pg_auto_failover Documentation

Release 1.3.1

Microsoft

Aug 10, 2020

Contents:

CHAPTER 1

Introduction to pg_auto_failover

pg_auto_failover is an extension for PostgreSQL that monitors and manages failover for a postgres clusters. It is optimised for simplicity and correctness.

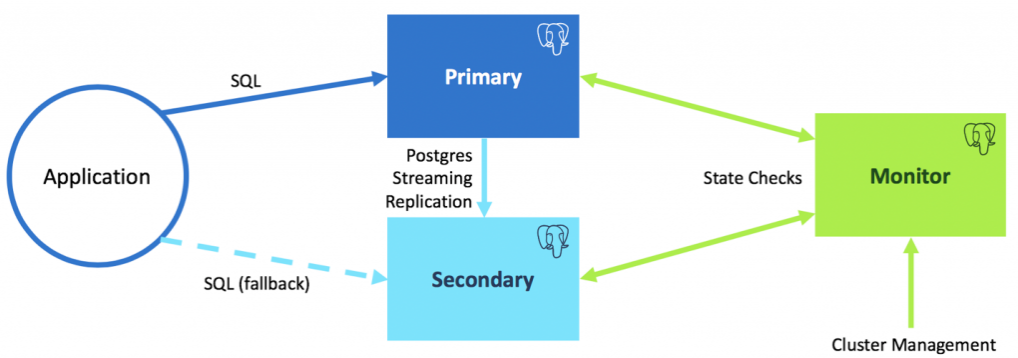

Fig. 1: pg_auto_failover Architecture for a standalone PostgreSQL service

pg_auto_failover implements Business Continuity for your PostgreSQL services. pg_auto_failover implements a single PostgreSQL service using multiple nodes

with automated failover, and automates PostgreSQL maintenance operations in a way that guarantees availability of the service to its users and applications.

To that end, pg_auto_failover uses three nodes (machines, servers) per PostgreSQL service:

- a PostgreSQL primary node,
- a PostgreSQL secondary node, using Synchronous Hot Standby,
- a pg_auto_failover Monitor node that acts both as a witness and an orchestrator.

The pg_auto_failover Monitor implements a state machine and relies on in-core PostgreSQL facilities to deliver HA. For example. when the *secondary* node is detected to be unavailable, or when its lag is reported above a defined threshold (the default is 1 WAL files, or 16MB, see the *pgautofailover.promote_wal_log_threshold* GUC on the pg_auto_failover monitor), then the Monitor removes it from the *synchronous_standby_names* setting on the *primary* node. Until the *secondary* is back to being monitored healthy, failover and switchover operations are not allowed, preventing data loss.

CHAPTER 2

pg_auto_failover Tutorial

In this guide we'll create a primary and secondary Postgres node and set up pg_auto_failover to replicate data between them. We'll simulate failure in the primary node and see how the system smoothly switches (fails over) to the secondary.

For illustration, we'll run our databases on virtual machines in the Azure platform, but the techniques here are relevant to any cloud provider or on-premise network. We'll use four virtual machines: a primary database, a secondary database, a monitor, and an "application." The monitor watches the other nodes' health, manages global state, and assigns nodes their roles.

2.1 Create virtual network

Our database machines need to talk to each other and to the monitor node, so let's create a virtual network.

```
az group create \
    --name ha-demo \
    --location eastus

az network vnet create \
    --resource-group ha-demo \
    --name ha-demo-net \
    --address-prefix 10.0.0.0/16
```

We need to open ports 5432 (Postgres) and 22 (SSH) between the machines, and also give ourselves address from our remote IP. We'll do this with a network security group and a subnet.

```
az network nsg create \
    --resource-group ha-demo \
    --name ha-demo-nsg

az network nsg rule create \
    --resource-group ha-demo \
    --nsg-name ha-demo-nsg \
    --name ha-demo-ssh-and-pg \
    --access allow \
    --protocol Tcp \
    --direction Inbound \
    --priority 100 \
    --source-address-prefixes `curl ifconfig.me` 10.0.1.0/24 \
    --source-port-range "*" \
    --destination-address-prefix "*" \
    --destination-port-ranges 22 5432

az network vnet subnet create \
    --resource-group ha-demo \
    --vnet-name ha-demo-net \
    --name ha-demo-subnet \
    --address-prefixes 10.0.1.0/24 \
    --network-security-group ha-demo-nsg
```

Finally add four virtual machines (ha-demo-a, ha-demo-b, ha-demo-monitor, and ha-demo-app). For speed we background the `az vm create` processes and run them in parallel:

```
# create VMs in parallel
for node in monitor a b app
do
az vm create \
    --resource-group ha-demo \
    --name ha-demo-${node} \
    --vnet-name ha-demo-net \
    --subnet ha-demo-subnet \
    --nsg ha-demo-nsg \
    --public-ip-address ha-demo-${node}-ip \
    --image debian \
```

(continues on next page)

(continued from previous page)

```
    --admin-username ha-admin \
    --generate-ssh-keys &
done
wait
```

To make it easier to SSH into these VMs in future steps, let's make a shell function to retrieve their IP addresses:

```
# run this in your local shell as well

vm_ip () {
  az vm list-ip-addresses -g ha-demo -n ha-demo-$1 -o tsv \
    --query '[] [] .virtualMachine.network.publicIpAddresses[0].ipAddress'
}
```

Let's review what we created so far.

```
az resource list --output table --query \
  "[?resourceGroup=='ha-demo'].{ name: name, flavor: kind, resourceType: type,␣
 ↪region: location }"
```

This shows the following resources:

```
Name                             ResourceType                                          ␣
 ↪          Region
-------------------------------  ------------------------------------------------
 ↪------  --------
ha-demo-a                        Microsoft.Compute/virtualMachines                     ␣
 ↪          eastus
ha-demo-app                      Microsoft.Compute/virtualMachines                     ␣
 ↪          eastus
ha-demo-b                        Microsoft.Compute/virtualMachines                     ␣
 ↪          eastus
ha-demo-monitor                  Microsoft.Compute/virtualMachines                     ␣
 ↪          eastus
ha-demo-appVMNic                 Microsoft.Network/networkInterfaces                   ␣
 ↪          eastus
ha-demo-aVMNic                   Microsoft.Network/networkInterfaces                   ␣
 ↪          eastus
ha-demo-bVMNic                   Microsoft.Network/networkInterfaces                   ␣
 ↪          eastus
ha-demo-monitorVMNic             Microsoft.Network/networkInterfaces                   ␣
 ↪          eastus
ha-demo-nsg                      Microsoft.Network/networkSecurityGroups               ␣
 ↪          eastus
ha-demo-a-ip                     Microsoft.Network/publicIPAddresses                   ␣
 ↪          eastus
ha-demo-app-ip                   Microsoft.Network/publicIPAddresses                   ␣
 ↪          eastus
ha-demo-b-ip                     Microsoft.Network/publicIPAddresses                   ␣
 ↪          eastus
```

(continues on next page)

(continued from previous page)

```
ha-demo-monitor-ip                Microsoft.Network/publicIPAddresses
 ↪          eastus
ha-demo-net                       Microsoft.Network/virtualNetworks
 ↪          eastus
```

2.2 Install the "pg_autoctl" executable

This guide uses Debian Linux, but similar steps will work on other distributions. All that differs are the packages and paths. See *Installing pg_auto_failover*.

The pg_auto_failover system is distributed as a single `pg_autoctl` binary with subcommands to initialize and manage a replicated PostgreSQL service. We'll install the binary with the operating system package manager on all nodes. It will help us run and observe PostgreSQL.

```
for node in monitor a b app
do
az vm run-command invoke \
   --resource-group ha-demo \
   --name ha-demo-${node} \
   --command-id RunShellScript \
   --scripts \
      "curl https://install.citusdata.com/community/deb.sh | sudo bash" \
      "sudo apt-get install -q -y postgresql-common" \
      "echo 'create_main_cluster = false' | sudo tee -a /etc/postgresql-common/
↪createcluster.conf" \
      "sudo apt-get install -q -y postgresql-11-auto-failover-1.3" \
      "sudo usermod -a -G postgres ha-admin" &
done
wait
```

2.3 Run a monitor

The pg_auto_failover monitor is the first component to run. It periodically attempts to contact the other nodes and watches their health. It also maintains global state that "keepers" on each node consult to determine their own roles in the system.

```
# on the monitor virtual machine
```

(continues on next page)

(continued from previous page)

```
ssh -l ha-admin `vm_ip monitor` -- \
  pg_autoctl create monitor \
    --auth trust \
    --ssl-self-signed \
    --pgdata monitor \
    --pgctl /usr/lib/postgresql/11/bin/pg_ctl
```

This command initializes a PostgreSQL cluster at the location pointed by the `--pgdata` option. When `--pgdata` is omitted, `pg_autoctl` attempts to use the `PGDATA` environment variable. If a PostgreSQL instance had already existing in the destination directory, this command would have configured it to serve as a monitor.

`pg_auto_failover`, installs the `pgautofailover` Postgres extension, and grants access to a new `autoctl_node` user.

In the Quick Start we use `--auth trust` to avoid complex security settings. The Postgres trust authentication method[1] is not considered a reasonable choice for production environments. Consider either using the `--skip-pg-hba` option or `--auth scram-sha-256` and then setting up passwords yourself.

At this point the monitor is created. Now we'll install it as a service with systemd so that it will resume if the VM restarts.

```
ssh -l ha-admin `vm_ip monitor` << CMD
  pg_autoctl -q show systemd --pgdata ~ha-admin/monitor > pgautofailover.service
  sudo mv pgautofailover.service /etc/systemd/system
  sudo systemctl daemon-reload
  sudo systemctl enable pgautofailover
  sudo systemctl start pgautofailover
CMD
```

2.4 Bring up the nodes

We'll create the primary database using the `pg_autoctl create` subcommand.

```
ssh -l ha-admin `vm_ip a` -- \
  pg_autoctl create postgres \
    --pgdata ha \
```

(continues on next page)

[1] https://www.postgresql.org/docs/current/auth-trust.html_

(continued from previous page)

```
    --auth trust \
    --ssl-self-signed \
    --username ha-admin \
    --dbname appdb \
    --nodename ha-demo-a.internal.cloudapp.net \
    --pgctl /usr/lib/postgresql/11/bin/pg_ctl \
    --monitor 'postgres://autoctl_node@ha-demo-monitor.internal.cloudapp.net/pg_
↪auto_failover?sslmode=require'
```

Notice the user and database name in the monitor connection string – these are what monitor init created. We also give it the path to pg_ctl so that the keeper will use the correct version of pg_ctl in future even if other versions of postgres are installed on the system.

In the example above, the keeper creates a primary database. It chooses to set up node A as primary because the monitor reports there are no other nodes in the system yet. This is one example of how the keeper is state-based: it makes observations and then adjusts its state, in this case from "init" to "single."

Also add a setting to trust connections from our "application" VM:

```
ssh -l ha-admin `vm_ip a` << CMD
  echo 'hostssl "appdb" "ha-admin" ha-demo-app.internal.cloudapp.net trust' \
    >> ~ha-admin/ha/pg_hba.conf
CMD
```

At this point the monitor and primary node are created and running. Next we need to run the keeper. It's an independent process so that it can continue operating even if the PostgreSQL process goes terminates on the node. We'll install it as a service with systemd so that it will resume if the VM restarts.

```
ssh -l ha-admin `vm_ip a` << CMD
  pg_autoctl -q show systemd --pgdata ~ha-admin/ha > pgautofailover.service
  sudo mv pgautofailover.service /etc/systemd/system
  sudo systemctl daemon-reload
  sudo systemctl enable pgautofailover
  sudo systemctl start pgautofailover
CMD
```

Next connect to node B and do the same process. We'll do both steps at once:

```
ssh -l ha-admin `vm_ip b` -- \
  pg_autoctl create postgres \
    --pgdata ha \
    --auth trust \
    --ssl-self-signed \
```

(continues on next page)

(continued from previous page)

```
    --username ha-admin \
    --dbname appdb \
    --nodename ha-demo-b.internal.cloudapp.net \
    --pgctl /usr/lib/postgresql/11/bin/pg_ctl \
    --monitor 'postgres://autoctl_node@ha-demo-monitor.internal.cloudapp.net/pg_
↪auto_failover?sslmode=require'

ssh -l ha-admin `vm_ip b` << CMD
  pg_autoctl -q show systemd --pgdata ~ha-admin/ha > pgautofailover.service
  sudo mv pgautofailover.service /etc/systemd/system
  sudo systemctl daemon-reload
  sudo systemctl enable pgautofailover
  sudo systemctl start pgautofailover
CMD
```

It discovers from the monitor that a primary exists, and then switches its own state to be a hot standby and begins streaming WAL contents from the primary.

2.5 Node communication

For convenience, pg_autoctl modifies each node's `pg_hba.conf` file to allow the nodes to connect to one another. For instance, pg_autoctl added the following lines to node A:

```
# automatically added to node A

host "appdb" "ha-admin" ha-demo-a.internal.cloudapp.net trust
host replication "pgautofailover_replicator" ha-demo-b.internal.cloudapp.net␣
↪trust
host "appdb" "pgautofailover_replicator" ha-demo-b.internal.cloudapp.net trust
```

For `pg_hba.conf` on the monitor node pg_autoctl inspects the local network and makes its best guess about the subnet to allow. In our case it guessed correctly:

```
# automatically added to the monitor

hostssl "pg_auto_failover" "autoctl_node" 10.0.1.0/24 trust
```

If worker nodes have more ad-hoc addresses and are not in the same subnet, it's better to disable pg_autoctl's automatic modification of pg_hba using the `--skip-pg-hba` command line option during creation. You will then need to edit the hba file by hand. Another reason for manual edits would be to use special authentication methods.

