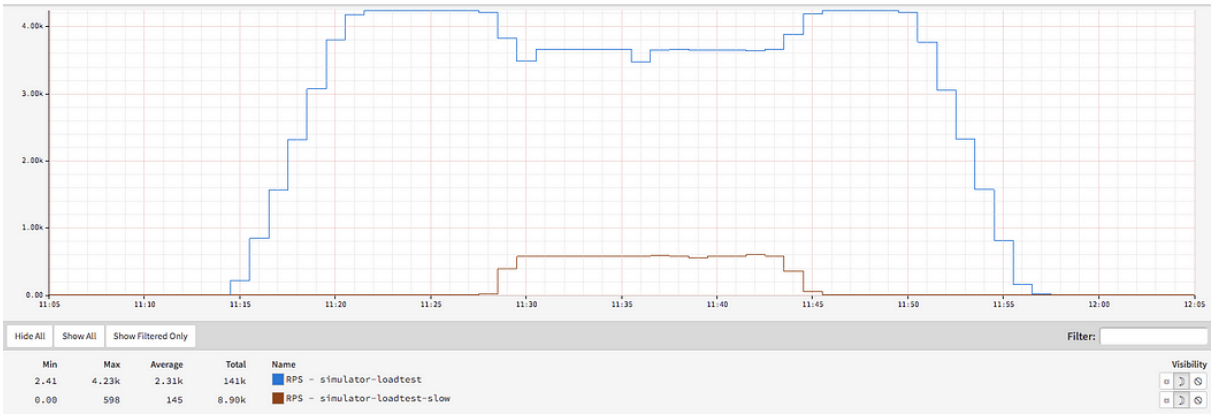
All Features Enabled

Here is a chart of the metrics for the new load balancer with all of it's features enabled.



Load-Test with New Load Balancer and All Features Enabled

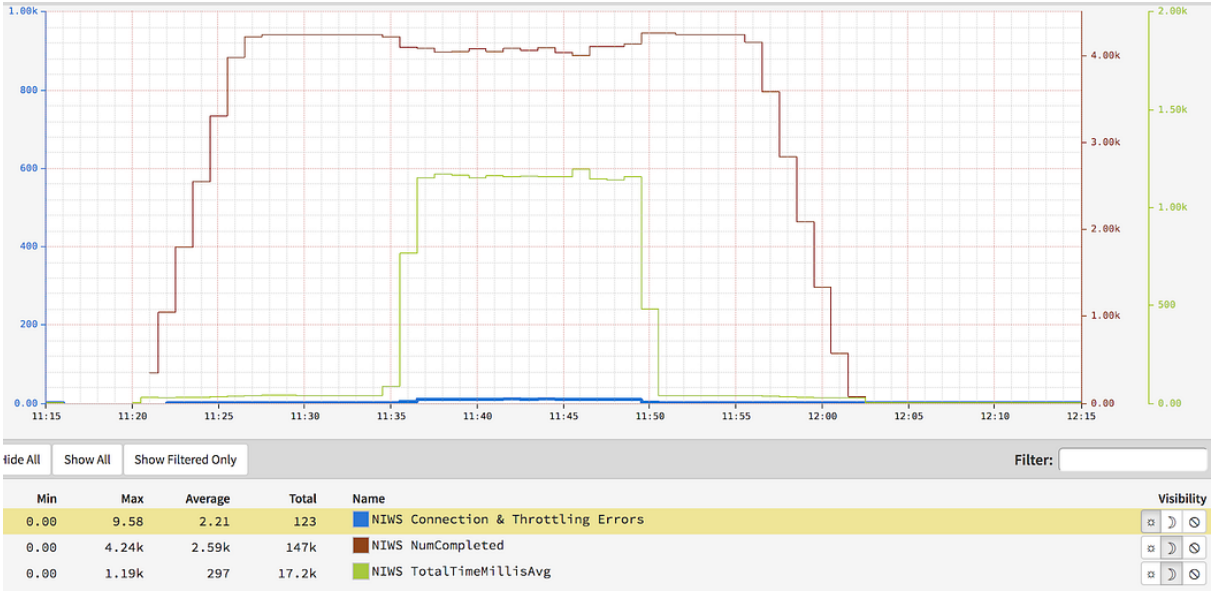
And for reference, looking at how the traffic was split between the faster and slower server groups, you can see that the load-balancer was reducing the proportion to about 15% sent to the slower group (vs an expected 50%).



