Initial Setup

We recycled three 10-year old Intel Atom mini-computers for our experiment but you could use some virtual machines instead: even though you will miss the excitement of unplugging real cables, this can still be simulated with a VM. We installed the server version of Ubuntu 20.04 and configured them to know “each other” by hostname; here’s how the *hosts* file of the first node looked like:

Shell

|  |  |
| --- | --- |
| 1  2  3  4  5 | $ cat /etc/hosts  127.0.0.1 localhost node1  192.168.1.11 node1  192.168.1.12 node2  192.168.1.13 node3 |

etcd

Patroni supports a myriad of systems for [Distribution Configuration Store](https://github.com/zalando/patroni#patroni-a-template-for-postgresql-ha-with-zookeeper-etcd-or-consul) but *etcd* remains a popular choice. We installed the version available from the Ubuntu repository on all three nodes:

Shell

|  |  |
| --- | --- |
| 1 | sudo apt-get install etcd |

It is necessary to initialize the etcd cluster from one of the nodes and we did that from node1 using the following configuration file:

Shell

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | $ cat /etc/default/etcd  ETCD\_NAME=node1  ETCD\_INITIAL\_CLUSTER="node1=http://192.168.1.11:2380"  ETCD\_INITIAL\_CLUSTER\_TOKEN="devops\_token"  ETCD\_INITIAL\_CLUSTER\_STATE="new"  ETCD\_INITIAL\_ADVERTISE\_PEER\_URLS="http://192.168.1.11:2380"  ETCD\_DATA\_DIR="/var/lib/etcd/postgresql"  ETCD\_LISTEN\_PEER\_URLS="http://192.168.1.11:2380"  ETCD\_LISTEN\_CLIENT\_URLS="http://192.168.1.11:2379,http://localhost:2379"  ETCD\_ADVERTISE\_CLIENT\_URLS="http://192.168.1.11:2379" |

Note how ETCD\_INITIAL\_CLUSTER\_STATE is defined with “new”.

We then restarted the service:

Shell

|  |  |
| --- | --- |
| 1 | sudo systemctl restart etcd |

We can then move on to install etcd on node2. The configuration file follows the same structure as that of node1, except that we are adding node2 to an existing cluster so we should indicate the other node(s):

Shell

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | ETCD\_NAME=node2  ETCD\_INITIAL\_CLUSTER="node1=http://192.168.1.11:2380,node2=http://192.168.1.12:2380"  ETCD\_INITIAL\_CLUSTER\_TOKEN="devops\_token"  ETCD\_INITIAL\_CLUSTER\_STATE="existing"  ETCD\_INITIAL\_ADVERTISE\_PEER\_URLS="http://192.168.1.12:2380"  ETCD\_DATA\_DIR="/var/lib/etcd/postgresql"  ETCD\_LISTEN\_PEER\_URLS="http://192.168.1.12:2380"  ETCD\_LISTEN\_CLIENT\_URLS="http://192.168.1.12:2379,http://localhost:2379"  ETCD\_ADVERTISE\_CLIENT\_URLS="http://192.168.1.12:2379" |

Before we restart the service, we need to formally add node2 to the etcd cluster by running the following command on node1:

Shell

|  |  |
| --- | --- |
| 1 | sudo etcdctl member add node2 http://192.168.1.12:2380 |

We can then restart the etcd service on node2:

Shell

|  |  |
| --- | --- |
| 1 | sudo systemctl restart etcd |

The configuration file for node3 looks like this:

Shell

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | ETCD\_NAME=node3  ETCD\_INITIAL\_CLUSTER="node1=http://192.168.1.11:2380,node2=http://192.168.1.12:2380,node3=http://192.168.1.13:2380"  ETCD\_INITIAL\_CLUSTER\_TOKEN="devops\_token"  ETCD\_INITIAL\_CLUSTER\_STATE="existing"  ETCD\_INITIAL\_ADVERTISE\_PEER\_URLS="http://192.168.1.13:2380"  ETCD\_DATA\_DIR="/var/lib/etcd/postgresql"  ETCD\_LISTEN\_PEER\_URLS="http://192.168.1.13:2380"  ETCD\_LISTEN\_CLIENT\_URLS="http://192.168.1.13:2379,http://localhost:2379"  ETCD\_ADVERTISE\_CLIENT\_URLS="http://192.168.1.13:2379" |