2.6 Watch the replication

First let's verify that the monitor knows about our nodes, and see what states it has assigned them:

This looks good. We can add data to the primary, and later see it appear in the secondary. We'll connect to the database from inside our "app" virtual machine, using a connection string obtained from the monitor.

```
ssh -l ha-admin `vm_ip monitor` pg_autoctl show uri --pgdata monitor

      Type |    Name | Connection String
-----------+---------+-------------------------------
   monitor | monitor | postgres://autoctl_node@ha-demo-monitor.internal.
 ↪cloudapp.net:5432/pg_auto_failover?sslmode=require
 formation | default | postgres://ha-demo-b.internal.cloudapp.net:5432,ha-demo-
 ↪a.internal.cloudapp.net:5432/appdb?target_session_attrs=read-write&
 ↪sslmode=require
```

Now we'll get the connection string and store it in a local environment variable:

```
APP_DB_URI=$( \
  ssh -l ha-admin `vm_ip monitor` \
    pg_autoctl show uri --formation default --pgdata monitor \
)
```

The connection string contains both our nodes, comma separated, and includes the url parameter `?target_session_attrs=read-write` telling psql that we want to connect to whichever of these servers supports reads *and* writes. That will be the primary server.

```
# connect to database via psql on the app vm and
# create a table with a million rows
ssh -l ha-admin -t `vm_ip app` -- \
  psql "\"$APP_DB_URI\"" \
    -c "\"CREATE TABLE foo AS SELECT generate_series(1,1000000) bar;\""
```

2.7 Cause a failover

Now that we've added data to node A, let's switch which is considered the primary and which the secondary. After the switch we'll connect again and query the data, this time from node B.

```
# initiate failover to node B
ssh -l ha-admin -t `vm_ip monitor` \
  pg_autoctl perform switchover --pgdata monitor

# watch the progress
ssh -t -l ha-admin `vm_ip monitor` -- \
  watch -n 1 -d pg_autoctl show state --pgdata monitor

# press CTRL-C to stop watching
```

Once node B is marked "primary" (or "wait_primary") we can connect and verify that the data is still present:

```
# connect to database via psql on the app vm
ssh -l ha-admin -t `vm_ip app` -- \
  psql "\"$APP_DB_URI\"" \
    -c "\"SELECT count(*) FROM foo;\""
```

It shows

2.8 Cause a node failure

This plot is too boring, time to introduce a problem. We'll turn off VM for node B (currently the primary after our previous failover) and watch node A get promoted.

In one terminal let's keep an eye on events:

```
ssh -t -l ha-admin `vm_ip monitor` -- \
  watch -n 1 -d pg_autoctl show state --pgdata monitor
```

In another terminal we'll turn off the virtual server.

```
az vm stop \
  --resource-group ha-demo \
  --name ha-demo-b
```

After a number of failed attempts to talk to node B, the monitor determines the node is unhealthy and puts it into the "demoted" state. The monitor promotes node A to be the new primary.

```
.
                             Name |    Port | Group |  Node |      Current State |␣
↪    Assigned State
```

(continues on next page)

(continued from previous page)

```
--------------------------------+---------+-------+-------+--------------------+--
 ↪----------------
ha-demo-a.internal.cloudapp.net |    5432 |     0 |     1 |       wait_primary |␣
 ↪      wait_primary
ha-demo-b.internal.cloudapp.net |    5432 |     0 |     2 |            demoted |␣
 ↪       catchingup
```

Node A cannot be considered in full "primary" state since there is no secondary present, but it can still serve client requests. It is marked as "wait_primary" until a secondary appears, to indicate that it's running without a backup.

Let's add some data while B is offline.

```
# notice how $APP_DB_URI continues to work no matter which node
# is serving as primary
ssh -l ha-admin -t `vm_ip app` -- \
  psql "\"$APP_DB_URI\"" \
    -c "\"INSERT INTO foo SELECT generate_series(1000001, 2000000);\""
```

2.9 Resurrect node B

Run this command to bring node B back online:

```
az vm start \
  --resource-group ha-demo \
  --name ha-demo-b
```

Now the next time the keeper retries its health check, it brings the node back. Node B goes through the state "catchingup" while it updates its data to match A. Once that's done, B becomes a secondary, and A is now a full primary again.

```
.
                               Name |    Port | Group |  Node |     Current State |␣
 ↪    Assigned State
--------------------------------+---------+-------+-------+--------------------+--
 ↪----------------
ha-demo-a.internal.cloudapp.net |    5432 |     0 |     1 |            primary |␣
 ↪          primary
ha-demo-b.internal.cloudapp.net |    5432 |     0 |     2 |          secondary |␣
 ↪        secondary
```

What's more, if we connect directly to the database again, all two million rows are still present.

```
ssh -l ha-admin -t `vm_ip app` -- \
  psql "\"$APP_DB_URI\"" \
    -c "\"SELECT count(*) FROM foo;\""
```

It shows

CHAPTER 3

pg_auto_failover Architecture

pg_auto_failover is designed to handle a single PostgreSQL service using three nodes, and is resilient to losing any **one** of **three** nodes.

Note that a single Monitor can handle many PostgreSQL services, so that in practice if you want to handle N PostgreSQL services, you need at minimum 2 * N + 1 servers (not 3 * N).

pg_auto_failover considers that a PostgreSQL service is Highly-Available when the following two guarantees are respected, in this order:

1. Data loss is prevented in any situation that include the failure of a single node in the system.
2. In case of service downtime, service is back available as soon as possible, taking care of rule 1 first.

It is important to understand that pg_auto_failover is optimized for *Business Continuity*. In the event of losing a single node, then pg_auto_failover is capable of continuing the PostgreSQL service, and prevents any data loss when doing so, thanks to PostgreSQL *Synchronous Replication*.

That said, pg_auto_failover design trade-off towards business continuity involves relaxing replication guarantees to *asynchronous replication* in the event of a standby node failure. This allows the PostgreSQL service to accept writes when there's a single server available, and this opens the service for potential data loss if now the primary server were to be failing too.

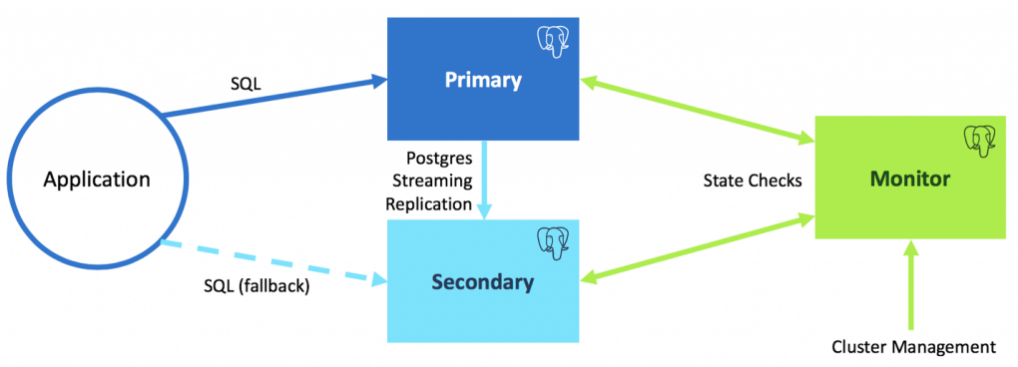

Fig. 1: pg_auto_failover Architecture for a standalone PostgreSQL service

3.1 The pg_auto_failover Monitor

Each PostgreSQL node in pg_auto_failover runs a Keeper process which informs a central Monitor node about notable local changes. Some changes require the Monitor to orchestrate a correction across the cluster:

- New nodes

  At initialization time, it's necessary to prepare the configuration of each node for PostgreSQL streaming replication, and get the cluster to converge to the nominal state with both a primary and a secondary node in each group. The monitor determines each new node's role

- Node failure

  The monitor orchestrates a failover when it detects an unhealthy node. The design of pg_auto_failover allows the monitor to shut down service to a previously designated primary node without causing a "split-brain" situation.

The monitor is the authoritative node that manages global state and makes changes in the cluster by issuing commands to the nodes' keeper processes. A pg_auto_failover monitor node failure has limited impact on the system. While it prevents reacting to other nodes' failures, it does not affect replication. The PostgreSQL streaming replication setup installed by pg_auto_failover does not depend on having the monitor up and running.

3.2 pg_auto_failover Glossary

pg_auto_failover handles a single PostgreSQL service with the following concepts:

- the pg_auto_failover MONITOR is a service that keeps track of one or several *formations* containing *groups* of two *nodes* each.

  The monitor is implemented as a PostgreSQL extension, so when you run the command `pg_autoctl create monitor` a PostgreSQL instance is initialized, configured with the extension, and started. The monitor service is embedded into a PostgreSQL service.

- a FORMATION is a logical set of PostgreSQL services.

- a GROUP of two PostgreSQL NODES work together to provide a single PostgreSQL service in a Highly Available fashion. A GROUP consists of a PostgreSQL primary server and a secondary server setup with Hot Standby synchronous replication. Note that pg_auto_failover can orchestrate the whole setting-up of the replication for you.

- the pg_auto_failover KEEPER is an agent that must be running on the same server where your PostgreSQL nodes are running. The KEEPER controls the local PostgreSQL instance (using both the `pg_ctl` command line tool and SQL queries), and communicates with the MONITOR:

  - it sends updated data about the local node, such as the WAL delta in between servers, measured via PostgreSQL statistics views,
  - it receives state assignments from the monitor.

  Also the KEEPER maintains a local state that includes the most recent communication established with the MONITOR and the other PostgreSQL

node of its group, enabling it to detect network partitions. More on that later.

- a NODE is a server (virtual or physical) that runs a PostgreSQL instances and a KEEPER service.
- a STATE is the representation of the per-instance and per-group situation. The monitor and the keeper implement a Finite State Machine to drive operations in the PostgreSQL groups and implement High Availability without data loss.

  The KEEPER main loop enforce the current expected state of the local PostgreSQL instance, and reports the current state and some more information to the MONITOR. The MONITOR uses this set of information and its own health-check information to drive the State Machine and assign a GOAL STATE to the KEEPER.

  The KEEPER implements the transitions between a current state and a MONITOR assigned goal state.

3.3 Client-side HA

Implementing client-side High Availability is included in PostgreSQL's driver *libpq* from version 10 onward. Using this driver, it is possible to specify multiple host names or IP addresses in the same connection string:

```
$ psql -d "postgresql://host1,host2/dbname?target_session_attrs=read-write"
$ psql -d "postgresql://host1:port2,host2:port2/dbname?target_session_
 ↪attrs=read-write"
$ psql -d "host=host1,host2 port=port1,port2 target_session_attrs=read-write"
```

When using either of the syntax above, the *psql* application attempts to connect to *host1*, and when successfully connected, checks the *target_session_attrs* as per the PostgreSQL documentation of it:

> If this parameter is set to read-write, only a connection in which read-write transactions are accepted by default is considered acceptable. The query SHOW transaction_read_only will be sent upon any successful connection; if it returns on, the connection will be closed. If multiple hosts were specified in the connection string, any remain-

ing servers will be tried just as if the connection attempt had failed. The default value of this parameter, any, regards all connections as acceptable.

When the connection attempt to *host1* fails, or when the *target_session_attrs* can not be verified, then the `psql` application attempts to connect to *host2*.

The behavior is implemented in the connection library *libpq*, so any application using it can benefit from this implementation, not just `psql`.

When using pg_auto_failover, configure your application connection string to use the primary and the secondary server host names, and set `target_session_attrs=read-write` too, so that your application automatically connects to the current primary, even after a failover occurred.

3.4 Monitoring protocol

The monitor interacts with the data nodes in 2 ways:

- Data nodes periodically connect and run *SELECT pgautofailover.node_active(...)* to communicate their current state and obtain their goal state.
- The monitor periodically connects to all the data nodes to see if they are healthy, doing the equivalent of `pg_isready`.

When a data node calls *node_active*, the state of the node is stored in the *pgautofailover.node* table and the state machines of both nodes are progressed. The state machines are described later in this readme. The monitor typically only moves one state forward and waits for the node(s) to converge except in failure states.

If a node is not communicating to the monitor, it will either cause a failover (if node is a primary), disabling synchronous replication (if node is a secondary), or cause the state machine to pause until the node comes back (other cases). In most cases, the latter is harmless, though in some cases it may cause downtime to last longer, e.g. if a standby goes down during a failover.

To simplify operations, a node is only considered unhealthy if the monitor cannot connect *and* it hasn't reported its state through *node_active* for a while. This

allows, for example, PostgreSQL to be restarted without causing a health check failure.

3.5 Synchronous vs. asynchronous replication

By default, pg_auto_failover uses synchronous replication, which means all writes block until the standby has accepted them. To handle cases in which the standby fails, the primary switches between two states called *wait_primary* and *primary* based on the health of the standby.

In *wait_primary*, synchronous replication is disabled by automatically setting `synchronous_standby_names = ''` to allow writes to proceed, but failover is also disabled since the standby might get arbitrarily far behind. If the standby is responding to health checks and within 1 WAL segment of the primary (configurable), synchronous replication is re-enabled on the primary by setting `synchronous_standby_names = '*'` which may cause a short latency spike since writes will then block until the standby has caught up.

