Chapter 15. Load Balancing MySQL

There are different ways to connect to MySQL. For example, to perform a write test, a connection is created, the statement is executed, and then the connection is closed. To avoid the cost of opening a connection every time it is needed, the concept of the *connection pool* was developed. Connection pooling is a technique of creating and managing a pool of connections that are ready for use by any thread of the application.

Extending the concept of high availability discussed in Chapter 13 to connections in order to improve a production system's resilience, it is possible to use *load balancers* to connect to a database cluster. With load balancing and MySQL high availability, it is possible to keep the application running without interruption (or with only minor downtime). Basically, if the source server or one of the nodes of the database cluster fails, the client just needs to connect to another database node and it can continue to serve requests.

Load balancers were built to provide transparency for clients when connecting to MySQL infrastructure. In this way, the application does not need to be aware of the MySQL topology; whether you're using a classic replication, Group Replication, or Galera Cluster does not matter. The load balancer will provide an online node where it will be possible to read and write queries. Having a robust MySQL architecture and a proper load balancer in place can help DBAs avoid sleepless nights.

Load Balancing with Application Drivers

To connect an application to MySQL, you need a driver. A *driver* is an adapter used to connect the application to a different system type. It is similar to connecting a video card to your computer; you may need to download and install a driver for it to work with your application.

Modern MySQL drivers from commonly used programming languages support connection pooling, load balancing, and failover. Examples include the JDBC driver for MySQL (MySQL Connector/J) and the PDO_MYSQL

driver, which implements the PHP Data Objects (PDO) interface to enable access from PHP to MySQL databases.

The database drivers we've mentioned are built to provide transparency for clients when connecting to standalone MySQL Server or MySQL replication setups. We won't show you how to use them in code because that would be outside the scope of this book; however, you should be aware that adding a driver library facilitates code development, since the driver abstracts away a substantial amount of work for the developer.

But for other topologies, such as a clustering setup like Galera Cluster for MySQL or MariaDB, the JDBC and PHP drivers are not aware of internal Galera state information. For instance, a Galera donor node might be in read-only mode while it is helping another node resynchronize (if the SST method is `mysqldump` or `rsync`), or it could be up in non-primary state if split-brain happens. Another solution is to use a load balancer between the clients and the database cluster.

ProxySQL Load Balancer

ProxySQL is a SQL proxy. ProxySQL implements the MySQL protocol, and because of this, it can do things that other proxies cannot do. Here are some of its advantages:

- It provides "intelligent" load balancing of application requests to multiple databases.
- It understands the MySQL traffic that passes through it and can split reads from writes. Understanding the MySQL protocol is especially useful in a source/replica replication setup, where writes should only go to the source and reads to the replicas, or in the case of Galera Cluster for distributing the read queries evenly (linear read scaling).
- It understands the underlying database topology, including whether the instances are up or down, and therefore can route requests to healthy databases.
- It provides query workload analytics and a query cache, which is useful for analyzing and improving performance.
- It provides administrators with robust, rich query rule definitions to efficiently distribute queries and cache data to maximize the database service's efficiency.

ProxySQL runs as a daemon watched by a monitoring process. The process monitors the daemon and restarts it in case of a crash to minimize downtime. The daemon accepts incoming traffic from MySQL clients and forwards it to backend MySQL servers.

The proxy is designed to run continuously without needing to be restarted. Most configurations can be done at runtime using queries similar to SQL statements in the ProxySQL admin interface. These include runtime parameters, server grouping, and traffic-related settings.

While it is common to install ProxySQL on a standalone node between the application and the database, this can affect query performance due to the additional latency from network hops. Figure 15-1 shows ProxySQL as a middle layer.