Distribution of Traffic between the Normal and Slow Clusters

Server-Utilization Disabled

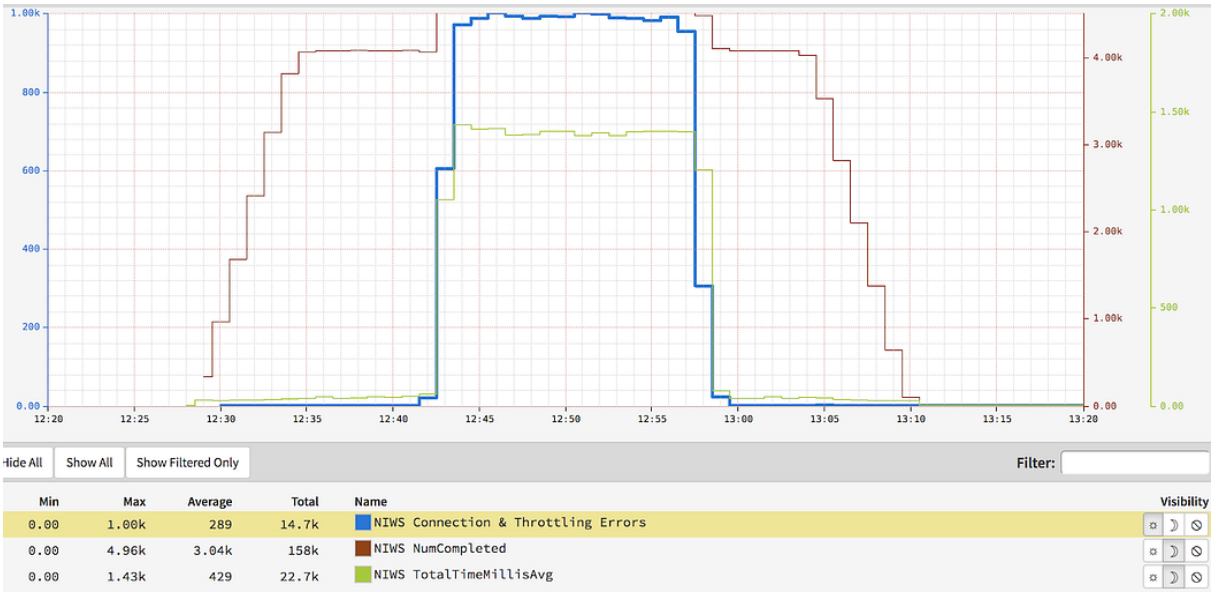
This is again for the new load balancer, but with the server-utilization feature disabled — so therefore only client-side data was used for balancing.



Load-Test with New Load Balancer but Server-Utilization Feature Disabled

Original Implementation

This was for the original round-robin load balancer with it's server blacklisting feature.



Load-Test with Original Load Balancer

Building a Highly Available PostgreSQL Cluster with Patroni, etcd, and HAProxy

medium.com/@dickson.gathima/building-a-highly-available-postgresql-cluster-with-patroni-etcd-and-haproxy-1fd465e2c17f

Dickson Gathima March 14, 2025

Achieving high availability in PostgreSQL requires the right combination of tools and architecture.

I'll walk you through the step-by-step process of setting up a **highly available PostgreSQL cluster** using **Patroni** for automated failover, **etcd** for distributed configuration management, and **HAProxy** for efficient load balancing.

Name	IP Address	Role
nodeAPP	192.168.174.153	etcd + Haproxy
nodeOne	192.168.174.154	Postgresql + Patroni
nodeTwo	192.168.174.155	Postgresql + Patroni
nodeThree	192.168.174.156	Postgresql + Patroni

We'll be setting up a four-node cluster with the following roles:

- nodeAPP (192.168.174.153) — Hosts etcd for distributed coordination and HAProxy for load balancing.
- nodeOne (192.168.174.154) — Runs PostgreSQL with Patroni for high availability management.
- nodeTwo (192.168.174.155) — Runs PostgreSQL with Patroni.
- nodeThree (192.168.174.156) — Runs PostgreSQL with Patroni.

This setup ensures that in the event of a primary node failure, Patroni will automatically promote one of the replicas, and HAProxy will reroute traffic accordingly — keeping the database cluster resilient and accessible.

Step By Step

PostgreSQL

Install PostgreSQL on nodes (nodeOne, nodeTwo and nodeThree)

```
sudo apt updatecurl -fsSL https://www.postgresql.org/media/keys/ACCC4CF8.asc |
sudo gpg --dearmor -o /etc/apt/trusted.gpg.d/postgresql.gpg  | sudo
/etc/apt/sources.list.d/pgdg.listsudo apt install -y postgresql-16 postgresql-
contribsudo systemctl stop postgresql -s /usr/lib/postgresql/16/bin/* /usr/sbin/
```

Patroni Setup

Install and configure patroni nodes (nodeOne, nodeTwo and nodeThree) . To install patroni on Ubuntu 24.04, follow these steps:

```
sudo apt install y python3pip python3dev libpqdev python3venvapt install
python3pip python3dev libpqdev ypip3 install pip install pythonetcd pip install
psycopg2apt install patroni patronietcd
```

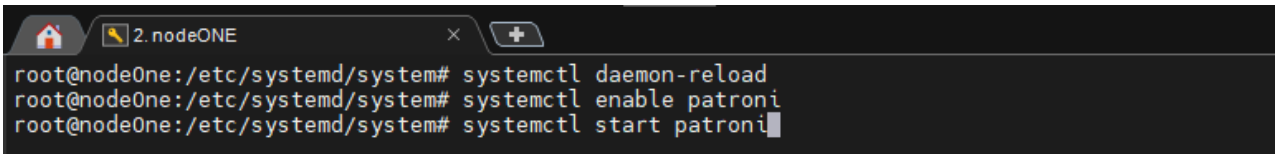
On each node above, edit the /etc/patroni.yml file and add the following lines.

Change the lines starting with ‘listen’ and ‘connect_address’ to correspond to the IP address of the current node.

Create a service for patroni on /etc/systemd/system/patroni.service and add the content below.

```
=Patroni Orchestration=syslog.target
network.target=simple=postgres=postgres=/usr/local/bin/patroni
/etc/patroni.yml=process===multi-user.target
```

Reload the daemon and enable the service.