If you wish to disable synchronous replication, you need to add the following to `postgresql.conf`:

```
synchronous_commit = 'local'
```

This ensures that writes return as soon as they are committed on the primary under all circumstances. In that case, failover might lead to some data loss, but failover is not initiated if the secondary is more than 10 WAL segments (configurable) behind on the primary. During a manual failover, the standby will continue accepting writes from the old primary and will stop only if it's fully caught up (most common), the primary fails, or it does not receive writes for 2 minutes.

3.6 Node recovery

When bringing back a node after a failover, the keeper (`pg_autoctl run`) can simply be restarted. It will also restart postgres if needed and obtain its goal

state from the monitor. If the failed node was a primary and was demoted, it will learn this from the monitor. Once the node reports, it is allowed to come back as a standby by running `pg_rewind`. If it is too far behind the node performs a new `pg_basebackup`.

3.7 Failover logic

This section needs to be expanded further, but below is the failover state machine for each node that is implemented by the monitor:

Since the state machines of the data nodes always move in tandem, a pair (group) of data nodes also implicitly has the following state machine:

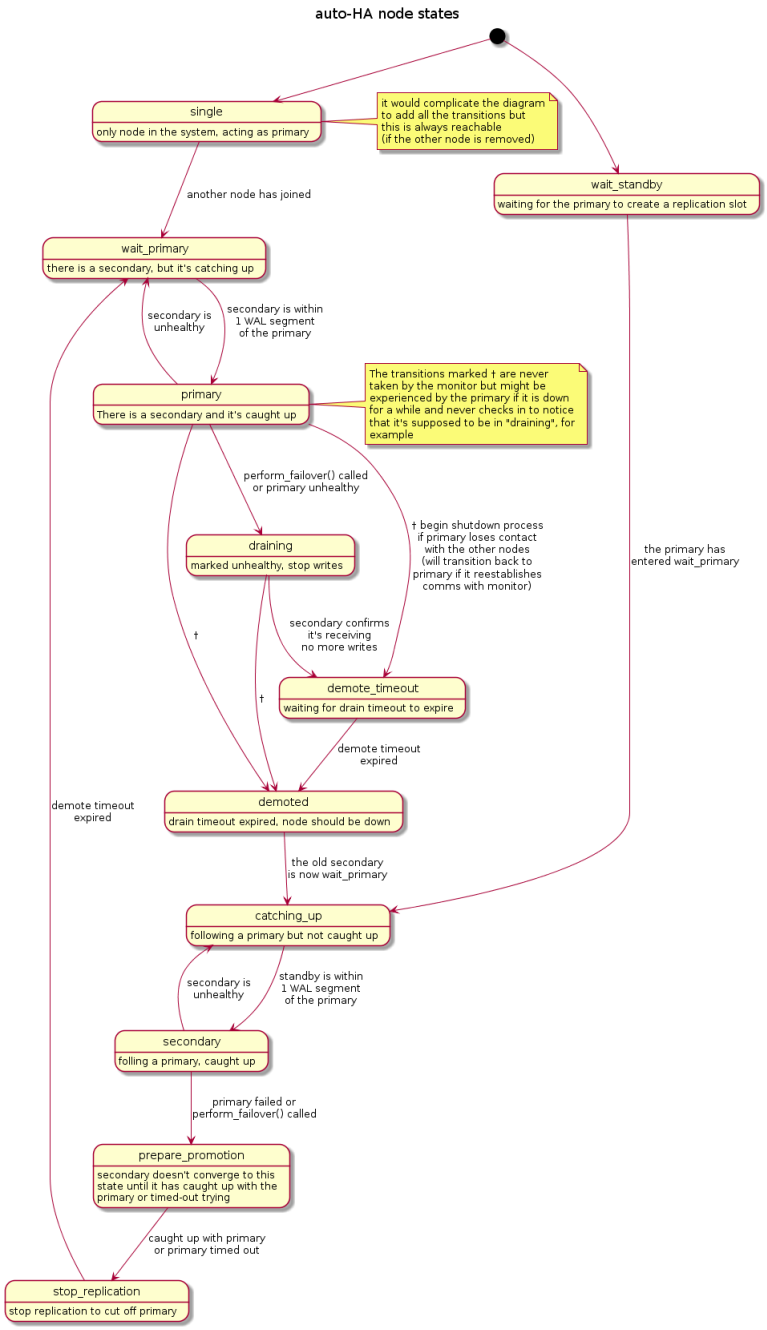

Fig. 2: Node state machine

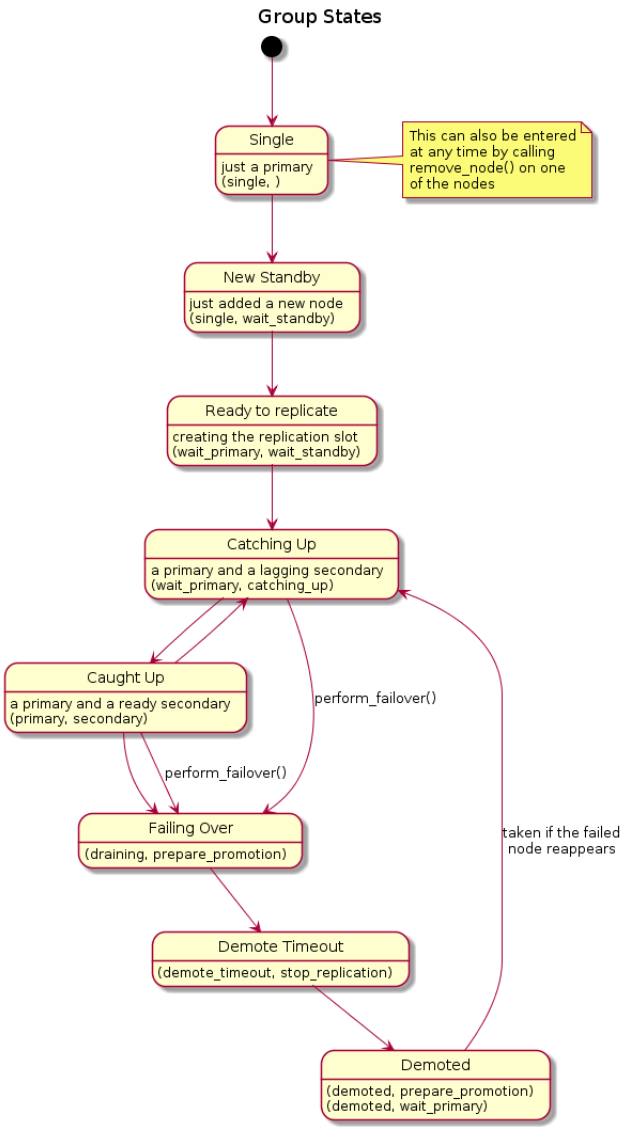

Fig. 3: Group state machine

3.8 pg_auto_failover keeper's State Machine

When built in TEST mode, it is then possible to use the following command to get a visual representation of the Keeper's Finite State Machine:

```
$ PG_AUTOCTL_DEBUG=1 pg_autoctl do fsm gv | dot -Tsvg > fsm.svg
```

The *dot* program is part of the Graphviz suite and produces the following output:

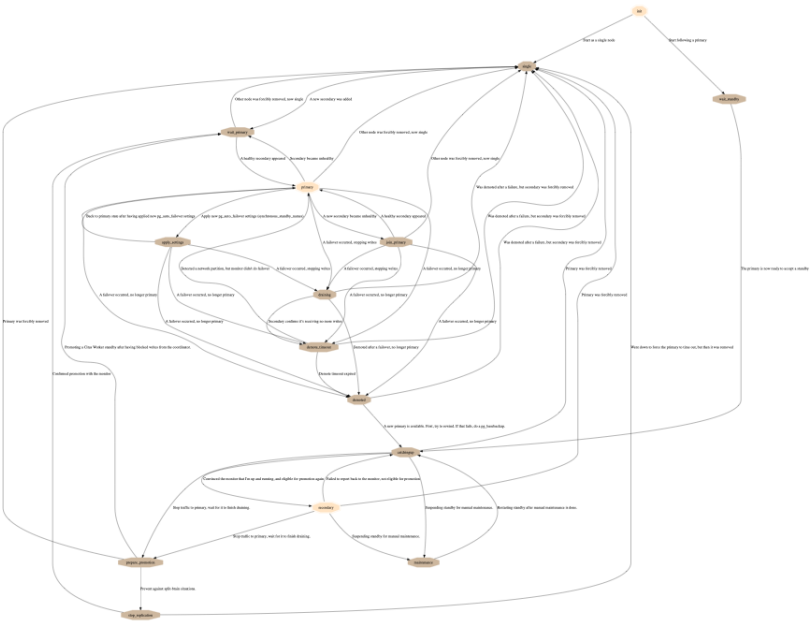

Fig. 4: Keeper State Machine

CHAPTER 4

pg_auto_failover Fault Tolerance

At the heart of the pg_auto_failover implementation is a State Machine. The state machine is driven by the monitor, and its transitions are implemented in the keeper service, which then reports success to the monitor.

The keeper is allowed to retry transitions as many times as needed until they succeed, and reports also failures to reach the assigned state to the monitor node. The monitor also implements frequent health-checks targeting the registered PostgreSQL nodes.

When the monitor detects something is not as expected, it takes action by assigning a new goal state to the keeper, that is responsible for implementing the transition to this new state, and then reporting.

4.1 Unhealthy Nodes

The pg_auto_failover monitor is responsible for running regular health-checks with every PostgreSQL node it manages. A health-check is successful when it is

able to connect to the PostgreSQL node using the PostgreSQL protocol (libpq), imitating the `pg_isready` command.

How frequent those health checks are (20s by default), the PostgreSQL connection timeout in use (5s by default), and how many times to retry in case of a failure before marking the node unhealthy (2 by default) are GUC variables that you can set on the Monitor node itself. Remember, the monitor is implemented as a PostgreSQL extension, so the setup is a set of PostgreSQL configuration settings:

```
  SELECT name, setting
    FROM pg_settings
   WHERE name ~ 'pgautofailover\.health';
                  name                    | setting
------------------------------------------+---------
 pgautofailover.health_check_max_retries | 2
 pgautofailover.health_check_period      | 20000
 pgautofailover.health_check_retry_delay | 2000
 pgautofailover.health_check_timeout     | 5000
(4 rows)
```

The pg_auto_failover keeper also reports if PostgreSQL is running as expected. This is useful for situations where the PostgreSQL server / OS is running fine and the keeper (`pg_autoctl run`) is still active, but PostgreSQL has failed. Situations might include *File System is Full* on the WAL disk, some file system level corruption, missing files, etc.

Here's what happens to your PostgreSQL service in case of any single-node failure is observed:

- Primary node is monitored unhealthy

  When the primary node is unhealthy, and only when the secondary node is itself in good health, then the primary node is asked to transition to the DRAINING state, and the attached secondary is asked to transition to the state PREPARE_PROMOTION. In this state, the secondary is asked to catch-up with the WAL traffic from the primary, and then report success.

  The monitor then continues orchestrating the promotion of the standby: it stops the primary (implementing STONITH in order to prevent any data loss), and promotes the secondary into being a primary now.

  Depending on the exact situation that triggered the primary unhealthy, it's possible that the secondary fails to catch-up with WAL from it, in that case after the PREPARE_PROMOTION_CATCHUP_TIMEOUT the standby

reports success anyway, and the failover sequence continues from the monitor.

- Secondary node is monitored unhealthy

  When the secondary node is unhealthy, the monitor assigns to it the state CATCHINGUP, and assigns the state WAIT_PRIMARY to the primary node. When implementing the transition from PRIMARY to WAIT_PRIMARY, the keeper disables synchronous replication.

  When the keeper reports an acceptable WAL difference in the two nodes again, then the replication is upgraded back to being synchronous. While a secondary node is not in the SECONDARY state, secondary promotion is disabled.

- Monitor node has failed

  Then the primary and secondary node just work as if you didn't have setup pg_auto_failover in the first place, as the keeper fails to report local state from the nodes. Also, health checks are not performed. It means that no automated failover may happen, even if needed.

4.2 Network Partitions

Adding to those simple situations, pg_auto_failover is also resilient to Network Partitions. Here's the list of situation that have an impact to pg_auto_failover behavior, and the actions taken to ensure High Availability of your PostgreSQL service:

- Primary can't connect to Monitor

  Then it could be that either the primary is alone on its side of a network split, or that the monitor has failed. The keeper decides depending on whether the secondary node is still connected to the replication slot, and if we have a secondary, continues to serve PostgreSQL queries.

  Otherwise, when the secondary isn't connected, and after the NETWORK_PARTITION_TIMEOUT has elapsed, the primary considers it might be alone in a network partition: that's a potential split brain situation and with only one way to prevent it. The primary stops, and reports

a new state of DEMOTE_TIMEOUT.

The network_partition_timeout can be setup in the keeper's configuration and defaults to 20s.

- Monitor can't connect to Primary

  Once all the retries have been done and the timeouts are elapsed, then the primary node is considered unhealthy, and the monitor begins the failover routine. This routine has several steps, each of them allows to control our expectations and step back if needed.

  For the failover to happen, the secondary node needs to be healthy and caught-up with the primary. Only if we timeout while waiting for the WAL delta to resorb (30s by default) then the secondary can be promoted with uncertainty about the data durability in the group.

- Monitor can't connect to Secondary

  As soon as the secondary is considered unhealthy then the monitor changes the replication setting to asynchronous on the primary, by assigning it the WAIT_PRIMARY state. Also the secondary is assigned the state CATCHINGUP, which means it can't be promoted in case of primary failure.