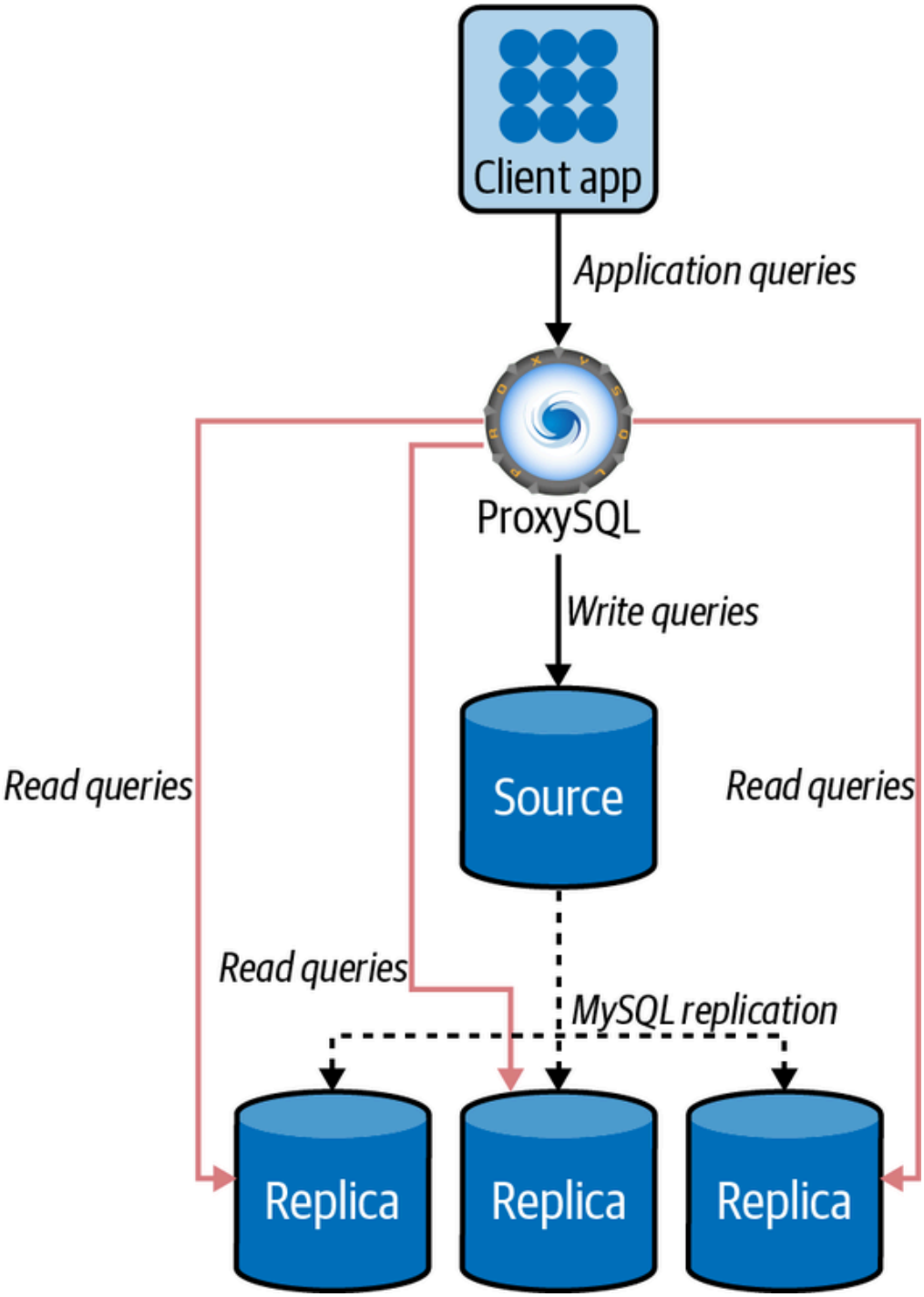

Figure 15-1. ProxySQL between the application and MySQL

To reduce the impact on performance (and avoid the additional network hop), another architecture option is installing ProxySQL on the application servers. The application then connects to ProxySQL (acting as a MySQL server) on localhost using a Unix domain socket, avoiding extra latency. It uses its routing rules to reach out and talk to the actual MySQL servers with their connection pooling. The application doesn't have any idea what happens beyond its connection to ProxySQL. Figure 15-2 shows ProxySQL on the same server as the application.