ETCD Setup

Install etcd on nodeAPP.

```
sudo apt install -y etcd
```

Edit /etc/default/etcd and add the following lines.

```
=    =    =    =    =    =    =  =
```

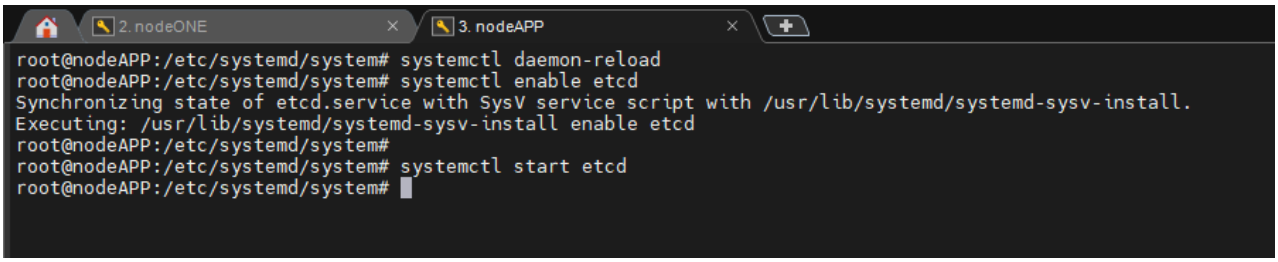
Create and enable a service file at /etc/systemd/system/etcd.service to ensure that etcd starts automatically upon system reboot.

```
=multi-user.target=etcd2.service

=etcd - highly-available key value
store=https://etcd.io/docs=man:etcd=network.target=network-line.target


=DAEMON_ARGS==ETCD_NAME=%H=ETCD_DATA_DIR=/var/lib/etcd/default=-/etc/default/%p=not
 =-abnormal=
```

Reload the daemon and enable the service.



HAProxy setup

Use the following command to install the HAProxy package along with its necessary dependencies.

```
apt- install haproxy
```

Edit the HAProxy configuration file to listen to the PostgreSQL nodes on port 5432, then restart the HAProxy service to apply the changes.

```
global      maxconn defaults      log global      mode tcp      retries
timeout client 30m      timeout connect 4s      timeout server 30m      timeout
check   5slisten stats      mode http      bind *      stats enable      stats uri
/listen postgres      bind *      option httpchk      http-check expect status
default-server inter 3s fall  rise  on-marked-down shutdown-sessions      server
nodeOne . maxconn    check   port       server nodeTwo . maxconn    check   port
server nodeThree . maxconn    check   port rootrootroot
```

```
rootrootrootSynchronizing state of haproxy.service with SysV service script with
/usr/lib/systemd/systemd-sysv-install. /usr/lib/systemd/systemd-sysv-install
enable haproxyrootrootroot
```

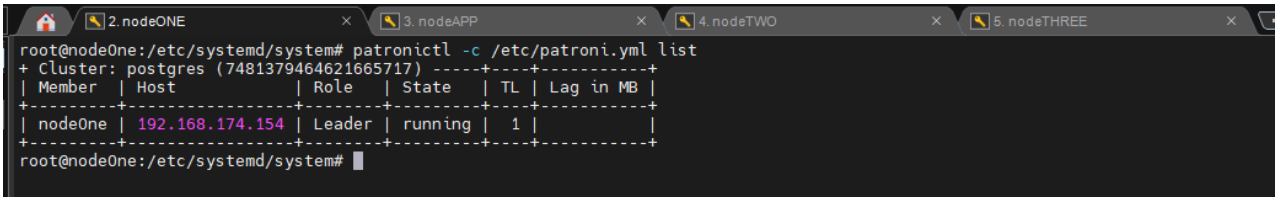
Finally

Start Patroni on NodeOne to initiate it as the leader node.

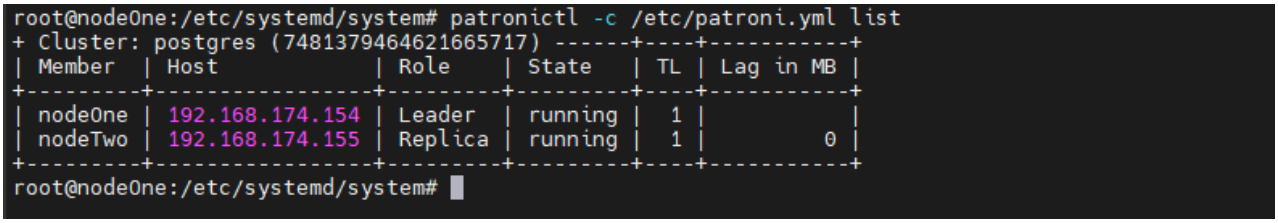
```
                                                  Mar   nodeOne patroni[]: --   [] :
database system is ready to accept connectionsMar   nodeOne patroni[]: -- , :
postmaster pid=Mar   nodeOne patroni[]: . - accepting connectionsMar   nodeOne
patroni[]: . - accepting connectionsMar   nodeOne patroni[]: -- , : establishing a
new patroni heartbeat connection to postgresMar   nodeOne patroni[]: -- , :
running post_bootstrapMar   nodeOne patroni[]: -- , : User creation is  be
supported starting from v4.. Please use  script to create users.Mar   nodeOne
patroni[]: -- , : Could  activate Linux watchdog  Can/dev/watchdog

rootrootroot● patroni.service - Patroni Orchestration loaded (.service; enabled;
enabled) active (running) since Thu --; 5s ago   Main :  (patroni) ().8M (.4M):
.870s /system.slice/patroni.service             ├─ /usr/bin/python3
/usr/local/bin/patroni /etc/patroni.yml             ├─ postgres -D
/var/lib/postgresql//main --config-file=/main/postgresql.conf --listen_addresses=.
--port= --cluster_name=postgres --wal_level=repl>             ├─            ├─
├─            ├─            ├─            ├─          └─
```