Remember we need to add node3 to the cluster by running the following command on node1:

Shell

|  |  |
| --- | --- |
| 1 | sudo etcdctl member add node3 http://192.168.1.13:2380 |

before we can restart the service on node3:

Shell

|  |  |
| --- | --- |
| 1 | sudo systemctl restart etcd |

We can verify the cluster state to confirm it has been deployed successfully by running the following command from any of the nodes:

Shell

|  |  |
| --- | --- |
| 1  2  3  4 | $ sudo etcdctl member list  2ed43136d81039b4: name=node3 peerURLs=http://192.168.1.13:2380 clientURLs=http://192.168.1.13:2379 isLeader=false  d571a1ada5a5afcf: name=node1 peerURLs=http://192.168.1.11:2380 clientURLs=http://192.168.1.11:2379 isLeader=true  ecec6c549ebb23bc: name=node2 peerURLs=http://192.168.1.12:2380 clientURLs=http://192.168.1.12:2379 isLeader=false |

As we can see above, node1 is the leader at this point, which is expected since the etcd cluster has been bootstrapped from it. If you get a different result, check for etcd entries logged to */var/log/syslog* on each node.

Watchdog

Quoting [Patroni’s manual](https://patroni.readthedocs.io/en/latest/watchdog.html?highlight=watchdog" \l "watchdog-support):

*Watchdog devices are software or hardware mechanisms that will reset the whole system when they do not get a keepalive heartbeat within a specified timeframe. This adds an additional layer of fail safe in case usual Patroni split-brain protection mechanisms fail.*

While the use of a watchdog mechanism with Patroni is optional, you shouldn’t really consider deploying a PostgreSQL HA environment in production without it.

For our tests, we used the standard software implementation for watchdog that is shipped with Ubuntu 20.04, a module called *softdog*. Here’s the procedure we used in all three nodes to configure the module to load:

Shell

|  |  |
| --- | --- |
| 1 | sudo sh -c 'echo "softdog" >> /etc/modules' |

Patroni will be the component interacting with the watchdog device. Since Patroni is run by the *postgres* user, we need to either set the permissions of the watchdog device open enough so the *postgres*user can write to it or make the device owned by *postgres* itself, which we consider a safer approach (as it is more restrictive):

Shell

|  |  |
| --- | --- |
| 1 | sudo sh -c 'echo "KERNEL=="watchdog", OWNER="postgres", GROUP="postgres"" >> /etc/udev/rules.d/61-watchdog.rules' |

These two steps looked like all that would be required for watchdog to work but to our surprise, the *softdog* module wasn’t loaded after restarting the servers. After spending quite some time digging around we figured the module was blacklisted by default and there was a strain file with such a directive still lingering around:

Shell

|  |  |
| --- | --- |
| 1  2 | $ grep blacklist /lib/modprobe.d/\* /etc/modprobe.d/\* |grep softdog  /lib/modprobe.d/blacklist\_linux\_5.4.0-72-generic.conf:blacklist softdog |

Editing that file in each of the nodes to remove the line above and restarting the servers did the trick:

Shell

|  |  |
| --- | --- |
| 1  2 | $ lsmod | grep softdog  softdog                16384  0 |

Shell

|  |  |
| --- | --- |
| 1  2  3 | $ ls -l /dev/watchdog\*  crw-rw---- 1 postgres postgres  10, 130 May 21 21:30 /dev/watchdog  crw------- 1 root     root     245,   0 May 21 21:30 /dev/watchdog0 |

PostgreSQL

[*Percona Distribution for PostgreSQL*](https://www.percona.com/postgresql/software/postgresql-distribution) can be easily installed from the Percona Repository in a few easy steps:

Shell

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | sudo apt-get update -y; sudo apt-get install -y wget gnupg2 lsb-release curl  wget https://repo.percona.com/apt/percona-release\_latest.generic\_all.deb  sudo dpkg -i percona-release\_latest.generic\_all.deb  sudo apt-get update  sudo percona-release setup ppg-12  sudo apt-get install percona-postgresql-12 |