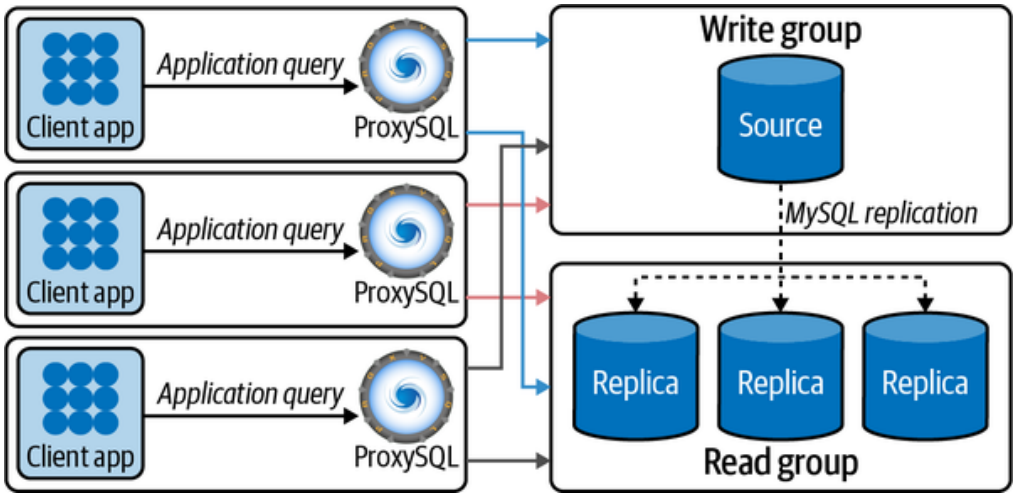

Figure 15-2. ProxySQL on the same server as the application

Installing and Configuring ProxySQL

Let's take a look at how to deploy ProxySQL for a source/replica configuration.

The tool's developers provide official packages for a variety of Linux distributions for all ProxySQL releases on their GitHub releases page, so we'll download the latest package version from there and install it.

Before installing, the following instances are the ones we will use in this process:

```
+--------------------------------------+-------------
| vinicius-grippa-default(mysql)       | 10.124.33.5 (
+--------------------------------------+-------------
| vinicius-grippa-node1(mysql)         | 10.124.33.169
+--------------------------------------+-------------
| vinicius-grippa-node2(mysql)         | 10.124.33.130
+--------------------------------------+-------------
| vinicius-grippa-node3(proxysql)      | 10.124.33.170
+--------------------------------------+-------------
```

To begin, find the proper distribution for your operating system. In this example, we will then install for CentOS 7. First, we will become root, install the MySQL client to connect to ProxySQL, and install ProxySQL itself. We get the URL from the downloads page and refer it to `yum`:

```
$ sudo su - root
# yum -y install https://repo.percona.com/yum/percona-r
```

```
# yum -y install Percona-Server-client-57
# yum install -y https://github.com/sysown/proxysql/rel
    proxysql-2.0.15-1-centos7.x86_64.rpm
```

We have all the requirements to run ProxySQL, but the service doesn't automatically start after installation, so we start it manually:

```
# sudo systemctl start proxysql
```

ProxySQL should now be running with its default configuration in place. We can check it by running this command:

```
# systemctl status proxysql
```

The output of the ProxySQL process in the active state should be similar to the following:

```
 proxysql.service - High Performance Advanced Proxy for
   Loaded: loaded (/etc/systemd/system/proxysql.service
   Active: active (running) since Sun 2021-05-23 18:50:
  Process: 1422 ExecStart=/usr/bin/proxysql --idle-thre
 Main PID: 1425 (proxysql)
   CGroup: /system.slice/proxysql.service
           ├─1425 /usr/bin/proxysql --idle-threads -c /
           └─1426 /usr/bin/proxysql --idle-threads -c /

May 23 18:50:27 vinicius-grippa-node3 systemd[1]: Start
May 23 18:50:27 vinicius-grippa-node3 proxysql[1422]: 2
May 23 18:50:27 vinicius-grippa-node3 proxysql[1422]: 2
May 23 18:50:27 vinicius-grippa-node3 proxysql[1422]: 2
May 23 18:50:28 vinicius-grippa-node3 systemd[1]: Start
```

ProxySQL splits the application interface from the admin interface. This means that ProxySQL will listen on two network ports: the admin interface will listen on 6032, and the application will listen on 6033 (to make it easier to remember, that's the reverse of MySQL's default port, 3306).