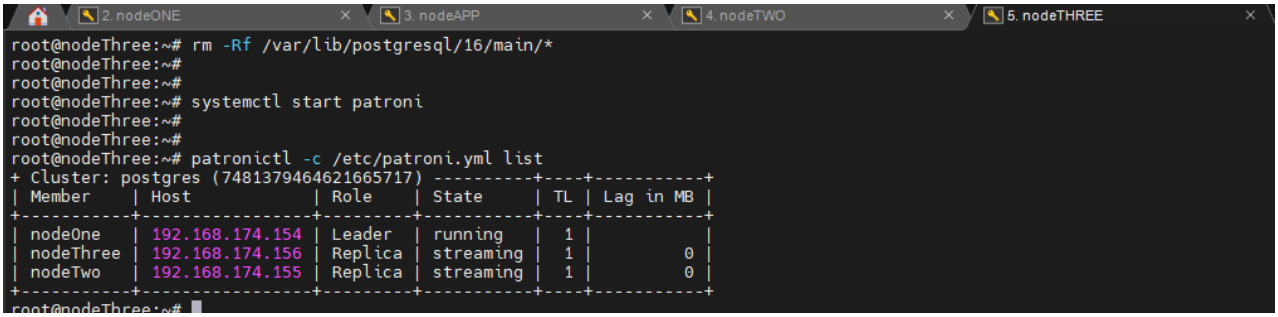
NodeApp is now operating as the leader. To view the nodes and their roles, run the command “patronictl -c /etc/patroni.yml list “ . Note: This command can be executed from any Patroni node.



Now start patroni on nodeTwo and verify.



Finally start patroni on nodeThree and verify.



At this point we have 3 nodes (a leader and two replicas).

Connecting to the Database using HAproxy

To connect to the High Availability (HA) PostgreSQL cluster, use **HAProxy** on *nodeApp* as the entry point. HAProxy acts as a load balancer and traffic router, ensuring that connections are directed to the current primary node in the cluster.

Since HAProxy is configured to listen on **port 5000**, you can establish a connection to the cluster using this port. This setup ensures that applications or clients don't need to be aware of which PostgreSQL node is currently the primary, as HAProxy manages traffic routing transparently.

```
root@nodeAPP:~#
root@nodeAPP:~#
root@nodeAPP:~#
root@nodeAPP:~# psql -h 192.168.174.153 -p 5000 -d postgres -U postgres
Password for user postgres:
psql (16.8 (Ubuntu 16.8-0ubuntu0.24.04.1))
Type "help" for help.

postgres=#
postgres=#
postgres=# \l
                                                  List of databases
   Name    |  Owner   | Encoding | Locale Provider |   Collate   |    Ctype    | ICU Locale | ICU Rules |   Access privileges
-----------+----------+----------+-----------------+-------------+-------------+------------+-----------+-----------------------
 postgres  | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |            |           |
 template0 | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |            |           | =c/postgres          +
           |          |          |                 |             |             |            |           | postgres=CTc/postgres
 template1 | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |            |           | =c/postgres          +
           |          |          |                 |             |             |            |           | postgres=CTc/postgres
(3 rows)

postgres=#
```

Testing Failover

To test the failover mechanism, we will simulate a failure by restarting the **Patroni** service on the current leader node. This approach mimics a real-world scenario where the leader becomes unavailable due to a crash or maintenance activity.

Steps to Simulate Failover

1. First, confirm which node is currently acting as the leader by running the following command from any Patroni node:

```
root@nodeOne:~# patronictl -c /etc/patroni.yml list
+ Cluster: postgres (7481379464621665717) ----------+----+-----------+
| Member    | Host             | Role    | State     | TL | Lag in MB |
+-----------+------------------+---------+-----------+----+-----------+
| nodeOne   | 192.168.174.154 | Leader  | running   |  1 |           |
| nodeThree | 192.168.174.156 | Replica | streaming |  1 |         0 |
| nodeTwo   | 192.168.174.155 | Replica | streaming |  1 |         0 |
+-----------+------------------+---------+-----------+----+-----------+
root@nodeOne:~#
```

2. Restart Patroni on the Leader Node

Once the leader node is identified, restart the Patroni service to simulate its failure:

```
root@nodeOne:~#
root@nodeOne:~# systemctl restart patroni
root@nodeOne:~#
```

3. Observe the Failover Process

Patroni, in coordination with etcd, will detect the leader's unavailability. An automatic failover will occur, promoting one of the replica nodes to become the new leader.

4. Verify the New Leader
After a few moments, check the cluster status again to confirm which node has been promoted:

```
root@nodeOne:~#
root@nodeOne:~# systemctl restart patroni
root@nodeOne:~#
root@nodeOne:~#
root@nodeOne:~# patronictl -c /etc/patroni.yml list
+ Cluster: postgres (7481379464621665717) ----------+----+-----------+
| Member    | Host             | Role    | State     | TL | Lag in MB |
+-----------+------------------+---------+-----------+----+-----------+
| nodeOne   | 192.168.174.154 | Replica | streaming |  2 |         0 |
| nodeThree | 192.168.174.156 | Leader  | running   |  2 |           |
| nodeTwo   | 192.168.174.155 | Replica | streaming |  2 |         0 |
+-----------+------------------+---------+-----------+----+-----------+
root@nodeOne:~#
```

5. Test Database Connectivity
Ensure that client connections through HAProxy are still functioning correctly, confirming that the failover process was successful.

Conclusion of the High Availability (HA) Setup

Setting up a **highly available PostgreSQL cluster** with **Patroni, etcd, and HAProxy** provides a robust and resilient architecture that ensures continuous database operations, even in the event of node failures.