  As the monitor tracks the WAL delta between the two servers, and they both report it independently, the standby is eligible to promotion again as soon as it's caught-up with the primary again, and at this time it is assigned the SECONDARY state, and the replication will be switched back to synchronous.

4.3 Failure handling and network partition detection

If a node cannot communicate to the monitor, either because the monitor is down or because there is a problem with the network, it will simply remain in the same state until the monitor comes back.

If there is a network partition, it might be that the monitor and secondary can still communicate and the monitor decides to promote the secondary since the

primary is no longer responsive. Meanwhile, the primary is still up-and-running on the other side of the network partition. If a primary cannot communicate to the monitor it starts checking whether the secondary is still connected. In PostgreSQL, the secondary connection automatically times out after 30 seconds. If last contact with the monitor and the last time a connection from the secondary was observed are both more than 30 seconds in the past, the primary concludes it is on the losing side of a network partition and shuts itself down. It may be that the secondary and the monitor were actually down and the primary was the only node that was alive, but we currently do not have a way to distinguish such a situation. As with consensus algorithms, availability can only be correctly preserved if at least 2 out of 3 nodes are up.

In asymmetric network partitions, the primary might still be able to talk to the secondary, while unable to talk to the monitor. During failover, the monitor therefore assigns the secondary the *stop_replication* state, which will cause it to disconnect from the primary. After that, the primary is expected to shut down after at least 30 and at most 60 seconds. To factor in worst-case scenarios, the monitor waits for 90 seconds before promoting the secondary to become the new primary.

CHAPTER 5

Installing pg_auto_failover

We provide native system packages for pg_auto_failover on most popular Linux distributions.

Use the steps below to install pg_auto_failover on PostgreSQL 11. At the current time pg_auto_failover is compatible with both PostgreSQL 10 and PostgreSQL 11.

5.1 Ubuntu or Debian

5.1.1 Quick install

The following installation method downloads a bash script that automates several steps. The full script is available for review at our package cloud installation instructions[2] page.

[2] https://packagecloud.io/citusdata/community/install#bash

```
# add the required packages to your system
curl https://install.citusdata.com/community/deb.sh | sudo bash

# install pg_auto_failover
sudo apt-get install postgresql-11-auto-failover

# confirm installation
/usr/bin/pg_autoctl --version
```

5.1.2 Manual Installation

If you'd prefer to install your repo on your system manually, follow the instructions from package cloud manual installation[3] page. This page will guide you with the specific details to achieve the 3 steps:

1. install CitusData GnuPG key for its package repository
2. install a new apt source for CitusData packages
3. update your available package list

Then when that's done, you can proceed with installing pg_auto_failover itself as in the previous case:

```
# install pg_auto_failover
sudo apt-get install postgresql-11-auto-failover

# confirm installation
/usr/bin/pg_autoctl --version
```

5.2 Fedora, CentOS, or Red Hat

5.2.1 Quick install

The following installation method downloads a bash script that automates several steps. The full script is available for review at our package cloud installation instructions page[4] url.

[3] https://packagecloud.io/citusdata/community/install#manual
[4] https://packagecloud.io/citusdata/community/install#bash

```
# add the required packages to your system
curl https://install.citusdata.com/community/rpm.sh | sudo bash

# install pg_auto_failover
sudo yum install -y pg-auto-failover10_11

# confirm installation
/usr/pgsql-11/bin/pg_autoctl --version
```

5.2.2 Manual installation

If you'd prefer to install your repo on your system manually, follow the instructions from package cloud manual installation[5] page. This page will guide you with the specific details to achieve the 3 steps:

1. install the pygpgme yum-utils packages for your distribution
2. install a new RPM reposiroty for CitusData packages
3. update your local yum cache

Then when that's done, you can proceed with installing pg_auto_failover itself as in the previous case:

```
# install pg_auto_failover
sudo yum install -y pg-auto-failover10_11

# confirm installation
/usr/pgsql-11/bin/pg_autoctl --version
```

5.3 Installing a pgautofailover Systemd unit

The command `pg_autoctl show systemd` outputs a systemd unit file that you can use to setup a boot-time registered service for pg_auto_failover on your machine.

Here's a sample output from the command:

[5] https://packagecloud.io/citusdata/community/install#manual-rpm

```
$ export PGDATA=/var/lib/postgresql/monitor
$ pg_autoctl show systemd
13:44:34 INFO  HINT: to complete a systemd integration, run the following␣
↪commands:
13:44:34 INFO  pg_autoctl -q show systemd --pgdata "/var/lib/postgresql/monitor
↪" | sudo tee /etc/systemd/system/pgautofailover.service
13:44:34 INFO  sudo systemctl daemon-reload
13:44:34 INFO  sudo systemctl start pgautofailover
[Unit]
Description = pg_auto_failover

[Service]
WorkingDirectory = /var/lib/postgresql
Environment = 'PGDATA=/var/lib/postgresql/monitor'
User = postgres
ExecStart = /usr/lib/postgresql/10/bin/pg_autoctl run
Restart = always
StartLimitBurst = 0

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

Copy/pasting the commands given in the hint output from the command will enable the pgautofailer service on your system, when using systemd.

It is important that PostgreSQL is started by `pg_autoctl` rather than by systemd itself, as it might be that a failover has been done during a reboot, for instance, and that once the reboot complete we want the local Postgres to re-join as a secondary node where it used to be a primary node.

CHAPTER 6

Security settings for pg_auto_failover

In order to be able to orchestrate fully automated failovers, pg_auto_failover needs to be able to establish the following Postgres connections:

- from the monitor node to each Postgres node to check the node's "health"
- from each Postgres node to the monitor to implement our *node_active* protocol and fetch the current assigned state for this node
- from the secondary node to the primary node for Postgres streaming replication.

Postgres Client authentication is controlled by a configuration file: `pg_hba.conf`. This file contains a list of rules where each rule may allow or reject a connection attempt.

For pg_auto_failover to work as intended, some HBA rules need to be added to each node configuration. You can choose to provision the `pg_hba.conf` file yourself thanks to `pg_autoctl` options' `--skip-pg-hba`, or you can use the following options to control which kind of rules are going to be added for you.

6.1 Postgres HBA rules

For your application to be able to connect to the current Postgres primary servers, some application specific HBA rules have to be added to `pg_hba.conf`. There is no provision for doing that in pg_auto_failover.

In other words, it is expected that you have to edit `pg_hba.conf` to open connections for your application needs.

6.2 The trust security model

As its name suggests the trust security model is not enabling any kind of security validation. This setting is popular for testing deployments though, as it makes it very easy to verify that everything works as intended before putting security restrictions in place.

To enable a "trust" security model with pg_auto_failover, use the `pg_autoctl` option `--auth trust` when creating nodes:

```
$ pg_autoctl create monitor --auth trust ...
$ pg_autoctl create postgres --auth trust ...
$ pg_autoctl create postgres --auth trust ...
```

When using `--auth trust` pg_autoctl adds new HBA rules in the monitor and the Postgres nodes to enable connections as seen above.

6.3 Authentication with passwords

To setup pg_auto_failover with password for connections, you can use one of the password based authentication methods supported by Postgres, such as `password` or `scram-sha-256`. We recommend the latter, as in the following example:

```
$ pg_autoctl create monitor --auth scram-sha-256 ...
```

The pg_autoctl does not set the password for you. The first step is to set the database user password in the monitor database thanks to the following command:

```
$ psql postgres://monitor.host/pg_auto_failover
> alter user autoctl_node password 'h4ckm3';
```

Now that the monitor is ready with our password set for the autoctl_node user, we can use the password in the monitor connection string used when creating Postgres nodes.

On the primary node, we can create the Postgres setup as usual, and then set our replication password, that we will use if we are demoted and then re-join as a standby:

```
$ pg_autoctl create postgres       \
       --auth scram-sha-256        \
       ...                         \
       --monitor postgres://autoctl_node:h4ckm3@monitor.host/pg_auto_failover

$ pg_autoctl config set replication.password h4ckm3m0r3
```

The second Postgres node is going to be initialized as a secondary and pg_autoctl then calls pg_basebackup at create time. We need to have the replication password already set at this time, and we can achieve that the following way:

```
$ export PGPASSWORD=h4ckm3m0r3
$ pg_autoctl create postgres       \
       --auth scram-sha-256        \
       ...                         \
       --monitor postgres://autoctl_node:h4ckm3@monitor.host/pg_auto_failover

$ pg_autoctl config set replication.password h4ckm3m0r3
```

Note that you can use The Password File[6] mechanism as discussed in the Postgres documentation in order to maintain your passwords in a separate file, not in your main pg_auto_failover configuration file. This also avoids using passwords in the environment and in command lines.

[6] https://www.postgresql.org/docs/current/libpq-pgpass.html

6.4 Encryption of network communications

Postgres knows how to use SSL to enable network encryption of all communications, including authentication with passwords and the whole data set when streaming replication is used.

To enable SSL on the server an SSL certificate is needed. It could be as simple as a self-signed certificate, and `pg_autoctl` creates such a certificate for you when using `--ssl-self-signed` command line option:

```
$ pg_autoctl create monitor --ssl-self-signed ...      \
                            --auth scram-sha-256 ...   \
                            --ssl-mode require         \
                            ...

$ pg_autoctl create postgres --ssl-self-signed ...      \
                             --auth scram-sha-256 ...   \
                             ...

$ pg_autoctl create postgres --ssl-self-signed ...      \
                             --auth scram-sha-256 ...   \
                             ...
```

In that example we setup SSL connections to encrypt the network traffic, and we still have to setup an authentication mechanism exactly as in the previous sections of this document. Here `scram-sha-256` has been selected, and the password will be sent over an encrypted channel.

When using the `--ssl-self-signed` option, `pg_autoctl` creates a self-signed certificate, as per the Postgres documentation at the Creating Certificates[7] page.

The certificate subject CN defaults to the `--hostname` parameter, which can be given explicitely or computed by `pg_autoctl` as either your hostname when you have proper DNS resolution, or your current IP address.

Self-signed certificates provide protection against eavesdropping; this setup does NOT protect against Man-In-The-Middle attacks nor Impersonation attacks. See PostgreSQL documentation page SSL Support[8] for details.

[7] https://www.postgresql.org/docs/current/ssl-tcp.html#SSL-CERTIFICATE-CREATION
[8] https://www.postgresql.org/docs/current/libpq-ssl.html

6.5 Using your own SSL certificates

In many cases you will want to install certificates provided by your local security department and signed by a trusted Certificate Authority. In that case one solution is to use `--skip-pg-hba` and do the whole setup yourself.

It is still possible to give the certificates to pg_auto_failover and have it handle the setup for you, including the creation of and signing of client certificates for the `autoctl_node` and `pgautofailover_replication` users:

```
$ pg_autoctl create monitor --ssl-ca-file root.crt   \
                            --ssl-crl-file root.crl  \
                            --server-cert server.crt  \
                            --server-key server.key  \
                            --ssl-mode verify-full \
                            ...

$ pg_autoctl create postgres --ssl-ca-file root.crt   \
                             --server-cert server.crt  \
                             --server-key server.key  \
                             --ssl-mode verify-full \
                             ...

$ pg_autoctl create postgres --ssl-ca-file root.crt   \
                             --server-cert server.crt  \
                             --server-key server.key  \
                             --ssl-mode verify-full \
                             ...
```

The option `--ssl-mode` can be used to force connection strings used by `pg_autoctl` to contain your prefered ssl mode. It defaults to `require` when using `--ssl-self-signed` and to `allow` when `--no-ssl` is used. Here, we set `--ssl-mode` to `validate-ca` which requires SSL Certificates Authentication, covered next.

The default `--ssl-mode` when providing your own certificates (signed by your trusted CA) is then `verify-full`. This setup applies to the client connection where the server identity is going to be checked against the root certificate provided with `--ssl-ca-file` and the revocation list optionally provided with the `--ssl-crl-file`. Both those files are used as the respective parameters `sslrootcert` and `sslcrl` in pg_autoctl connection strings to both the monitor and the streaming replication primary server.

6.6 SSL Certificates Authentication

Given those files, it is then possible to use certificate based authentication of client connections. For that, it is necessary to prepare client certificates signed by your root certificate private key and using the target user name as its CN, as per Postgres documentation for Certificate Authentication[9]:

> The cn (Common Name) attribute of the certificate will be compared to the requested database user name, and if they match the login will be allowed

For enabling the *cert* authentication method with pg_auto_failover, you need to prepare a client certificate for the user `postgres` and used by pg_autoctl when connecting to the monitor, to place in `~/.postgresql/postgresql.cert` along with its key `~/.postgresql/postgresql.key`, in the home directory of the user that runs the pg_autoctl service (which defaults to `postgres`).

Then you need to create a user name map as documented in Postgres page User Name Maps[10] so that your certificate can be used to authenticate pg_autoctl users.

The ident map in `pg_ident.conf` on the pg_auto_failover monitor should then have the following entry, to allow `postgres` to connect as the `autoctl_node` user for `pg_autoctl` operations:

```
# MAPNAME       SYSTEM-USERNAME         PG-USERNAME

# pg_autoctl runs as postgres and connects to the monitor autoctl_node user
pgautofailover   postgres               autoctl_node
```

To enable streaming replication, the `pg_ident.conf` file on each Postgres node should now allow the `postgres` user in the client certificate to connect as the `pgautofailover_replicator` database user:

```
# MAPNAME       SYSTEM-USERNAME         PG-USERNAME

# pg_autoctl runs as postgres and connects to the monitor autoctl_node user
pgautofailover  postgres               pgautofailover_replicator
```

[9] https://www.postgresql.org/docs/current/auth-cert.html
[10] https://www.postgresql.org/docs/current/auth-username-maps.html

Given that user name map, you can then use the `cert` authentication method. As with the `pg_ident.conf` provisioning, it is best to now provision the HBA rules yourself, using the `--skip-pg-hba` option:

```
$ pg_autoctl create postgres --skip-pg-hba --ssl-ca-file ...
```

The HBA rule will use the authentication method `cert` with a map option, and might then look like the following on the monitor:

```
# allow certificate based authentication to the monitor
hostssl pg_auto_failover autoctl_node 10.0.0.0/8 cert map=pgautofailover
```

Then your pg_auto_failover nodes on the 10.0.0.0 network are allowed to connect to the monitor with the user `autoctl_node` used by `pg_autoctl`, assuming they have a valid and trusted client certificate.

The HBA rule to use on the Postgres nodes to allow for Postgres streaming replication connections looks like the following:

```
# allow streaming replication for pg_auto_failover nodes
hostssl replication pgautofailover_replicator 10.0.0.0/8 cert map=pgautofailover
```

Because the Postgres server runs as the `postgres` system user, the connection to the primary node can be made with SSL enabled and will then use the client certificates installed in the `postgres` home directory in `~/.postgresql/postgresql.{key,cert}` locations.

6.7 Postgres HBA provisioning

While pg_auto_failover knows how to manage the Postgres HBA rules that are necessary for your stream replication needs and for its monitor protocol, it will not manage the Postgres HBA rules that are needed for your applications.

If you have your own HBA provisioning solution, you can include the rules needed for pg_auto_failover and then use the `--skip-pg-hba` option to the `pg_autoctl create` commands.

6.8 Enable SSL connections on an existing setup

Whether you upgrade pg_auto_failover from a previous version that did not have support for the SSL features, or when you started with `--no-ssl` and later change your mind, it is possible with pg_auto_failover to add SSL settings on system that has already been setup without explicit SSL support.

In this section we detail how to upgrade to SSL settings.

Installing Self-Signed certificates on-top of an already existing pg_auto_failover setup is done with one of the following pg_autoctl command variants, depending if you want self-signed certificates or fully verified ssl certificates:

```
$ pg_autoctl enable ssl --ssl-self-signed --ssl-mode required

$ pg_autoctl enable ssl --ssl-ca-file root.crt   \
                        --ssl-crl-file root.crl  \
                        --server-cert server.crt  \
                        --server-key server.key  \
                        --ssl-mode verify-full
```

The `pg_autoctl enable ssl` command edits the `postgresql-auto-failover.conf` Postgres configuration file to match the command line arguments given and enable SSL as instructed, and then updates the pg_autoctl configuration.

The connection string to connect to the monitor is also automatically updated by the `pg_autoctl enable ssl` command. You can verify your new configuration with:

```
$ pg_autoctl config get pg_autoctl.monitor
```

Note that an already running pg_autoctl deamon will try to reload its configuration after `pg_autoctl enable ssl` has finished. In some cases this is not possible to do without a restart. So be sure to check the logs from a running daemon to confirm that the reload succeeded. If it did not you may need to restart the daemon to ensure the new connection string is used.

The HBA settings are not edited, irrespective of the `--skip-pg-hba` that has been used at creation time. That's because the `host` records match either SSL or non-SSL connection attempts in Postgres HBA file, so the pre-existing setup

will continue to work. To enhance the SSL setup, you can manually edit the HBA files and change the existing lines from `host` to `hostssl` to dissallow unencrypted connections at the server side.

In summary, to upgrade an existing pg_auto_failover setup to enable SSL:

1. run the `pg_autoctl enable ssl` command on your monitor and then all the Postgres nodes,
2. on the Postgres nodes, review your pg_autoctl logs to make sure that the reload operation has been effective, and review your Postgres settings to verify that you have the expected result,
3. review your HBA rules setup to change the pg_auto_failover rules from `host` to `hostssl` to disallow insecure connections.

CHAPTER 7

pg_autoctl commands reference

pg_auto_failover comes with a PostgreSQL extension and a service:

- The *pgautofailover* PostgreSQL extension implements the pg_auto_failover monitor.
- The `pg_autoctl` command manages pg_auto_failover services.

7.1 pg_autoctl

The `pg_autoctl` command implements a service that is meant to run in the background. The command line controls the service, and uses the service API for manual operations.

The pg_auto_failover service implements the steps to set up replication and node promotion according to instructions from the pg_auto_failover monitor. Every 5 seconds, the keeper service (started by `pg_autoctl run`) connects to the PostgreSQL monitor database and runs `SELECT pg_auto_failover.`

`node_active(...)` to simultaneously communicate its current state and obtain its goal state. It stores its current state in its own state file, as configured.

The `pg_autoctl` command includes facilities for initializing and operating both the pg_auto_failover monitor and the PostgreSQL instances with a pg_auto_failover keeper:

```
$ pg_autoctl help
  pg_autoctl
  + create    Create a pg_auto_failover node, or formation
  + drop      Drop a pg_auto_failover node, or formation
  + config    Manages the pg_autoctl configuration
  + show      Show pg_auto_failover information
  + enable    Enable a feature on a formation
  + disable   Disable a feature on a formation
  + perform  Perform an action orchestrated by the monitor
    run       Run the pg_autoctl service (monitor or keeper)
    stop      signal the pg_autoctl service for it to stop
    reload    signal the pg_autoctl for it to reload its configuration
    help      print help message
    version   print pg_autoctl version

  pg_autoctl create
    monitor      Initialize a pg_auto_failover monitor node
    postgres     Initialize a pg_auto_failover standalone postgres node
    formation    Create a new formation on the pg_auto_failover monitor

  pg_autoctl drop
    monitor    Drop the pg_auto_failover monitor
    node       Drop a node from the pg_auto_failover monitor
    formation  Drop a formation on the pg_auto_failover monitor

  pg_autoctl config
    check  Check pg_autoctl configuration
    get    Get the value of a given pg_autoctl configuration variable
    set    Set the value of a given pg_autoctl configuration variable

  pg_autoctl show
    uri      Show the postgres uri to use to connect to pg_auto_failover nodes
    events   Prints monitor's state of nodes in a given formation and group
    state    Prints monitor's state of nodes in a given formation and group
    file     List pg_autoctl internal files (config, state, pid)
    systemd  Print systemd service file for this node

  pg_autoctl enable
    secondary    Enable secondary nodes on a formation
    maintenance  Enable Postgres maintenance mode on this node
    ssl          Enable SSL configuration on this node

  pg_autoctl disable
    secondary    Disable secondary nodes on a formation
    maintenance  Disable Postgres maintenance mode on this node
    ssl          Disable SSL configuration on this node
```

(continues on next page)

(continued from previous page)

```
pg_autoctl perform
  failover    Perform a failover for given formation and group
  switchover  Perform a switchover for given formation and group
```

The first step consists of creating a pg_auto_failover monitor thanks to the command `pg_autoctl create monitor`, and the command `pg_autoctl show uri` can then be used to find the Postgres connection URI string to use as the `--monitor` option to the `pg_autoctl create` command for the other nodes of the formation.

7.1.1 pg_auto_failover Monitor

The main piece of the pg_auto_failover deployment is the monitor. The following commands are dealing with the monitor:

- `pg_autoctl create monitor`

  This command initializes a PostgreSQL cluster and installs the *pgautofailover* extension so that it's possible to use the new instance to monitor PostgreSQL services:

  ```
  $ pg_autoctl create monitor --help
  pg_autoctl create monitor: Initialize a pg_auto_failover␣
   ↪monitor node
  usage: pg_autoctl create monitor  [ --pgdata --pgport --pgctl -
   ↪-hostname ]

    --pgctl       path to pg_ctl
    --pgdata      path to data directory
    --pgport      PostgreSQL's port number
    --hostname    hostname by which postgres is reachable
    --auth        authentication method for connections from␣
   ↪data nodes
    --skip-pg-hba skip editing pg_hba.conf rules
    --run         create node then run pg_autoctl service
  ```

  The `--pgdata` option is mandatory and default to the environment variable `PGDATA`. The `--pgport` default value is 5432, and the `--pgctl` option defaults to the first `pg_ctl` entry found in your *PATH*.

  The `--hostname` option allows setting the hostname that the other nodes of the cluster will use to access to the monitor.

When not provided, a default value is computed by running the following algorithm:

1. Open a connection to the 8.8.8.8:53 public service and looks up the TCP/IP client address that has been used to make that connection.
2. Do a reverse DNS lookup on this IP address to fetch a hostname for our local machine.
3. If the reverse DNS lookup is successful , then *pg_autoctl* does with a forward DNS lookup of that hostname.

When the forward DNS lookup response in step 3. is an IP address found in one of our local network interfaces, then *pg_autoctl* uses the hostname found in step 2. as the default *–hostname*. Otherwise it uses the IP address found in step 1.

You may use the *–hostname* command line option to bypass the whole DNS lookup based process and force the local node name to a fixed value.

The `--auth` option allows setting up authentication method to be used for connections from data nodes with `autoctl_node` user. When testing pg_auto_failover for the first time using `--auth trust` makes things easier. When getting production ready, review your options here and choose at least `--auth scram-sha-256` and make sure password is manually set on the monitor, and appropriate setting is added to *.pgpass* file on data node. You could also use some of the advanced Postgres authentication mechanism such as SSL certificates.

See *Security* for notes on *.pgpass*

- `pg_autoctl run`

  This command makes sure that the PostgreSQL instance for the monitor is running, then connects to it and listens to the monitor notifications, displaying them as log messages:

```
$ pg_autoctl run --help
pg_autoctl run: Run the pg_autoctl service (monitor or keeper)
usage: pg_autoctl run  [ --pgdata ]
```

(continues on next page)

(continued from previous page)

```
  --pgdata      path to data directory
```

The option *–pgdata* (or the environment variable `PGDATA`) allows pg_auto_failover to find the monitor configuration file.

- `pg_autoctl create formation`

  This command registers a new formation on the monitor, with the specified kind:

```
$ pg_autoctl create formation --help
pg_autoctl create formation: Create a new formation on the pg_auto_
 ↪failover monitor
usage: pg_autoctl create formation  [ --pgdata --formation --kind --
 ↪dbname --with-secondary --without-secondary ]

  --pgdata            path to data directory
  --formation         name of the formation to create
  --kind              formation kind, either "pgsql" or "citus"
  --dbname            name for postgres database to use in this formation
  --enable-secondary  create a formation that has multiple nodes that␣
 ↪can be
                      used for fail over when others have issues
  --disable-secondary create a citus formation without nodes to fail␣
 ↪over to
```

- `pg_autoctl drop formation`

  This command drops an existing formation on the monitor:

```
$ pg_autoctl drop formation --help
pg_autoctl drop formation: Drop a formation on the pg_auto_failover␣
 ↪monitor
usage: pg_autoctl drop formation  [ --pgdata --formation ]

  --pgdata      path to data directory
  --formation   name of the formation to drop
```

7.1.2 pg_autoctl show command

To discover current information about a pg_auto_failover setup, the `pg_autoctl show` commands can be used, from any node in the setup.

- `pg_autoctl show uri`

This command outputs the monitor or the coordinator Postgres URI to use from an application to connect to the service:

```
$ pg_autoctl show uri --help
pg_autoctl show uri: Show the postgres uri to use to connect to pg_auto_
 ↪failover nodes
usage: pg_autoctl show uri  [ --pgdata --formation ]

  --pgdata      path to data directory
  --formation   show the coordinator uri of given formation
```

The option `--formation default` outputs the Postgres URI to use to connect to the Postgres server.

- `pg_autoctl show events`

  This command outputs the latest events known to the pg_auto_failover monitor:

```
$ pg_autoctl show events --help
pg_autoctl show events: Prints monitor's state of nodes in a given␣
 ↪formation and group
usage: pg_autoctl show events  [ --pgdata --formation --group --count ]

  --pgdata      path to data directory
  --formation   formation to query, defaults to 'default'
  --group       group to query formation, defaults to all
  --count       how many events to fetch, defaults to 10
```

  The events are available in the `pgautofailover.event` table in the PostgreSQL instance where the monitor runs, so the `pg_autoctl show events` command needs to be able to connect to the monitor. To this end, the `--pgdata` option is used either to determine a local PostgreSQL instance to connect to, when used on the monitor, or to determine the pg_auto_failover keeper configuration file and read the monitor URI from there.

  See below for more information about `pg_auto_failover` configuration files.

  The options `--formation` and `--group` allow to filter the output to a single formation, and group. The `--count` option limits the output to that many lines.

- `pg_autoctl show state`

  This command outputs the current state of the formation and groups regis-

tered to the pg_auto_failover monitor:

```
$ pg_autoctl show state --help
pg_autoctl show state: Prints monitor's state of nodes in a given␣
 ↪formation and group
usage: pg_autoctl show state  [ --pgdata --formation --group ]

  --pgdata      path to data directory
  --formation   formation to query, defaults to 'default'
  --group       group to query formation, defaults to all
```

For details about the options to the command, see above in the `pg_autoctl show events` command.

- `pg_autoctl show file`

  This command outputs the configuration, state, initial state, and pid files used by this instance. The files are placed in a path that follows the XDG Base Directory Specification[11] and in a way allows to find them when given only `$PGDATA`, as in PostgreSQL:

```
$ pg_autoctl show file --help
pg_autoctl show file: List pg_autoctl internal files (config, state, pid)
usage: pg_autoctl show file  [ --pgdata --all --config | --state | --
 ↪init | --pid --contents ]

  --pgdata      path to data directory
  --all         show all pg_autoctl files
  --config      show pg_autoctl configuration file
  --state       show pg_autoctl state file
  --init        show pg_autoctl initialisation state file
  --pid         show pg_autoctl PID file
  --contents    show selected file contents
```

  The command `pg_auctoctl show file` outputs a JSON object with the single key `config` for a monitor, and with the four keys `config`, `state`, `init`, and `pid` for a keeper. When one of the options with the same name is used, a single line containing only the file path is printed.

  Here's an example of the JSON output:

```
$ pg_autoctl show file --pgdata /data/pgsql
{
  "config": "/Users/dim/.config/pg_autoctl/data/pgsql/pg_autoctl.cfg",
  "state": "/Users/dim/.local/share/pg_autoctl/data/pgsql/pg_autoctl.
 ↪state",
  "init": "/Users/dim/.local/share/pg_autoctl/data/pgsql/pg_autoctl.init
 ↪",
```

(continues on next page)

[11] https://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html

(continued from previous page)

```
  "pid": "/private/tmp/pg_autoctl/data/pgsql/pg_autoctl.pid"
}
```

- `pg_autoctl show systemd`

  This command outputs a configuration unit that is suitable for registering `pg_autoctl` as a systemd service.

7.1.3 pg_auto_failover Postgres Node Initialization

Initializing a pg_auto_failover Postgres node is done with one of the available `pg_autoctl create` commands, depending on which kind of node is to be initialized:

- monitor

  The pg_auto_failover monitor is a special case and has been documented in the previous sections.

- postgres

  The command `pg_autoctl create postgres` initializes a standalone Postgres node to a pg_auto_failover monitor. The monitor is then handling auto-failover for this Postgres node (as soon as a secondary has been registered too, and is known to be healthy).

Here's the full help message for the `pg_autoctl create postgres` command. The other commands accept the same set of options.

```
$ pg_autoctl create postgres --help
pg_autoctl create postgres: Initialize a pg_auto_failover standalone postgres␣
 ↪node
usage: pg_autoctl create postgres

  --pgctl       path to pg_ctl
  --pgdata      path to data director
  --pghost      PostgreSQL's hostname
  --pgport      PostgreSQL's port number
  --listen      PostgreSQL's listen_addresses
  --username    PostgreSQL's username
  --dbname      PostgreSQL's database name
  --hostname    pg_auto_failover node
  --formation   pg_auto_failover formation
  --monitor     pg_auto_failover Monitor Postgres URL
```

(continues on next page)

(continued from previous page)

```
  --auth        authentication method for connections from monitor
  --skip-pg-hba skip editing pg_hba.conf rules
```

Three different modes of initialization are supported by this command, corresponding to as many implementation strategies.

1. Initialize a primary node from scratch

   This happens when `--pgdata` (or the environment variable `PGDATA`) points to an non-existing or empty directory. Then the given `--hostname` is registered to the pg_auto_failover `--monitor` as a member of the `--formation`.

   The monitor answers to the registration call with a state to assign to the new member of the group, either *SINGLE* or *WAIT_STANDBY*. When the assigned state is *SINGLE*, then `pg_autoctl create postgres` proceedes to initialize a new PostgreSQL instance.

2. Initialize an already existing primary server

   This happens when `--pgdata` (or the environment variable `PGDATA`) points to an already existing directory that belongs to a PostgreSQL instance. The standard PostgreSQL tool `pg_controldata` is used to recognize whether the directory belongs to a PostgreSQL instance.

   In that case, the given `--hostname` is registered to the monitor in the tentative *SINGLE* state. When the given `--formation` and `--group` is currently empty, then the monitor accepts the registration and the `pg_autoctl create` prepares the already existing primary server for pg_auto_failover.

3. Initialize a secondary node from scratch

   This happens when `--pgdata` (or the environment variable `PGDATA`) points to a non-existing or empty directory, and when the monitor registration call assigns the state *WAIT_STANDBY* in step 1.

   In that case, the `pg_autoctl create` command steps through the initial states of registering a secondary server, which includes preparing the primary server PostgreSQL HBA rules and creating a replication slot.

   When the command ends successfully, a PostgreSQL secondary server has

been created with `pg_basebackup` and is now started, catching-up to the primary server.

Currently, `pg_autoctl create` doesn't know how to initialize from an already running PostgreSQL standby node. In that situation, it is necessary to prepare a new secondary system from scratch.

When *–hostname* is omitted, it is computed as above (see *pg_auto_failover Monitor*), with the difference that step 1 uses the monitor IP and port rather than the public service 8.8.8.8:53.

The `--auth` option allows setting up authentication method to be used when monitor node makes a connection to data node with *pgautofailover_monitor* user. As with the `pg_autoctl create monitor` command, you could use `--auth trust` when playing with pg_auto_failover at first and consider something production grade later. Also, consider using `--skip-pg-hba` if you already have your own provisioning tools with a security compliance process.

See *Security* for notes on *.pgpass*

7.1.4 pg_autoctl configuration and state files

When initializing a pg_auto_failover keeper service via pg_autoctl, both a configuration file and a state file are created. pg_auto_failover follows the XDG Base Directory Specification[12].

When initializing a pg_auto_failover keeper with `--pgdata /data/pgsql`, then:

- `~/.config/pg_autoctl/data/pgsql/pg_autoctl.cfg`

  is the configuration file for the PostgreSQL instance located at `/data/pgsql`, written in the INI file format.

  It is possible to get the location of the configuration file by using the command `pg_autoctl show file --config --pgdata /data/pgsql` and to output its content by using the command

[12] https://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html

`pg_autoctl show file --config --content --pgdata /data/pgsql`.

Here's an example of such a configuration file:

```
[pg_autoctl]
role = keeper
monitor = postgres://autoctl_node@192.168.1.34:6000/pg_auto_failover
formation = default
group = 1
hostname = node1.db

[postgresql]
pgdata = /data/pgsql/
pg_ctl = /usr/pgsql-10/bin/pg_ctl
dbname = postgres
host = /tmp
port = 5000

[replication]
slot = pgautofailover_standby
maximum_backup_rate = 100M

[timeout]
network_partition_timeout = 20
prepare_promotion_catchup = 30
prepare_promotion_walreceiver = 5
postgresql_restart_failure_timeout = 20
postgresql_restart_failure_max_retries = 3
```

It is possible to edit the configuration file with a tooling of your choice, and with the `pg_autoctl config` subcommands, see below.

- `~/.local/share/pg_autoctl/data/pgsql/pg_autoctl.state`

  is the state file for the pg_auto_failover keeper service taking care of the PostgreSQL instance located at `/data/pgsql`, written in binary format. This file is not intended to be written by anything else than `pg_autoctl` itself. In case of state corruption, see the trouble shooting section of the documentation.

  It is possible to get the location of the state file by using the command `pg_autoctl show file --state --pgdata /data/pgsql` and to output its content by using the command `pg_autoctl show file --state --content --pgdata /data/pgsql`. Here's an example of the output when using that command:

```
$ pg_autoctl show file --state --content --pgdata /data/pgsql
Current Role:             secondary
Assigned Role:            secondary
Last Monitor Contact:     Mon Dec 23 13:31:23 2019
Last Secondary Contact:   0
pg_autoctl state version: 1
group:                    0
node id:                  1
nodes version:            0
PostgreSQL Version:       1100
PostgreSQL CatVersion:    201809051
PostgreSQL System Id:     6772497431723510412
```

- `~/.local/share/pg_autoctl/data/pgsql/pg_autoctl.init`

  is the initial state file for the pg_auto_failover keeper service taking care of the PostgreSQL instance located at `/data/pgsql`, written in binary format. This file is not intended to be written by anything else than `pg_autoctl` itself. In case of state corruption, see the trouble shooting section of the documentation.

  This initialization state file only exists during the initialization of a pg_auto_failover node. In normal operations, this file does not exists.

  It is possible to get the location of the state file by using the command `pg_autoctl show file --init --pgdata /data/pgsql` and to output its content by using the command `pg_autoctl show file --init --content --pgdata /data/pgsql`.

- `/tmp/pg_autoctl/data/pgsql/pg_autoctl.pid`

  is the PID file for the `pg_autoctl` service, located in a temporary directory by default, or in the `XDG_RUNTIME_DIR` directory when this is setup.

  The PID file contains a single line with the PID of the running `pg_autoctl` process, and is supposed to only exists when the process is running. Stale PID files are detected automatically by sending the signal 0 to the PID.

To output, edit and check entries of the configuration, the following commands are provided. Both commands need the *–pgdata* option or the *PGDATA* environment variable to be set in order to find the intended configuration file:

```
pg_autoctl config check [--pgdata <pgdata>]
pg_autoctl config get [--pgdata <pgdata>] section.option
pg_autoctl config set [--pgdata <pgdata>] section.option value
```

7.1.5 Running the pg_auto_failover Keeper service

To run the pg_auto_failover keeper as a background service in your OS, use the following command:

```
$ pg_autoctl run --help
pg_autoctl run: Run the pg_autoctl service (monitor or keeper)
usage: pg_autoctl run  [ --pgdata ]

  --pgdata      path to data directory
```

Thanks to using the XDG Base Directory Specification for our configuration and state file, the only option needed to run the service is `--pgdata`, which defaults to the environment variable `PGDATA`.

7.1.6 Removing a node from the pg_auto_failover monitor

To clean-up an installation and remove a PostgreSQL instance from pg_auto_failover keeper and monitor, use the following command:

```
$ pg_autoctl drop node --help
pg_autoctl drop node: Drop a node from the pg_auto_failover monitor
usage: pg_autoctl drop node [ --pgdata --destroy --hostname --nodeport ]

  --pgdata      path to data directory
  --destroy     also destroy Postgres database
  --hostname    hostname to remove from the monitor
  --nodeport    Postgres port of the node to remove
```

The `pg_autoctl drop node` connects to the monitor and removes the node from it, then removes the local pg_auto_failover keeper state file. The configuration file is not removed.

It is possible to run the `pg_autoctl drop node` command either from the node itself and then the `--destroy` option is available to wipe out everything, including configuration files and PGDATA; or to run the command from the

monitor and then use the `--hostname` and `--nodeport` options to target a (presumably dead) node to remove from the monitor registration.

7.2 pg_autoctl do

When testing pg_auto_failover, it is helpful to be able to play with the local nodes using the same lower-level API as used by the pg_auto_failover Finite State Machine transitions. The low-level API is made available through the following commands, only available in debug environments:

```
$ PG_AUTOCTL_DEBUG=1 pg_autoctl help
  pg_autoctl
  + create    Create a pg_auto_failover node, or formation
  + drop      Drop a pg_auto_failover node, or formation
  + config    Manages the pg_autoctl configuration
  + show      Show pg_auto_failover information
  + enable    Enable a feature on a formation
  + disable   Disable a feature on a formation
  + do        Manually operate the keeper
    run       Run the pg_autoctl service (monitor or keeper)
    stop      signal the pg_autoctl service for it to stop
    reload    signal the pg_autoctl for it to reload its configuration
    help      print help message
    version   print pg_autoctl version

  pg_autoctl create
    monitor      Initialize a pg_auto_failover monitor node
    postgres     Initialize a pg_auto_failover standalone postgres node
    formation    Create a new formation on the pg_auto_failover monitor

  pg_autoctl drop
    node       Drop a node from the pg_auto_failover monitor
    formation  Drop a formation on the pg_auto_failover monitor

  pg_autoctl config
    check  Check pg_autoctl configuration
    get    Get the value of a given pg_autoctl configuration variable
    set    Set the value of a given pg_autoctl configuration variable

  pg_autoctl show
    uri     Show the postgres uri to use to connect to pg_auto_failover nodes
    events  Prints monitor's state of nodes in a given formation and group
    state   Prints monitor's state of nodes in a given formation and group

  pg_autoctl enable
    secondary    Enable secondary nodes on a formation
    maintenance  Enable Postgres maintenance mode on this node
    ssl          Enable SSL configuration on this node
```

(continues on next page)

(continued from previous page)

```
  pg_autoctl disable
    secondary    Disable secondary nodes on a formation
    maintenance  Disable Postgres maintenance mode on this node
    ssl          Disable SSL configuration on this node

  pg_autoctl do
  + monitor      Query a pg_auto_failover monitor
  + fsm          Manually manage the keeper's state
  + primary      Manage a PostgreSQL primary server
  + standby      Manage a PostgreSQL standby server
    discover     Discover local PostgreSQL instance, if any

  pg_autoctl do monitor
  + get       Get information from the monitor
    register  Register the current node with the monitor
    active    Call in the pg_auto_failover Node Active protocol
    version   Check that monitor version is 1.0; alter extension update if not

  pg_autoctl do monitor get
    primary      Get the primary node from pg_auto_failover in given formation/
↪group
    other        Get the other node from the pg_auto_failover group of hostname/
↪port
    coordinator  Get the coordinator node from the pg_auto_failover formation

  pg_autoctl do fsm
    init    Initialize the keeper's state on-disk
    state   Read the keeper's state from disk and display it
    list    List reachable FSM states from current state
    gv      Output the FSM as a .gv program suitable for graphviz/dot
    assign  Assign a new goal state to the keeper
    step    Make a state transition if instructed by the monitor

  pg_autoctl do primary
  + slot      Manage replication slot on the primary server
  + syncrep   Manage the synchronous replication setting on the primary server
    defaults  Add default settings to postgresql.conf
  + adduser   Create users on primary
  + hba       Manage pg_hba settings on the primary server

  pg_autoctl do primary slot
    create  Create a replication slot on the primary server
    drop    Drop a replication slot on the primary server

  pg_autoctl do primary syncrep
    enable   Enable synchronous replication on the primary server
    disable  Disable synchronous replication on the primary server

  pg_autoctl do primary adduser
    monitor  add a local user for queries from the monitor
    replica  add a local user with replication privileges

  pg_autoctl do primary hba
    setup  Make sure the standby has replication access in pg_hba
```

(continues on next page)

(continued from previous page)