Next, ProxySQL needs to communicate with the MySQL nodes to be able to check their condition. To achieve this, ProxySQL needs to connect to each server with a dedicated user.

First, we are going to create the user on the source server. Connect to the MySQL source instance and run these commands:

```
mysql> CREATE USER 'proxysql'@'%' IDENTIFIED by '$3Kr$t
mysql> GRANT USAGE ON *.* TO 'proxysql'@'%';
```

Next, we will configure ProxySQL parameters to recognize the user. First we connect to ProxySQL:

```
# mysql -uadmin -padmin -h 127.0.0.1 -P 6032
```

And then we set the parameters:

```
proxysql> UPDATE global_variables SET variable_value='p
       -> WHERE variable_name='mysql-monitor_username';
proxysql> UPDATE global_variables SET variable_value='$
       -> WHERE variable_name='mysql-monitor_password';
proxysql> LOAD MYSQL VARIABLES TO RUNTIME;
proxysql> SAVE MYSQL VARIABLES TO DISK;
```

Now that we've set the user in the database and ProxySQL, it is time to tell ProxySQL which MySQL servers are present in the topology:

```
proxysql> INSERT INTO mysql_servers(hostgroup_id, hostr
       -> VALUES (10,'10.124.33.5',3306);
proxysql> INSERT INTO mysql_servers(hostgroup_id, hostr
       -> VALUES (11,'10.124.33.169',3306);
proxysql> INSERT INTO mysql_servers(hostgroup_id, hostr
       -> VALUES (11,'10.124.33.130',3306);
proxysql> LOAD MYSQL SERVERS TO RUNTIME;
proxysql> SAVE MYSQL SERVERS TO DISK;
```

The next step is to define who our writer and reader groups. The servers present in the writer group will be able to receive DML operations, while `SELECT` queries will use the servers in the reader group. In this example, the host group 10 will be the writer, and host group 11 will be the reader:

```
proxysql> INSERT INTO mysql_replication_hostgroups
       -> (writer_hostgroup, reader_hostgroup) VALUES (
proxysql> LOAD MYSQL SERVERS TO RUNTIME;
proxysql> SAVE MYSQL SERVERS TO DISK;
```

Next, ProxySQL must have users that can access backend nodes to manage connections. Let's create the user on the backend source server:

```
mysql> CREATE USER 'app'@'%' IDENTIFIED by '$3Kr$t';
mysql> GRANT ALL PRIVILEGES ON *.* TO 'app'@'%';
```

And now we will configure ProxySQL with the user:

```
proxysql> INSERT INTO mysql_users (username,password,de
       -> VALUES ('app','$3Kr$t',10);
proxysql> LOAD MYSQL USERS TO RUNTIME;
proxysql> SAVE MYSQL USERS TO DISK;
```

The next step is the most exciting because it is here that we define the rules. The rules will tell ProxySQL where to send write and read queries, balancing the load on the servers:

```
proxysql> INSERT INTO mysql_query_rules
       -> (rule_id,username,destination_hostgroup,activ
       -> VALUES(1,'app',10,1,'^SELECT.*FOR UPDATE',1);
proxysql> INSERT INTO mysql_query_rules
       -> (rule_id,username,destination_hostgroup,activ
       -> VALUES(2,'app',11,1,'^SELECT ',1);
proxysql> LOAD MYSQL QUERY RULES TO RUNTIME;
proxysql> SAVE MYSQL QUERY RULES TO DISK;
```