Here's why this setup is effective:

1. ensures that the cluster can automatically promote a replica to primary when the current primary node fails. It simplifies replication management and eliminates manual intervention during failures.
2. acts as the , maintaining cluster state and ensuring consistent coordination across nodes. It helps Patroni manage leadership elections and maintain cluster health.
3. acts as a single point of entry for client connections, intelligently routing traffic to the primary node for write operations and optionally to replicas for read operations.
4. Clients only need to connect to HAProxy, removing the complexity of identifying the current primary node.

Best Practices for Postgres Database Replication

medium.com/timescale/best-practices-for-postgres-database-replication-b5ed69caf96d

Team Timescale April 24, 2025

Timescale is the cloud database engineered for performance and resource efficiency, even with large workloads—resulting in speed, scale, and savings.

Scaling your database can be daunting, but with Postgres replicas, it doesn't have to be. Database replication ensures high availability and helps keep your applications running smoothly, even under the strain of unexpected traffic spikes or server failures.

While you can maintain your replica on-site in the same data center, ideally, the replica would be in a geographically distinct location in case of a critical outage or disaster.

A database replica is not only for disaster recovery, though. While it is the best way to ensure you can keep during a data center outage, replica nodes are also great for scaling and splitting workloads across multiple instances or bringing data closer to your users.

We've written about database replicas before. At the core, a database replica is a copy of your database. It can be a read replica, a hot standby, or a testing and performance node. When configuring replication, you designate one node as the primary and the others as the replicas. The write-ahead logs (WAL) maintain synchronicity by recording all changes (Inserts, Updates, and Deletes) made to the database.

Replicas can improve disaster resilience by keeping an entire copy of your database in a separate data center or availability zone. During an outage, the administrator can failover to the replica and continue operations. PostgreSQL doesn't offer native automatic failover, so this is a critical use. Timescale can help offload this responsibility in various ways depending on the failure scenario.

Another excellent use case for replicas is splitting the database system's workload. For example, using a replica for all read traffic reduces the potential impact on the application if there is a sudden influx of update operations like inserting or backfilling large amounts of data.

Ultimately, this benefits end users by increasing reliability. There is no distance limitation for replicas, so you can offer users improved performance via a read replica no matter what continent they are on.

An often-forgotten use for database replicas is leveraging them as testing environments. Maintaining multiple different datasets depending on the environment can be challenging for organizations. It might be easier to use a replica of the production environment database to benchmark load and performance changes to the application.

Replicas in non-production environments can also be great for tuning query performance in a safe environment before merging the changes to production.

How PostgreSQL Replication Works

PostgreSQL replication is a powerful feature that enables real-time data copying from one database instance (the primary server) to one or more standby servers. This process ensures high availability, fault tolerance, and load balancing in production systems.

Let's break down the key elements of PostgreSQL replication, focusing on the WAL and streaming replication, which together form the backbone of this architecture.

WAL streaming and replication

At the core of PostgreSQL replication is the write-ahead log. The WAL is a crucial mechanism that records changes to the database before they are written to disk. This ensures data consistency and durability, as it allows PostgreSQL to recover from crashes by replaying the log. The same WAL logs are used to ship changes to replica servers.

In WAL streaming replication, the primary server continuously sends chunks of its WAL to one or more standby servers, which then apply these logs to their own data files. This creates an almost real-time copy of the primary database. The standby server is in a read-only mode, capable of serving read queries (read scaling) while it replicates the data changes.

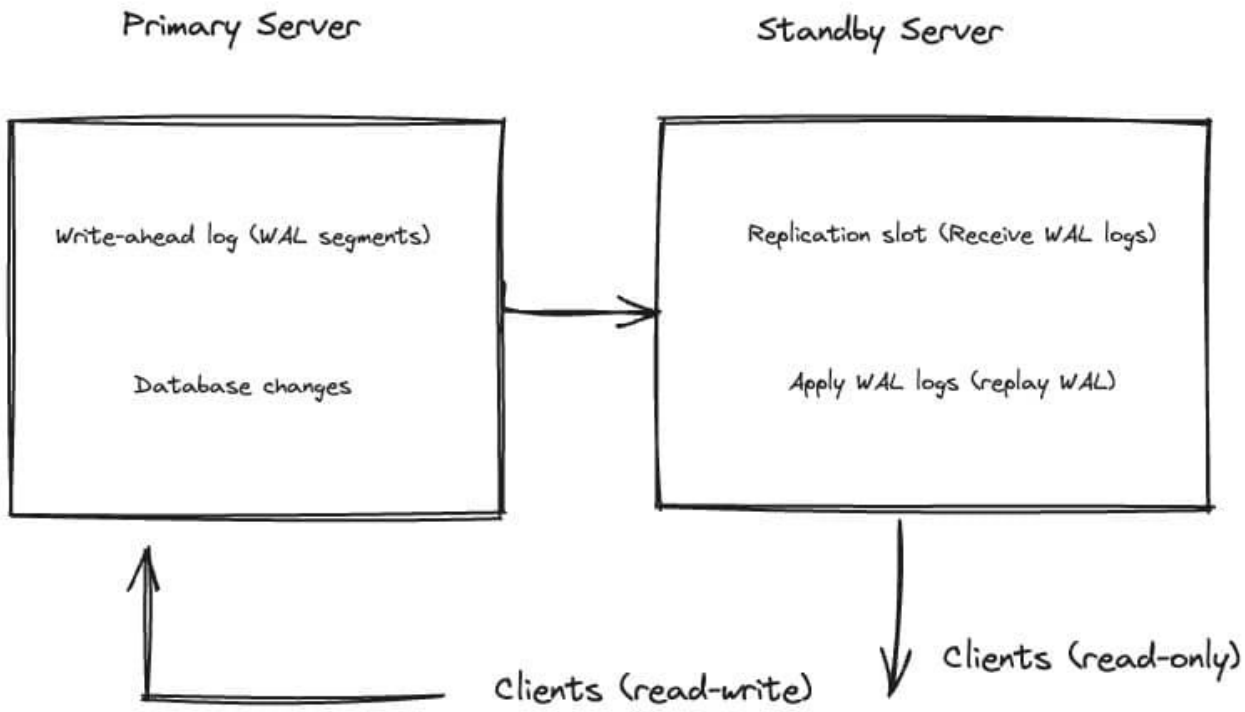
How the write-ahead log works for replication