An important concept to understand in a PostgreSQL HA environment like this one is that PostgreSQL should not be started automatically by systemd during the server initialization: we should leave it to Patroni to fully manage it, including the process of starting and stopping the server. Thus, we should disable the service:

Shell

|  |  |
| --- | --- |
| 1 | sudo systemctl disable postgresql |

For our tests, we want to start with a fresh new PostgreSQL setup and let Patroni bootstrap the cluster, so we stop the server and remove the data directory that has been created as part of the PostgreSQL installation:

Shell

|  |  |
| --- | --- |
| 1  2 | sudo systemctl stop postgresql  sudo rm -fr /var/lib/postgresql/12/main |

These steps should be repeated in nodes 2 and 3 as well.

Patroni

The Percona Repository also includes a package for Patroni so with it already configured in the nodes we can install Patroni with a simple:

Shell

|  |  |
| --- | --- |
| 1 | sudo apt-get install percona-patroni |

Here’s the configuration file we have used for node1:

Shell



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85 | $ cat /etc/patroni/config.yml  scope: stampede  name: node1    restapi:    listen: 0.0.0.0:8008    connect\_address: node1:8008    etcd:    host: node1:2379    bootstrap:    # this section will be written into Etcd:/<namespace>/<scope>/config after initializing new cluster    dcs:      ttl: 30      loop\_wait: 10      retry\_timeout: 10      maximum\_lag\_on\_failover: 1048576  #    master\_start\_timeout: 300  #    synchronous\_mode: false      postgresql:        use\_pg\_rewind: true        use\_slots: true        parameters:          wal\_level: replica          hot\_standby: "on"          logging\_collector: 'on'          max\_wal\_senders: 5          max\_replication\_slots: 5          wal\_log\_hints: "on"          #archive\_mode: "on"          #archive\_timeout: 600          #archive\_command: "cp -f %p /home/postgres/archived/%f"          #recovery\_conf:          #restore\_command: cp /home/postgres/archived/%f %p      # some desired options for 'initdb'    initdb:  # Note: It needs to be a list (some options need values, others are switches)    - encoding: UTF8    - data-checksums      pg\_hba:  # Add following lines to pg\_hba.conf after running 'initdb'    - host replication replicator 192.168.1.1/24 md5    - host replication replicator 127.0.0.1/32 trust    - host all all 192.168.1.1/24 md5    - host all all 0.0.0.0/0 md5  #  - hostssl all all 0.0.0.0/0 md5      # Additional script to be launched after initial cluster creation (will be passed the connection URL as parameter)  # post\_init: /usr/local/bin/setup\_cluster.sh    # Some additional users users which needs to be created after initializing new cluster    users:      admin:        password: admin        options:          - createrole          - createdb    postgresql:    listen: 0.0.0.0:5432    connect\_address: node1:5432    data\_dir: "/var/lib/postgresql/12/main"    bin\_dir: "/usr/lib/postgresql/12/bin"  #  config\_dir:    pgpass: /tmp/pgpass0    authentication:      replication:        username: replicator        password: vagrant      superuser:        username: postgres        password: vagrant    parameters:      unix\_socket\_directories: '/var/run/postgresql'    watchdog:    mode: required # Allowed values: off, automatic, required    device: /dev/watchdog    safety\_margin: 5    tags:      nofailover: false      noloadbalance: false      clonefrom: false      nosync: false |

With the configuration file in place, and now that we already have the *etcd* cluster up, all that is required is to restart the Patroni service:

Shell

|  |  |
| --- | --- |
| 1 | sudo systemctl restart patroni |

**When Patroni starts, it will take care of initializing PostgreSQL** (because the service is not currently running and the data directory is empty) following the directives in the *bootstrap*section of Patroni’s configuration file. If everything went according to the plan, you should be able to connect to PostgreSQL using the credentials in the configuration file (password is *vagrant*):

Shell

|  |  |
| --- | --- |
| 1  2  3  4  5 | $ psql -U postgres  psql (12.6 (Ubuntu 2:12.6-2.focal))  Type "help" for help.    postgres=# |

Repeat the operation for installing Patroni on nodes 2 and 3: the only difference is that you will need to replace the references to node1 in the configuration file (there are four of them, shown in **bold**) with the respective node name.