ProxySQL has a thread responsible for connecting on each server listed in the `mysql_servers` table and checking the value of the `read_only` variable. Suppose the replica is showing up in the writer group, like this:

```
proxysql> SELECT * FROM mysql_servers;
```

```
+--------------+--------------+------+-----------+...
| hostgroup_id | hostname     | port | gtid_port |...
+--------------+--------------+------+-----------+...
| 10           | 10.124.33.5  | 3306 | 0         |...
| 11           | 10.124.33.169 | 3306 | 0         |...
| 11           | 10.124.33.130 | 3306 | 0         |...
+--------------+--------------+------+-----------+...
...+--------------+--------------+------+-----------+
...| status | weight | compression | max_connections |.
...+--------+--------+-------------+-----------------+.
...| ONLINE | 1      | 0           | 1000            |.
...| ONLINE | 1      | 0           | 1000            |.
...| ONLINE | 1      | 0           | 1000            |.
...+--------+--------+-------------+-----------------+.
...+---------------------+---------+----------------+--
...| max_replication_lag | use_ssl | max_latency_ms | c
...+---------------------+---------+----------------+--
...| 0                   | 0       | 0              |
...| 0                   | 0       | 0              |
...| 0                   | 0       | 0              |
...+---------------------+---------+----------------+--
3 rows in set (0.00 sec)
```

Because we do not want ProxySQL writing data to the replica servers, which would cause data inconsistency, we need to set the `read_only` option in the replica servers, so these servers will serve only read queries.

```
mysql> SET GLOBAL read_only=1;
```

Now we're ready to use our application. Running the following command should return the hostname that ProxySQL connected from:

```
$ mysql -uapp -p'$3Kr$t' -h 127.0.0.1 -P 6033 -e "selec
```

```
+----------------------+
| @@hostname           |
+----------------------+
| vinicius-grippa-node1 |
+----------------------+
```

ProxySQL has a lot more features and flexibility than we've shown here; our goal in this section was just to present the tool so you're aware of this option when deciding on an architecture.

NOTE

As we mentioned when configuring replication in Chapter 13, we want to reinforce the idea that ProxySQL needs to reach the MySQL servers; otherwise, it won't work.

HAProxy Load Balancer

HAProxy stands for High Availability Proxy, and it is a TCP/HTTP load balancer. It distributes a workload across a set of servers to maximize performance and optimize resource usage.

With the intent to expand your knowledge regarding MySQL architectures and different topologies, we will configure Percona XtraDB Cluster (Galera Cluster) with HAProxy in this section instead of a classic replication topology.

The architecture options are similar to ProxySQL's. HAProxy can be placed together with the application or in a middle layer. Figure 15-3 shows an example where HAProxy is placed on the same server as the application.