Here's a simplified breakdown of the WAL's role in replication:

1. Transaction record: Whenever a transaction occurs (e.g., an insert, update, or delete), PostgreSQL writes this transaction to the WAL before committing it to the database files. This ensures that even if the server crashes, the transaction can be recovered from the log.
2. WAL shipping: The WAL segments are sent to the standby servers, either as files or via streaming in small chunks. This process is asynchronous by default but can be configured for synchronous replication to ensure no data is lost during a failure.
3. WAL replay: The standby server receives the WAL segments and applies them to its own data files, keeping its state in sync with the primary. This happens continuously, which minimizes the delay between the primary and standby.

By using this architecture, PostgreSQL ensures data durability and enables fault tolerance. If the primary server fails, the standby can take over with minimal downtime.

Here's a simple diagram illustrating how PostgreSQL WAL streaming replication works:



In this setup:

- The handles both read and write operations, sending the WAL logs to the standby server.
- The receives and replays the WAL logs, keeping its data synchronized with the primary.
- Clients can connect to the standby for read-only queries, helping distribute the load and improve performance.

This basic replication setup ensures that in the event of a failure, the standby server can quickly be promoted to act as the new primary, reducing downtime and data loss risks.

Types of Postgres Database Replicas

There are two types of replication in PostgreSQL: logical and physical. Both have pros and cons, as well as best practices. Logical replication copies the data in the database and the changes that happen and sends them to the replica to be applied. Physical replication operates by sending complete WAL files to the replica or streaming the changes directly to the replica using the changes recorded in the WAL as the source.

As mentioned earlier, there's also the concept of a "hot standby" or read replica. The read replica is great for running read-only queries while maintaining a database that can be a failover node in an emergency.

Logical replicas copy the entire database structure and then make each change that happens on the source database to the replica. Logical replication is great for testing major version upgrades or limiting users' access to specific subsets of data.

Physical replicas are precisely what they sound like: a copy of the entire database on another physical machine. As noted, this is typically accomplished by sending WAL files or streaming changes to the destination database. These replicas are excellent choices for offsite backups and read replicas. Since the changes can be replicated at the disk level, you can be sure that the changes that happen to each database will be the same.

Whether you use physical or logical replication, you can choose between asynchronous, synchronous write, or synchronous apply replication. Depending on the risk level, you may choose a different replication mode. For example, you only need eventual read consistency but need to support a high number of writes. In that case, you may choose asynchronous replication but understand that mode carries a level of data loss due to the nature of how the replication is done.

Here's a summary of the different replication options:

1. Asynchronous replication

Description:

> In asynchronous replication, the primary server does not wait for confirmation from the standby server before committing a transaction and returning success to the client. WAL records are sent to the standby server, but there is no guarantee they have been received or applied by the standby at the time of commit.

Risks:

- In the event of a primary server crash, any transactions that were committed on the primary but not yet replicated to the standby could be lost.
- While the standby remains in sync most of the time, there's no guarantee of immediate synchronization, which could lead to slight data inconsistencies between the primary and standby during a failover.

Use Case: Suitable for systems where high performance is prioritized over strict durability, and some level of data loss in case of failure is acceptable.

2. Synchronous write replication

Description:

> In synchronous write replication, the primary server waits until the WAL records have been sent to and written to disk (but not necessarily applied) by the standby server before confirming the transaction to the client. The transaction is guaranteed to be to the standby's WAL.

Risks:

- If the primary fails, transactions that were confirmed are already on the standby's disk, so there is no risk of losing confirmed transactions.
- Waiting for confirmation from the standby can increase latency, especially if the network between primary and standby has high latency or low bandwidth.
- Even though the WAL is written on the standby, it may not have been applied yet, meaning the standby might not be immediately usable for a failover until the logs are replayed.

Use Case: Suitable for systems where ensuring no data loss is critical, but some delay in failover readiness is acceptable.

3. Synchronous apply replication

Description:

> In synchronous apply replication, the primary server waits for the WAL records to be both and (replayed to the database files) on the standby server before confirming the transaction to the client. This ensures that the standby is completely synchronized with the primary at the time of commit.

Risks:

- There is no risk of losing any confirmed transactions because both the primary and standby have applied the transaction by the time the commit is confirmed.
- This mode introduces the most latency, as the primary waits for the standby to not only receive and write the WAL logs but also apply them to its database files. This adds extra delay to the commit process, especially in cases where the standby is slower or under heavy load.
- Since the standby has already applied all committed transactions, it is ready to take over immediately in the event of a failover.

Use Case: Suitable for mission-critical systems where both data durability and immediate failover readiness are required, and performance impact is secondary to consistency and high availability.

Check out our blog on streaming replication if you need help choosing between these different modes.