```
pg_autoctl do standby
  init     Initialize the standby server using pg_basebackup
  rewind   Rewind a demoted primary server using pg_rewind
  promote  Promote a standby server to become writable

pg_autoctl do show
  ipaddr    Print this node's IP address information
  cidr      Print this node's CIDR information
  lookup    Print this node's DNS lookup information
  hostname  Print this node's default hostname
```

CHAPTER 8

Configuring pg_auto_failover

Several defaults settings of pg_auto_failover can be reviewed and changed depending on the trade-offs you want to implement in your own production setup. The settings that you can change will have an impact of the following operations:

- Deciding when to promote the secondary

  pg_auto_failover decides to implement a failover to the secondary node when it detects that the primary node is unhealthy. Changing the following settings will have an impact on when the pg_auto_failover monitor decides to promote the secondary PostgreSQL node:

  ```
  pgautofailover.health_check_max_retries
  pgautofailover.health_check_period
  pgautofailover.health_check_retry_delay
  pgautofailover.health_check_timeout
  pgautofailover.node_considered_unhealthy_timeout
  ```

- Time taken to promote the secondary

  At secondary promotion time, pg_auto_failover waits for the following timeout to make sure that all pending writes on the primary server made it to the secondary at shutdown time, thus preventing data loss.:

```
pgautofailover.primary_demote_timeout
```

- Preventing promotion of the secondary

  pg_auto_failover implements a trade-off where data availability trumps service availability. When the primary node of a PostgreSQL service is detected unhealthy, the secondary is only promoted if it was known to be eligible at the moment when the primary is lost.

  In the case when *synchronous replication* was in use at the moment when the primary node is lost, then we know we can switch to the secondary safely, and the wal lag is 0 in that case.

  In the case when the secondary server had been detected unhealthy before, then the pg_auto_failover monitor switches it from the state SECONDARY to the state CATCHING-UP and promotion is prevented then.

  The following setting allows to still promote the secondary, allowing for a window of data loss:

  ```
  pgautofailover.promote_wal_log_threshold
  ```

8.1 pg_auto_failover Monitor

The configuration for the behavior of the monitor happens in the PostgreSQL database where the extension has been deployed:

```
pg_auto_failover=> select name, setting, unit, short_desc from pg_settings␣
 ↪where name ~ 'pgautofailover.';
-[ RECORD 1 ]----------------------------------------------------------------
 ↪--------------------------------
name       | pgautofailover.enable_sync_wal_log_threshold
setting    | 16777216
unit       |
short_desc | Don't enable synchronous replication until secondary xlog is␣
 ↪within this many bytes of the primary's
-[ RECORD 2 ]----------------------------------------------------------------
 ↪--------------------------------
name       | pgautofailover.health_check_max_retries
setting    | 2
unit       |
short_desc | Maximum number of re-tries before marking a node as failed.
-[ RECORD 3 ]----------------------------------------------------------------
 ↪--------------------------------
```

(continues on next page)

(continued from previous page)

```
name        | pgautofailover.health_check_period
setting     | 5000
unit        | ms
short_desc | Duration between each check (in milliseconds).
-[ RECORD 4 ]-------------------------------------------------------------------
 ↪---------------------------------
name        | pgautofailover.health_check_retry_delay
setting     | 2000
unit        | ms
short_desc | Delay between consecutive retries.
-[ RECORD 5 ]-------------------------------------------------------------------
 ↪---------------------------------
name        | pgautofailover.health_check_timeout
setting     | 5000
unit        | ms
short_desc | Connect timeout (in milliseconds).
-[ RECORD 6 ]-------------------------------------------------------------------
 ↪---------------------------------
name        | pgautofailover.node_considered_unhealthy_timeout
setting     | 20000
unit        | ms
short_desc | Mark node unhealthy if last ping was over this long ago
-[ RECORD 7 ]-------------------------------------------------------------------
 ↪---------------------------------
name        | pgautofailover.primary_demote_timeout
setting     | 30000
unit        | ms
short_desc | Give the primary this long to drain before promoting the secondary
-[ RECORD 8 ]-------------------------------------------------------------------
 ↪---------------------------------
name        | pgautofailover.promote_wal_log_threshold
setting     | 16777216
unit        |
short_desc | Don't promote secondary unless xlog is with this many bytes of␣
 ↪the master
-[ RECORD 9 ]-------------------------------------------------------------------
 ↪---------------------------------
name        | pgautofailover.startup_grace_period
setting     | 10000
unit        | ms
short_desc | Wait for at least this much time after startup before initiating␣
 ↪a failover.
```

You can edit the parameters as usual with PostgreSQL, either in the `postgresql.conf` file or using `ALTER DATABASE pg_auto_failover SET parameter = value;` commands, then issuing a reload.

8.2 pg_auto_failover Keeper Service

For an introduction to the `pg_autoctl` commands relevant to the pg_auto_failover Keeper configuration, please see *pg_autoctl configuration and state files*.

An example configuration file looks like the following:

```
[pg_autoctl]
role = keeper
monitor = postgres://autoctl_node@192.168.1.34:6000/pg_auto_failover
formation = default
group = 0
hostname = node1.db
nodekind = standalone

[postgresql]
pgdata = /data/pgsql/
pg_ctl = /usr/pgsql-10/bin/pg_ctl
dbname = postgres
host = /tmp
port = 5000

[replication]
slot = pgautofailover_standby
maximum_backup_rate = 100M
backup_directory = /data/backup/node1.db

[timeout]
network_partition_timeout = 20
postgresql_restart_failure_timeout = 20
postgresql_restart_failure_max_retries = 3
```

To output, edit and check entries of the configuration, the following commands are provided:

```
pg_autoctl config check [--pgdata <pgdata>]
pg_autoctl config get [--pgdata <pgdata>] section.option
pg_autoctl config set [--pgdata <pgdata>] section.option value
```

The `[postgresql]` section is discovered automatically by the `pg_autoctl` command and is not intended to be changed manually.

pg_autoctl.monitor

PostgreSQL service URL of the pg_auto_failover monitor, as given in the output of the `pg_autoctl show uri` command.

pg_autoctl.formation

A single pg_auto_failover monitor may handle several postgres formations. The default formation name *default* is usually fine.

pg_autoctl.group

This information is retrieved by the pg_auto_failover keeper when registering a node to the monitor, and should not be changed afterwards. Use at your own risk.

pg_autoctl.hostname

Node *hostname* used by all the other nodes in the cluster to contact this node. In particular, if this node is a primary then its standby uses that address to setup streaming replication.

replication.slot

Name of the PostgreSQL replication slot used in the streaming replication setup automatically deployed by pg_auto_failover. Replication slots can't be renamed in PostgreSQL.

replication.maximum_backup_rate

When pg_auto_failover (re-)builds a standby node using the `pg_basebackup` command, this parameter is given to `pg_basebackup` to throttle the network bandwidth used. Defaults to 100Mbps.

replication.backup_directory

When pg_auto_failover (re-)builds a standby node using the `pg_basebackup` command, this parameter is the target directory where to copy the bits from the primary server. When the copy has been successful, then the directory is renamed to **postgresql.pgdata**.

The default value is computed from `${PGDATA}/../backup/${hostname}` and can be set to any value of your preference. Remember that the directory renaming is an atomic operation only when both the source and the target of the copy are in the same filesystem, at least in Unix systems.

timeout

This section allows to setup the behavior of the pg_auto_failover keeper in interesting scenarios.

timeout.network_partition_timeout

Timeout in seconds before we consider failure to communicate with other nodes indicates a network partition. This check is only done on a PRIMARY server, so other nodes mean both the monitor and the standby.

When a PRIMARY node is detected to be on the losing side of a network partition, the pg_auto_failover keeper enters the DEMOTE state and stops the PostgreSQL instance in order to protect against split brain situations.

The default is 20s.

timeout.postgresql_restart_failure_timeout

timeout.postgresql_restart_failure_max_retries

When PostgreSQL is not running, the first thing the pg_auto_failover keeper does is try to restart it. In case of a transient failure (e.g. file system is full, or other dynamic OS resource constraint), the best course of action is to try again for a little while before reaching out to the monitor and ask for a failover.

The pg_auto_failover keeper tries to restart PostgreSQL `timeout.postgresql_restart_failure_max_retries` times in a row (default 3) or up to `timeout.postgresql_restart_failure_timeout` (defaults 20s) since it detected that PostgreSQL is not running, whichever comes first.

CHAPTER 9

Operating pg_auto_failover

This section is not yet complete. Please contact us with any questions.

9.1 Deployment

pg_auto_failover is a general purpose tool for setting up PostgreSQL replication in order to implement High Availability of the PostgreSQL service.

9.2 Provisioning

It is also possible to register pre-existing PostgreSQL instances with a pg_auto_failover monitor. The `pg_autoctl create` command honors the `PGDATA` environment variable, and checks whether PostgreSQL is already running. If Postgres is detected, the new node is registered in SINGLE mode, bypassing the monitor's role assignment policy.

9.3 Security

Connections between monitor and data nodes use *trust* authentication by default. This lets accounts used by `pg_auto_failover` to connect to nodes without needing a password. Default behaviour could be changed using `--auth` parameter when creating monitor or data Node. Any auth method supported by PostgreSQL could be used here. Please refer to PostgreSQL pg_hba documentation[13] for available options.

Security for following connections should be considered when setting up *.pgpass* file.

1. health check connection from monitor for *autoctl* user to both *postgres* and *pg_auto_failover* databases.
2. connections for *pg_autoctl* command from data nodes to monitor for *autoctl_node* user.
3. replication connections from secondary to primary data nodes for *replication* user. Notice that primary and secondary nodes change during failover. Thus this setting should be done on both primary and secondary nodes.
4. settings need to be updated after a new node is added.

See PostgreSQL documentation[14] on setting up *.pgpass* file.

9.4 Operations

It is possible to operate pg_auto_failover formations and groups directly from the monitor. All that is needed is an access to the monitor Postgres database as a client, such as `psql`. It's also possible to add those management SQL function calls in your own ops application if you have one.

For security reasons, the `autoctl_node` is not allowed to perform maintenance operations. This user is limited to what `pg_autoctl` needs. You can either create a specific user and authentication rule to expose for management,

[13] https://www.postgresql.org/docs/current/auth-pg-hba-conf.html
[14] https://www.postgresql.org/docs/current/libpq-pgpass.html

or edit the default HBA rules for the `autoctl` user. In the following examples we're directly connecting as the `autoctl` role.

The main operations with pg_auto_failover are node maintenance and manual failover, also known as a controlled switchover.

9.4.1 Maintenance of a secondary node

It is possible to put a secondary node in any group in a MAINTENANCE state, so that the Postgres server is not doing *synchronous replication* anymore and can be taken down for maintenance purposes, such as security kernel upgrades or the like.

The command line tool `pg_autoctl` exposes an API to schedule maintenance operations on the current node, which must be a secondary node at the moment when maintenance is requested.

Here's an example of using the maintenance commands on a secondary node, including the output. Of course, when you try that on your own nodes, dates and PID information might differ:

```
$ pg_autoctl enable maintenance
17:49:19 14377 INFO  Listening monitor notifications about state changes in␣
 ↪formation "default" and group 0
17:49:19 14377 INFO  Following table displays times when notifications are␣
 ↪received
    Time |  ID |      Host |   Port |       Current State |      Assigned State
---------+-----+-----------+--------+---------------------+--------------------
17:49:19 |   1 | localhost |   5001 |             primary |         wait_primary
17:49:19 |   2 | localhost |   5002 |           secondary |    wait_maintenance
17:49:19 |   2 | localhost |   5002 |    wait_maintenance |    wait_maintenance
17:49:20 |   1 | localhost |   5001 |        wait_primary |         wait_primary
17:49:20 |   2 | localhost |   5002 |    wait_maintenance |          maintenance
17:49:20 |   2 | localhost |   5002 |         maintenance |          maintenance
```

The command listens to the state changes in the current node's formation and group on the monitor and displays those changes as it receives them. The operation is done when the node has reached the `maintenance` state.

It is now possible to disable maintenance to allow `pg_autoctl` to manage this standby node again:

```
$ pg_autoctl disable maintenance
17:49:26 14437 INFO  Listening monitor notifications about state changes in␣
 ↪formation "default" and group 0
```

(continues on next page)

(continued from previous page)