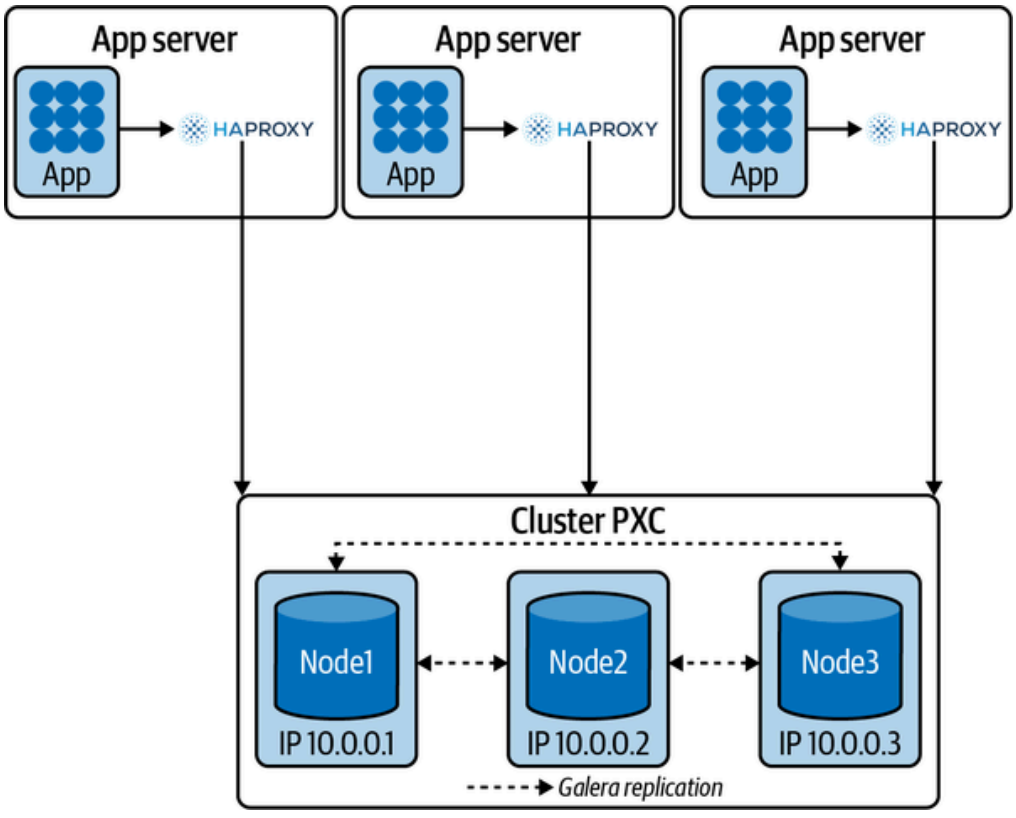

Figure 15-3. HAProxy together with the application

And Figure 15-4 shows a topology with HAProxy in a middle layer.

Again, these are archictectures with different pros and cons. While in the first one we do not have an extra hop (which reduces latency), we add extra load to the application server. Also, you have to configure HAProxy on each application server.

On the other hand, having HAProxy in the middle layer facilitates managing it and increases availability, because the application can connect to any HAProxy server. However, the extra hop adds latency.