Do’s and Don’ts of Database Replication

Replication can be complex, but fortunately, we can share some best practices and pitfalls to avoid reducing the difficulty of managing the process.

Best practices for replication

When setting up replication for PostgreSQL, it’s crucial to follow best practices to ensure high availability, data integrity, and optimal performance.

Failover replicas

One of the most important aspects is implementing an automatic failover system. Failover protocols are essential for maintaining uptime during emergencies, such as hardware failures or network outages.

By having a robust failover mechanism in place, you can minimize downtime and ensure that your application remains operational despite unexpected issues. Additionally, combining failover with regular backups provides a two-layer protection system, allowing you to recover from data loss or corruption. Failover allows for stability during planned downtime, such as software updates or maintenance tasks.

It’s worth noting that PostgreSQL does not have a built-in failover tool. However, several third-party solutions are available to help you set up and manage failover in your PostgreSQL environment. Timescale, for example, offers a replication and failover implementation built on top of PostgreSQL’s native replication capabilities, providing a seamless and reliable solution for high availability.

Choosing a replication mode

Another critical consideration is selecting the correct replication mode for your specific use case. Different replication modes offer varying levels of efficiency and consistency, and selecting the appropriate mode can significantly impact your system’s performance.

For example, if your primary focus is write efficiency, you may opt for an asynchronous replication mode. This mode allows the primary server to commit transactions without waiting for the replicas to acknowledge receipt, resulting in faster write performance. On the other hand, if you require a balance between write performance and data consistency,

synchronous write mode may be more suitable. In this mode, the primary server waits for at least one replica to acknowledge receipt before committing the transaction, ensuring consistent data across nodes.

Synchronous apply mode is the best choice for applications that demand maximum read consistency. In this mode, the primary server waits for all replicas to apply the transaction before considering it committed, guaranteeing that all nodes have the same data state. It's important to note that you can change the replication mode on a per-transaction basis, allowing you to fine-tune your replication setup based on the specific requirements of each operation.

Setting replication parameters

When optimizing your replication setup, it's essential to understand and adjust key replication parameters. These parameters can significantly impact the performance and behavior of your replicas, so you should conduct in-depth testing before deploying any of them to production. Some crucial parameters to consider include `max_wal_size`, `hot_standby`, `max_wal_senders`, `wal_keep_size`, and `hot_standby_feedback`.

The `max_wal_size` parameter determines the maximum size of the WAL files before automatic WAL file recycling occurs. Adjusting this value can affect the storage space and time needed for crash recovery.

The `hot_standby` parameter enables read-only queries on replica servers, allowing you to offload read workloads from the primary server. `max_wal_senders` sets the maximum number of concurrent connections from replicas while `wal_keep_size` specifies the minimum size of WAL files to retain for replica servers. Increasing these values can support more replicas but may consume more resources on the primary server. Finally, `hot_standby_feedback` allows replicas to send feedback to the primary server about their current state, helping to avoid query cancellations due to conflicts.

Monitoring replicas

Monitoring your replicas is crucial to ensure they perform optimally and that emergency protocols are in place. Effective monitoring helps you identify and resolve issues before they impact your application's availability or performance. PostgreSQL provides various built-in monitoring tools and commands.

For example, the `EXPLAIN` command can help you analyze query performance by providing insights into the query execution plan. The `pg_stat_activity` view offers information about the current activity in the database, including running queries and their associated resources.

In addition to built-in tools, several third-party monitoring solutions are available. These tools often provide more advanced features and visualizations, making monitoring and troubleshooting your replication setup easier. Timescale offers monitoring dashboards

that comprehensively overview your database's health, performance, and replication status. These dashboards can help you quickly identify bottlenecks, detect anomalies, and make informed decisions about scaling and optimization.

Replication pitfalls

When implementing replication, there are several pitfalls to be aware of that can impact the performance and consistency of your replicas.

Heavy write loads

One common issue is the impact of heavy write loads on the replication process. When the primary server experiences a high concentration of write queries, it can slow down the replicas as they struggle to keep up with the incoming changes.

In such cases, the replicas may either become desynchronized, leading to inconsistent data, or wait for the WAL replication to catch up, resulting in increased latency. To mitigate this issue, breaking up large write queries into smaller batches is the way to go, allowing the replicas to process the changes more efficiently and maintain better synchronization with the primary server.

Excessive locks

Another potential pitfall is exclusive locks on source tables. When a query acquires a `ACCESS EXCLUSIVE` lock on a table in the primary server, the replicas must wait until the lock is released before they can replay the changes. This locking can lead to significant delays in the replication process, especially if the lock lasts for an extended period.

To identify and address such delays, you can use the `pg_locks` view to monitor lock activity and adjust your queries accordingly. You can prevent replication delays and maintain a more responsive replica environment by minimizing exclusive locks and releasing them promptly.

Read desynchronization

Read replication desynchronization is another common pitfall, mainly when using read replicas. If a read replica is not configured correctly or experiences frequent delays, it may provide the application with stale or inconsistent data. Such inconsistencies can lead to incorrect query results and a poor user experience.

You can avoid this issue by configuring your read replicas correctly. If your application requires near-real-time data consistency, there may be better choices than a read replica. In such cases, exploring alternative replication strategies or adjusting your application's tolerance for slightly stale data may be necessary.

How Timescale Uses Database Replication

Timescale offers a range of tools and features that simplify database replication and enhance the overall experience. With Timescale, creating replicas is a straightforward process, thanks to intuitive tools and documentation. We have a comprehensive guide on PostgreSQL database replication that walks users through setting up and managing replicas. As mentioned, Timescale offers native failover capabilities, ensuring high availability and minimizing downtime in the event of a primary server failure.