You can also check the state of the Patroni cluster we just created with:

Shell

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | $ sudo patronictl -c /etc/patroni/config.yml list  +----------+--------+-------+--------+---------+----+-----------+  | Cluster  | Member |  Host |  Role  |  State  | TL | Lag in MB |  +----------+--------+-------+--------+---------+----+-----------+  | stampede | node1  | node1 | Leader | running |  2 |           |  | stampede | node2  | node2 |        | running |  2 |         0 |  | stampede | node3  | node3 |        | running |  2 |         0 |  +----------+--------+-------+--------+---------+----+-----------+ |

node1 started the Patroni cluster so it was automatically made the leader – and thus the primary/master PostgreSQL server. Nodes 2 and 3 are configured as read replicas (as the *hot\_standby* option was enabled in Patroni’s configuration file).

HAProxy

A common implementation of high availability in a PostgreSQL environment makes use of a proxy: instead of connecting directly to the database server, the application will be connecting to the proxy instead, which will forward the request to PostgreSQL. When [HAproxy](http://www.haproxy.org/) is used for this, it is also possible to route read requests to one or more replicas, for load balancing. However, this is not a transparent process: the application needs to be aware of this and split read-only from read-write traffic itself. With HAproxy, this is done by providing two different ports for the application to connect. We opted for the following setup:

* Writes   →  5000
* Reads   →  5001

HAproxy can be installed as an independent server (and you can have as many as you want) but it can also be installed on the application server or the database server itself – it is a light enough service. For our tests, we planned on using our own Linux workstations (which also run Ubuntu 20.04) to simulate application traffic so we installed HAproxy on them:

Shell

|  |  |
| --- | --- |
| 1 | sudo apt-get install haproxy |

With the software installed, we modified the main configuration file as follows:

Shell

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | $ cat /etc/haproxy/haproxy.cfg  global      maxconn 100    defaults      log    global      mode    tcp      retries 2      timeout client 30m      timeout connect 4s      timeout server 30m      timeout check 5s    listen stats      mode http      bind \*:7000      stats enable      stats uri /    listen primary      bind \*:5000      option httpchk OPTIONS /master      http-check expect status 200      default-server inter 3s fall 3 rise 2 on-marked-down shutdown-sessions      server node1 node1:5432 maxconn 100 check port 8008      server node2 node2:5432 maxconn 100 check port 8008      server node3 node3:5432 maxconn 100 check port 8008    listen standbys      balance roundrobin      bind \*:5001      option httpchk OPTIONS /replica      http-check expect status 200      default-server inter 3s fall 3 rise 2 on-marked-down shutdown-sessions      server node1 node1:5432 maxconn 100 check port 8008      server node2 node2:5432 maxconn 100 check port 8008      server node3 node3:5432 maxconn 100 check port 8008 |

Note there are two sections: *primary*, using port 5000, and *standbys*, using port 5001. All three nodes are included in both sections: that’s because they are all potential candidates to be either primary or secondary. For HAproxy to know which role each node currently has, it will send an HTTP request to port 8008 of the node: Patroni will answer. Patroni provides a built-in REST API support for [health check monitoring](https://patroni.readthedocs.io/en/latest/rest_api.html#health-check-endpoints) that integrates perfectly with HAproxy for this:

Shell

|  |  |
| --- | --- |
| 1  2 | $ curl -s http://node1:8008  {"state": "running", "postmaster\_start\_time": "2021-05-24 14:50:11.707 UTC", "role": "master", "server\_version": 120006, "cluster\_unlocked": false, "xlog": {"location": 25615248}, "timeline": 1, "database\_system\_identifier": "6965869170583425899", "patroni": {"version": "1.6.4", "scope": "stampede"}} |

We configured the *standbys* group to balance read-requests in a round-robin fashion, so each connection request (or reconnection) will alternate between the available replicas. We can test this in practice, let’s save the *postgres* user password in a file to facilitate the process:

Shell

|  |  |
| --- | --- |
| 1  2  3 | echo "localhost:5000:postgres:postgres:vagrant" > ~/.pgpass  echo "localhost:5001:postgres:postgres:vagrant" >> ~/.pgpass  chmod 0600 ~/.pgpass |

We can then execute two read-requests to verify the round-robin mechanism is working as intended:

Shell

|  |  |
| --- | --- |
| 1  2 | $ psql -Upostgres -hlocalhost -p<strong>5001</strong> -t -c "select inet\_server\_addr()"   192.168.1.13 |

Shell

|  |  |
| --- | --- |
| 1  2 | $ psql -Upostgres -hlocalhost -p<strong>5001</strong> -t -c "select inet\_server\_addr()"   192.168.1.12 |

as well as test the writer access:

Shell

|  |  |
| --- | --- |
| 1  2 | $ psql -Upostgres -hlocalhost -p<strong>5000</strong> -t -c "select inet\_server\_addr()"   192.168.1.11 |

You can also check the state of HAproxy by visiting <http://localhost:7000/> on your browser.

Workload

To best simulate a production environment to test our failure scenarios, we wanted to have continuous reads and writes to the database. We could have used a benchmark tool such as Sysbench or Pgbench but we were more interested in observing the switch of source server upon a server failure than load itself. Jobin wrote a simple Python script that is perfect for this, *[HAtester](https://github.com/jobinau/pgscripts/blob/main/patroni/HAtester.py)*. As was the case with HAproxy, we run the script from our Linux workstation. Since it is a Python script, you need to have a PostgreSQL driver for Python installed to execute it:

Shell

|  |  |
| --- | --- |
| 1  2  3 | sudo apt-get install python3-psycopg2  curl -LO https://raw.githubusercontent.com/jobinau/pgscripts/main/patroni/HAtester.py  chmod +x HAtester.py |

Edit the script with the credentials to access the PostgreSQL servers (through HAproxy) if you are using different settings from ours. The only requirement for it to work is to have the target table created beforehand, so first connect to the *postgres*database (unless you are using a different target) in the Primary and run:

Shell

|  |  |
| --- | --- |
| 1 | CREATE TABLE HATEST (TM TIMESTAMP); |

You can then start two different sessions:

1. One for writes:

Shell

|  |  |
| --- | --- |
| 1 | ./HAtester.py 5000 |

1. One for reads:

Shell

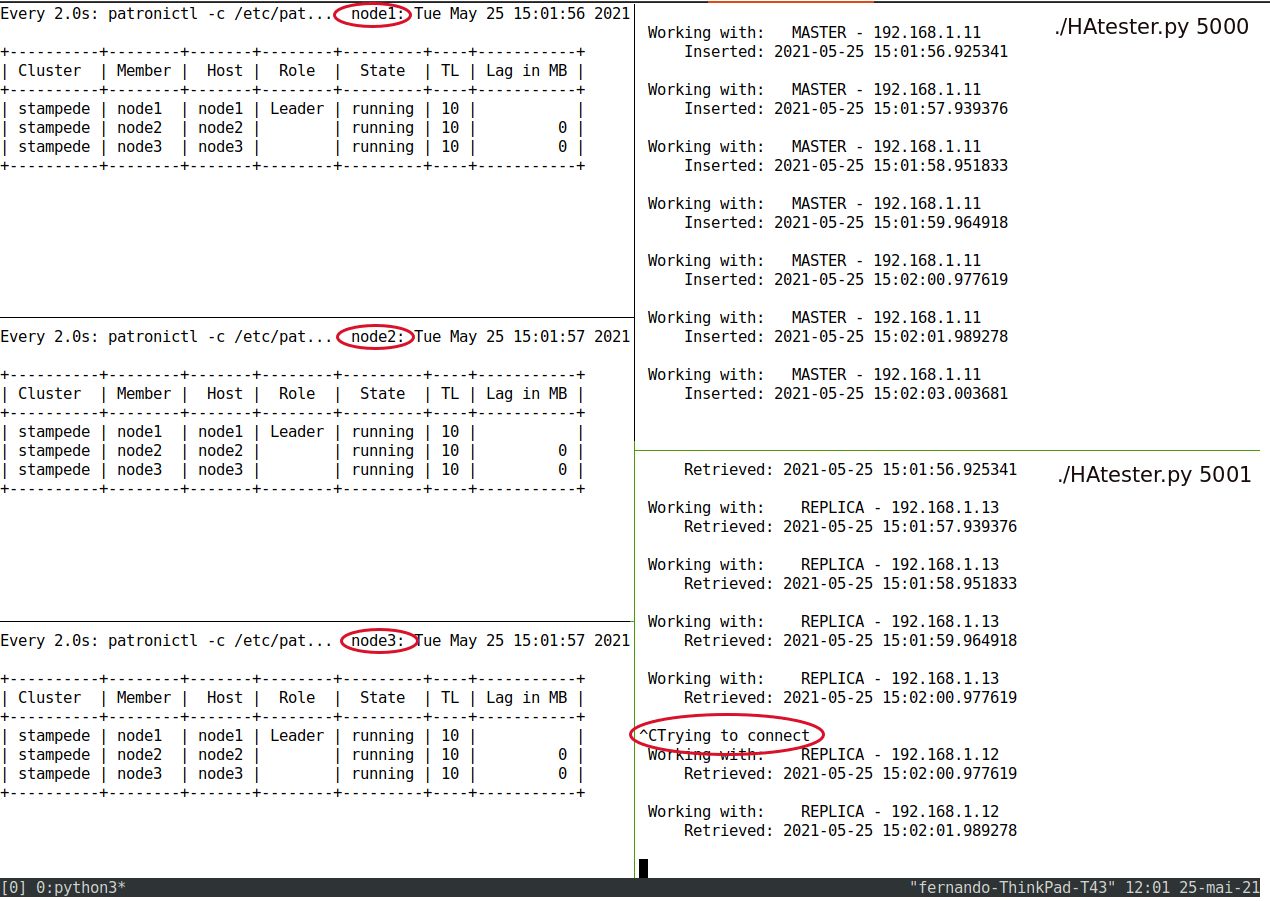
|  |  |
| --- | --- |
| 1 | ./HAtester.py 5001 |

The idea is to observe what happens with database traffic when the environment experiences a failure; that is, how HAproxy will route reads and writes as Patroni adjusts the PostgreSQL cluster. You can continuously monitor Patroni from the point of view of the nodes by opening a session in each of them and running the following command:

Shell

|  |  |
| --- | --- |
| 1 | sudo -u postgres watch patronictl -c /etc/patroni/config.yml list |

To facilitate observability and better follow the changes in real-time, we used the terminal multiplexer [Tmux](https://github.com/tmux/tmux/wiki) to visualize all 5 sessions on the same screen:



* On the left side, we have one session open for each of the 3 nodes, continuously running:

Shell

|  |  |
| --- | --- |
|  | sudo -u postgres watch patronictl -c /etc/patroni/config.yml list |

It’s better to have the Patroni view for each node independently because when you start the failure tests you will lose connection to a part of the cluster.

* On the right side, we are executing the *HAtester.py* script from our workstation:
  + Sending writes through port 5000:

Shell

|  |  |
| --- | --- |
| 1 | ./HAtester.py 5000 |

* + and reads through port 5001:

Shell

|  |  |
| --- | --- |
| 1 | ./HAtester.py 5001 |

A couple of notes on the execution of the HAtester.py script:

* Pressing Ctrl+C will break the connection but the script will reconnect, this time to a different replica (in the case of *reads*) due to having the *Standbys* group on HAproxy configured with round-robin balancing.
* When a switchover or failover takes place and the nodes are re-arranged in the cluster, you may temporarily see writes sent to a node that used to be a replica and was just promoted as primary and reads send to a node that used to be the primary and was demoted as secondary: that’s a limitation of the *HAtester.py* script but “by design”; we favored faster reconnections and minimal checks on the node’s role for demonstration purposes. On a production application, this part ought to be implemented differently.