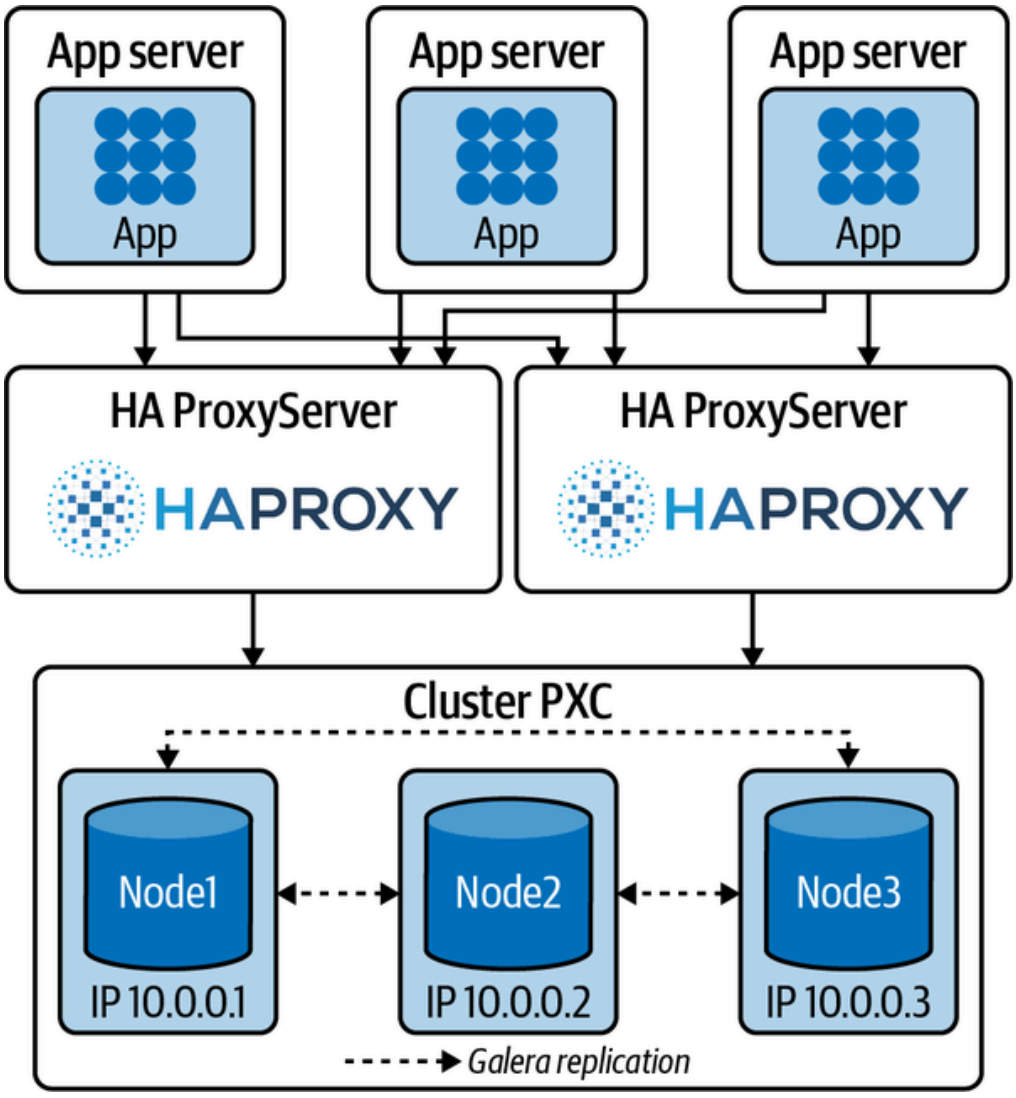

Figure 15-4. HAProxy in a middle layer running in dedicated servers

Installing and Configuring HAProxy

Common operating systems such as Red Hat/CentOS and Debian/Ubuntu provide the HAProxy package, and you can install it using the package manager. The installation process is relatively easy.

For Debian or Ubuntu, use these commands:

```
# apt update
# apt install haproxy
```

For Red Hat or CentOS, use:

```
# sudo yum update
# sudo yum install haproxy
```

When installed, HAProxy will set the default path for the configuration file as */etc/haproxy/haproxy.cfg*.

Before starting HAProxy, we need to configure it. For this demonstration, in our first scenario HAProxy will be located on the same server as the application. Here are the IPs of our three-node Galera Cluster:

```
172.16.3.45/Port:3306
172.16.1.72/Port:3306
172.16.0.161/Port:3306
```

Let's open our */etc/haproxy/haproxy.cfg* file and look at it. There are many parameters to customize, split into three sections:

`global`

A section in the configuration file for process-wide parameters

`defaults`

A section in the configuration file for default parameters

`listen`

A section in the configuration file that defines a complete proxy, including its frontend and backend parts

Table 15-1 shows the basic HAProxy parameters.

Table 15-1. HAProxy options (with links to HAProxy documentation)

Parameter	Description
balance	Defines the load balancing algorithm to be used in a backend.
clitimeout	Sets the maximum inactivity time on the client side
contimeout	Sets the maximum time to wait for a connection attempt to a server to succeed
daemon	Makes the process fork into background (recommended mode of operation)
gid	Changes the process's group ID to *<number>*
log	Adds a global syslog server
maxconn	Sets the maximum per-process number of concurrent connections to *<number>*
mode	Set the running mode or protocol of the instance
option dontlognull	Disable logging of null connections
optiontcplog	Enables advanced logging of TCP connections with session state and timers

To make HAProxy work, we will use the following configuration file based on our settings:

```
global
    log /dev/log   local0
    log /dev/log   local1 notice
    maxconn 4096
    #debug
    #quiet
```

```
    chroot      /var/lib/haproxy
    pidfile     /var/run/haproxy.pid
    user        haproxy
    group       haproxy
    daemon

    # turn on stats unix socket
    stats socket /var/lib/haproxy/stats

#---------------------------------------------------------
# common defaults that all the 'listen' and 'backend' s
# use if not designated in their block
#---------------------------------------------------------
defaults
     log     global
     mode    http
     option  tcplog
     option  dontlognull
     retries 3
     redispatch
     maxconn 2000
     contimeout      5000
     clitimeout      50000
     srvtimeout      50000

#---------------------------------------------------------
# round robin balancing between the various backends
#---------------------------------------------------------
listen mysql-pxc-cluster 0.0.0.0:3307
     mode tcp
     bind *:3307
     timeout client  10800s
     timeout server  10800s
     balance roundrobin
     option httpchk

     server vinicius-grippa-node2 172.16.0.161:3306 che
     inter 12000 rise 3 fall 3
     server vinicius-grippa-node1 172.16.1.72:3306 chec

     rise 3 fall 3
     server vinicius-grippa-default 172.16.3.45:3306 ch
     inter 12000 rise 3 fall 3
```

To start HAProxy, we use the `haproxy` command. We can pass any number of configuration parameters on the command line. To use a configuration file, use the `-f` option. For example, we can pass one configuration file:

```
# sudo haproxy -f /etc/haproxy/haproxy.cfg
```

or multiple configuration files:

```
# sudo haproxy -f /etc/haproxy/haproxy.cfg /etc/haproxy
```

or a directory:

```
# sudo haproxy -f conf-dir
```

With this configuration, HAProxy will balance the load between three nodes. In this case, it checks only if the `mysqld` process is listening on port 3306, but doesn't take into account the state of the node. So, it could be sending queries to a node that has `mysqld` running even if it's in the `JOINING` or `DISCONNECTED` state.