Monitoring replicas is crucial for maintaining a healthy and performant replication setup. Timescale provides a user-friendly replica monitoring interface that allows users to track their replicas' status and performance easily. Metrics on the replica, such as CPU, memory utilization, replication lag, and more, are easily accessible.

Timescale also leverages logical replication to facilitate seamless platform transitions and migrations. Using logical replication allows users to replicate data between different database versions, making it an ideal tool for migrating databases with minimal downtime. This blog post on migrating a terabyte-scale PostgreSQL database to Timescale demonstrates how easy it is to migrate without downtime.

Try it for free today to see how easy replication is with Timescale.

Written by , and originally published here.

How to implement Always on Availability Groups in SQL Server 2019 on Windows?

rafaelrampineli.medium.com/how-to-implement-always-on-availability-groups-in-sql-server-2019-on-windows-11f6fb8aad5f

Rafael Rampineli August 20, 2024

Implementing Always On Availability Groups in SQL Server 2019 involves several steps. This feature provides high availability and disaster recovery by allowing multiple copies of a database to be synchronized and available across different servers. Let's setting up using a step-by-step guide below:

1. : Ensure you have SQL Server 2019 Enterprise or Standard edition. Always On Availability Groups are not available in the SQL Server Express or Web editions.
2. : The servers must be running a supported version of Windows Server (e.g., Windows Server 2016 or later).
3. : All servers should be part of the same Active Directory domain.
4. : Install and configure the Failover Cluster feature on all nodes.

Step 1: Configure Windows Server Failover Clustering (WSFC)

Install Failover Clustering Feature:

- Open Server Manager.
- Go to `Manage` -> `Add Roles and Features`.

- Follow the wizard to install the `Failover Clustering` feature on all servers that will be part of the Always On Availability Group.

Create a Failover Cluster:

- Open the Failover Cluster Manager.
- Click on `Create Cluster` and follow the wizard to create a new cluster. You need to specify the cluster name and IP address.
- Validate the configuration and complete the creation.

Step 2: Configure SQL Server for Always On

Enable Always On Availability Groups:

- Open SQL Server Configuration Manager.
- Go to `SQL Server Services`and right-click on the SQL Server instance and select `Properties`.
- Go to the `Always On High Availability` tab.
- Check the box for `Enable Always On Availability Groups`.
- Specify the Windows Server Failover Cluster (WSFC) instance name and click `OK`.
- Restart the SQL Server service to apply the changes.

Step 3: Create the Availability Group

Create a Database Backup:

- Before you can add a database to an availability group, you need to back up the databases you want to include.
- Perform a full backup and a transaction log backup of each database.

```
 BACKUP LOG [YourDatabaseName]  DISK  N;

BACKUP DATABASE [YourDatabaseName]  DISK  N FORMAT, INIT;
```

Restore Databases on Secondary Instances:

- On each secondary replica server, restore the full backup with the `NORECOVERY` option.
- Restore the transaction log backups with the `NORECOVERY` option.

```
 RESTORE LOG [YourDatabaseName]  DISK  N  NORECOVERY;

RESTORE DATABASE [YourDatabaseName]  DISK  N NORECOVERY;
```

Create an Availability Group:

- Open SQL Server Management Studio (SSMS) and connect to the primary SQL Server instance.

- Right-click on the `Always On High Availability` node and select `New Availability Group Wizard`.
- Follow the wizard:
- : Enter a name for the availability group.
- : Choose the databases you want to add to the group.
- : Add replicas by specifying the secondary servers. Configure settings such as automatic failover, synchronous or asynchronous commit mode, and endpoint URLs.
- : Configure backup preferences and backup settings.
- : Configure the availability group listener (optional but recommended). This provides a virtual network name for applications to connect to.

```
 AVAILABILITY  [YourAGName] DATABASE [YourDatabaseName]REPLICA N  (   ENDPOINT_URL
N,   FAILOVER_MODE  AUTOMATIC,   AVAILABILITY_MODE  SYNCHRONOUS_COMMIT,
BACKUP_PRIORITY  ,   SECONDARY_ROLE (ALLOW_CONNECTIONS  READ_ONLY)),N  (
ENDPOINT_URL  N,   FAILOVER_MODE  AUTOMATIC,   AVAILABILITY_MODE
SYNCHRONOUS_COMMIT,   BACKUP_PRIORITY  ,   SECONDARY_ROLE (ALLOW_CONNECTIONS
READ_ONLY))LISTENER  N( IP ((, )));
```

Review and Create:

- Review the summary of your configuration.
- Click `Next` and then `Finish` to create the availability group.

Step 4: Configure Endpoints and Network

Create Endpoint:

- Ensure that the database mirroring endpoint is created on all replicas.
- You can check and create endpoints using T-SQL:

```
 ENDPOINT [Hadr_endpoint]STATESTARTED TCP (LISTENER_PORT  ) DATA_MIRRORING (
ROLE,  AUTHENTICATION  WINDOWS NEGOTIATE,  ENCRYPTION  REQUIRED);
```

Configure Firewall:

> Ensure that the necessary ports (e.g., 5022) are open on all servers for communication.

Step 5: Test the Availability Group

Failover Testing:

> Use SSMS to manually failover the availability group to verify that everything is working correctly.

Monitor and Verify:

- Use the Always On Dashboard in SSMS to monitor the health and status of the availability group.

- Check the SQL Server logs and Windows Event Viewer for any issues.