```
17:49:26 14437 INFO  Following table displays times when notifications are␣
 ↪received
    Time |  ID |      Host |   Port |       Current State |      Assigned State
---------+-----+-----------+--------+---------------------+--------------------
17:49:27 |   2 | localhost |   5002 |         maintenance |          catchingup
17:49:27 |   2 | localhost |   5002 |          catchingup |          catchingup
17:49:28 |   2 | localhost |   5002 |          catchingup |            secondary
17:49:28 |   1 | localhost |   5001 |       wait_primary |              primary
17:49:28 |   2 | localhost |   5002 |           secondary |            secondary
17:49:29 |   1 | localhost |   5001 |             primary |              primary
```

When a standby node is in maintenance, the monitor sets the primary node replication to WAIT_PRIMARY: in this role, the PostgreSQL streaming replication is now asynchronous and the standby PostgreSQL server may be stopped, rebooted, etc.

9.4.2 Maintenance of a primary node

A primary node must be available at all times in any formation and group in pg_auto_failover, that is the invariant provided by the whole solution. With that in mind, the only way to allow a primary node to go to a maintenance mode is to first failover and promote the secondary node.

The same command `pg_autoctl enable maintenance` implements that operation when run on a primary node with the option `--allow-failover`. Here is an example of such an operation:

```
$ pg_autoctl enable maintenance
11:53:03 50526 WARN  Enabling maintenance on a primary causes a failover
11:53:03 50526 FATAL Please use --allow-failover to allow the command proceed
```

As we can see the option `allow-maintenance` is mandatory. In the next example we use it:

```
$ pg_autoctl enable maintenance --allow-failover
13:13:42 1614 INFO  Listening monitor notifications about state changes in␣
 ↪formation "default" and group 0
13:13:42 1614 INFO  Following table displays times when notifications are␣
 ↪received
    Time |  ID |      Host |   Port |       Current State |      Assigned State
---------+-----+-----------+--------+---------------------+--------------------
13:13:43 |   2 | localhost |   5002 |             primary | prepare_maintenance
13:13:43 |   1 | localhost |   5001 |           secondary |   prepare_promotion
13:13:43 |   1 | localhost |   5001 |   prepare_promotion |   prepare_promotion
```

(continues on next page)

(continued from previous page)

```
13:13:43 |   2 | localhost |   5002 | prepare_maintenance | prepare_maintenance
13:13:44 |   1 | localhost |   5001 |   prepare_promotion |    stop_replication
13:13:45 |   1 | localhost |   5001 |    stop_replication |    stop_replication
13:13:46 |   1 | localhost |   5001 |    stop_replication |        wait_primary
13:13:46 |   2 | localhost |   5002 | prepare_maintenance |         maintenance
13:13:46 |   1 | localhost |   5001 |        wait_primary |        wait_primary
13:13:47 |   2 | localhost |   5002 |         maintenance |         maintenance
```

When the operation is done we can have the old primary re-join the group, this time as a secondary:

```
$ pg_autoctl disable maintenance
13:14:46 1985 INFO  Listening monitor notifications about state changes in␣
 ↪formation "default" and group 0
13:14:46 1985 INFO  Following table displays times when notifications are␣
 ↪received
    Time |  ID |      Host |   Port |       Current State |      Assigned State
---------+-----+-----------+--------+---------------------+--------------------
13:14:47 |   2 | localhost |   5002 |         maintenance |          catchingup
13:14:47 |   2 | localhost |   5002 |          catchingup |          catchingup
13:14:52 |   2 | localhost |   5002 |          catchingup |           secondary
13:14:52 |   1 | localhost |   5001 |        wait_primary |             primary
13:14:52 |   2 | localhost |   5002 |           secondary |           secondary
13:14:53 |   1 | localhost |   5001 |             primary |             primary
```

9.4.3 Triggering a failover

It is possible to trigger a manual failover, or a switchover, using the command `pg_autoctl perform failover`. Here's an example of what happens when running the command:

```
$ pg_autoctl perform failover
11:58:00 53224 INFO  Listening monitor notifications about state changes in␣
 ↪formation "default" and group 0
11:58:00 53224 INFO  Following table displays times when notifications are␣
 ↪received
    Time |  ID |      Host |   Port |      Current State |     Assigned State
---------+-----+-----------+--------+--------------------+-------------------
11:58:01 |   1 | localhost |   5001 |            primary |          draining
11:58:01 |   2 | localhost |   5002 |          secondary | prepare_promotion
11:58:01 |   1 | localhost |   5001 |           draining |          draining
11:58:01 |   2 | localhost |   5002 | prepare_promotion | prepare_promotion
11:58:02 |   2 | localhost |   5002 | prepare_promotion |  stop_replication
11:58:02 |   1 | localhost |   5001 |           draining |    demote_timeout
11:58:03 |   1 | localhost |   5001 |    demote_timeout |    demote_timeout
11:58:04 |   2 | localhost |   5002 |  stop_replication |  stop_replication
11:58:05 |   2 | localhost |   5002 |  stop_replication |       wait_primary
11:58:05 |   1 | localhost |   5001 |    demote_timeout |            demoted
```

(continues on next page)

(continued from previous page)

```
11:58:05 |   2 | localhost |   5002 |       wait_primary |       wait_primary
11:58:05 |   1 | localhost |   5001 |            demoted |            demoted
11:58:06 |   1 | localhost |   5001 |            demoted |         catchingup
11:58:06 |   1 | localhost |   5001 |         catchingup |         catchingup
11:58:08 |   1 | localhost |   5001 |         catchingup |          secondary
11:58:08 |   2 | localhost |   5002 |       wait_primary |            primary
11:58:08 |   1 | localhost |   5001 |          secondary |          secondary
11:58:08 |   2 | localhost |   5002 |            primary |            primary
```

Again, timings and PID numbers are not expected to be the same when you run the command on your own setup.

Also note in the output that the command shows the whole set of transitions including when the old primary is now a secondary node. The database is available for read-write traffic as soon as we reach the state `wait_primary`.

9.4.4 Implementing a controlled switchover

It is generally useful to distinguish a *controlled switchover* to a *failover*. In a controlled switchover situation it is possible to organise the sequence of events in a way to avoid data loss and lower downtime to a minimum.

In the case of pg_auto_failover, because we use **synchronous replication**, we don't face data loss risks when triggering a manual failover. Moreover, our monitor knows the current primary health at the time when the failover is triggered, and drives the failover accordingly.

So to trigger a controlled switchover with pg_auto_failover you can use the same API as for a manual failover:

```
$ pg_autoctl perform switchover
```

Because the subtelties of orchestrating either a controlled switchover or an unplanned failover are all handled by the monitor, rather than the client side command line, at the client level the two command `pg_autoctl perform failover` and `pg_autoctl perform switchover` are synonyms, or aliases.

9.5 Current state, last events

The following commands display information from the pg_auto_failover monitor tables `pgautofailover.node` and `pgautofailover.event`:

```
$ pg_autoctl show state
$ pg_autoctl show events
```

When run on the monitor, the commands outputs all the known states and events for the whole set of formations handled by the monitor. When run on a PostgreSQL node, the command connects to the monitor and outputs the information relevant to the service group of the local node only.

For interactive debugging it is helpful to run the following command from the monitor node while e.g. initializing a formation from scratch, or performing a manual failover:

```
$ watch pg_autoctl show state
```

9.6 Monitoring pg_auto_failover in Production

The monitor reports every state change decision to a LISTEN/NOTIFY channel named `state`. PostgreSQL logs on the monitor are also stored in a table, `pgautofailover.event`, and broadcast by NOTIFY in the channel `log`.

9.7 Trouble-Shooting Guide

pg_auto_failover commands can be run repeatedly. If initialization fails the first time – for instance because a firewall rule hasn't yet activated – it's possible to try `pg_autoctl create` again. pg_auto_failover will review its previous progress and repeat idempotent operations (`create database`, `create extension` etc), gracefully handling errors.

CHAPTER 10

The pg_auto_failover Finite State Machine

10.1 Introduction

pg_auto_failover uses a state machine for highly controlled execution. As keepers inform the monitor about new events (or fail to contact it at all), the monitor assigns each node both a current state and a goal state. A node's current state is a strong guarantee of its capabilities. States themselves do not cause any actions; actions happen during state transitions. The assigned goal states inform keepers of what transitions to attempt.

10.2 Example of state transitions in a new cluster

A good way to get acquainted with the states is by examining the transitions of a cluster from birth to high availability.

After starting a monitor and running keeper init for the first data node ("node A"), the monitor registers the state of that node as "init" with a goal state of "single." The init state means the monitor knows nothing about the node other than its existence because the keeper is not yet continuously running there to report node health.

Once the keeper runs and reports its health to the monitor, the monitor assigns it the state "single," meaning it is just an ordinary Postgres server with no failover. Because there are not yet other nodes in the cluster, the monitor also assigns node A the goal state of single – there's nothing that node A's keeper needs to change.

As soon as a new node ("node B") is initialized, the monitor assigns node A the goal state of "wait_primary." This means the node still has no failover, but there's hope for a secondary to synchronize with it soon. To accomplish the transition from single to wait_primary, node A's keeper adds node B's hostname to pg_hba.conf to allow a hot standby replication connection.

At the same time, node B transitions into wait_standby with the goal initially of staying in wait_standby. It can do nothing but wait until node A gives it access to connect. Once node A has transitioned to wait_primary, the monitor assigns B the goal of "catchingup," which gives B's keeper the green light to make the transition from wait_standby to catchingup. This transition involves running pg_basebackup, editing recovery.conf and restarting PostgreSQL in Hot Standby node.

Node B reports to the monitor when it's in hot standby mode and able to connect to node A. The monitor then assigns node B the goal state of "secondary" and A the goal of "primary." Postgres ships WAL logs from node A and replays them on B. Finally B is caught up and tells the monitor (specifically B reports its pg_stat_replication.sync_state and WAL replay lag). At this glorious moment the monitor assigns A the state primary (goal: primary) and B secondary (goal: secondary).

10.3 State reference

For a graph of the states and their transitions, see *pg_auto_failover keeper's State Machine*.

Init

A node is assigned the "init" state when it is first registered with the monitor. Nothing is known about the node at this point beyond its existence. If no other node has been registered with the monitor for the same formation and group ID then this node is assigned a goal state of "single." Otherwise the node has the goal state of "wait_standby."

Single

There is only one node in the group. It behaves as a regular PostgreSQL instance, with no high availability and no failover. If the administrator removes a node the other node will revert to the single state.

Wait_primary

Applied to a node intended to be the primary but not yet in that position. The primary-to-be at this point knows the secondary's node name or IP address, and has granted the node hot standby access in the pg_hba.conf file.

The wait_primary state may be caused either by a new potential secondary being registered with the monitor (good), or an existing secondary becoming unhealthy (bad). In the latter case, during the transition from primary to wait_primary, the primary node's keeper disables synchronous replication on the node. It also cancels currently blocked queries.

Primary

A healthy secondary node exists and has caught up with WAL replication. Specifically, the keeper reports the primary state only when it has verified that the secondary is reported "sync" in pg_stat_replication.sync_state, and with a WAL lag of 0.

The primary state is a strong assurance. It's the only state where we know we can fail over when required.

During the transition from wait_primary to primary, the keeper also enables synchronous replication. This means that after a failover the secondary will be fully up to date.

Wait_standby

Monitor decides this node is a standby. Node must wait until the primary has authorized it to connect and setup hot standby replication.

Catchingup

The monitor assigns catchingup to the standby node when the primary is ready for a replication connection (pg_hba.conf has been properly edited, connection role added, etc).

The standby node keeper runs pg_basebackup, connecting to the primary's hostname and port. The keeper then edits recovery.conf and starts PostgreSQL in hot standby node.

Secondary

A node with this state is acting as a hot standby for the primary, and is up to date with the WAL log there. In particular, it is within 16MB or 1 WAL segment of the primary.

Maintenance

The cluster administrator can manually move a secondary into the maintenance state to gracefully take it offline. The primary will then transition from state primary to wait_primary, during which time the secondary will be online to accept writes. When the old primary reaches the wait_primary state then the secondary is safe to take offline with minimal consequences.

Prepare_maintenance

The cluster administrator can manually move a primary node into the maintenance state to gracefully take it offline. The primary then transitions to the prepare_maintenance state to make sure the secondary is not missing any writes. In the prepare_maintenance state, the primary shuts down.

Wait_maintenance

The custer administrator can manually move a secondary into the maintenance state to gracefully take it offline. Before reaching the maintenance state though, we want to switch the primary node to asynchronous replication, in order to avoid writes being blocked. In the state wait_maintenance the standby waits until the primary has reached wait_primary.

Draining

A state between primary and demoted where replication buffers finish flushing. A draining node will not accept new client writes, but will continue to send existing data to the secondary.

Demoted

The primary keeper or its database were unresponsive past a certain threshold. The monitor assigns demoted state to the primary to avoid a split-brain scenario where there might be two nodes that don't communicate with each other and both accept client writes.

In that state the keeper stops PostgreSQL and prevents it from running.

Demote_timeout

If the monitor assigns the primary a demoted goal state but the primary keeper doesn't acknowledge transitioning to that state within a timeout window, then the monitor assigns demote_timeout to the primary.

Most commonly may happen when the primary machine goes silent. The keeper is not reporting to the monitor.

Stop_replication

The stop_replication state is meant to ensure that the primary goes to the demoted state before the standby goes to single and accepts writes (in case the primary can't contact the monitor anymore). Before promoting the secondary node, the keeper stops PostgreSQL on the primary to avoid split-brain situations.

For safety, when the primary fails to contact the monitor and fails to see the pg_auto_failover connection in pg_stat_replication, then it goes to the demoted state of its own accord.

Prepare_promotion

The prepare_promotion state is meant to prepare the standby server to being promoted. This state allows synchronisation on the monitor, making sure that the primary has stopped Postgres before promoting the secondary, hence preventing split brain situations.