To check the current status of a node, we need something a little more complex. This idea was taken from Codership's Google group.

To implement this setup, we will need two scripts:

- `clustercheck` , located in */usr/local/bin* and a config for `xinetd`
- `mysqlchk` , located in */etc/xinetd.d* on each node

Both scripts are available in binaries and source distributions of Percona XtraDB.

Change the */etc/services* file by adding the following line for each node:

```
mysqlchk        9200/tcp                # mysqlchk
```

If the */etc/services* file does not exist, it's likely that `xinetd` is not installed.

To install it for CentOS/Red Hat, use:

```
# yum install -y xinetd
```

For Debian/Ubuntu, use:

```
# sudo apt-get install -y xinetd
```

Next, we need to create a MySQL user so the script can check if the node is healthy. Ideally, for security reasons, this user should have the minimum privileges required:

```
mysql> CREATE USER 'clustercheckuser'@'localhost' IDENT
               'clustercheckpassword!';
    -> GRANT PROCESS ON *.* TO 'clustercheckuser'@'loca
```

To validate how our node is performing on the health check, we can run the following command and observe the output:

```
# /usr/bin/clustercheck
```

```
HTTP/1.1 200 OK
Content-Type: text/plain
Connection: close
Content-Length: 40

Percona XtraDB Cluster Node is synced.
```

If we do this for all nodes, we will be ready to test whether our HAProxy setup is working. The easiest way to do this is to connect to it and execute some MySQL commands. Let's run a command that retrieves the hostname from which we are connected:

```
# mysql -uroot -psecret -h 127.0.0.1 -P 3307 -e "select
```

```
+-----------------------+
| @@hostname            |
+-----------------------+
```

```
| vinicius-grippa-node1 |
+-----------------------+
```

Running this a second time gives us:

```
$ mysql -uroot -psecret -h 127.0.0.1 -P 3307 -e "select
```

```
mysql: [Warning] Using a password on the command line i
insecure.
+-----------------------+
| @@hostname            |
+-----------------------+
| vinicius-grippa-node2 |
+-----------------------+
```

And the third time we get:

```
$ mysql -uroot -psecret -h 127.0.0.1 -P 3307 -e "select
```

```
mysql: [Warning] Using a password on the command line i
insecure.
+-------------------------+
| @@hostname              |
+-------------------------+
| vinicius-grippa-default |
+-------------------------+
```

As you can see, our HAProxy is connecting in a round-robin fashion. If we shut down one of the nodes, HAProxy will route only to the remaining ones.

MySQL Router

MySQL Router is responsible for distributing the traffic between members of an InnoDB cluster. It is a proxy-like solution to hide the cluster topology from applications, so applications don't need to know which member of a cluster is the primary node and which are secondaries. Note that MySQL Router will *not* work with Galera Clusters; it was developed for *InnoDB Cluster only*.

MySQL Router is capable of performing read/write splitting by exposing different interfaces. A common setup is to have one read/write interface and one read-only interface. This is the default behavior that also exposes two similar interfaces to use the X Protocol (used for CRUD operations and async calls).

The read/write split is done using the concept of *roles*: primary for writes and secondary for read-only. This is analogous to how members of cluster are named. Additionally, each interface is exposed via a TCP port so applications only need to know the IP:port combination used for writes and the one used for reads. Then MySQL Router will take care of connections to cluster members depending on the type of traffic to the server.

When working in a production environment, the MySQL server instances that make up an InnoDB Cluster run on multiple host machines as part of a network rather than on single machine. So, as with ProxySQL and HAProxy, the MySQL router can be a middle layer in the architecture.

Figure 15-5 illustrates how the production scenario works.

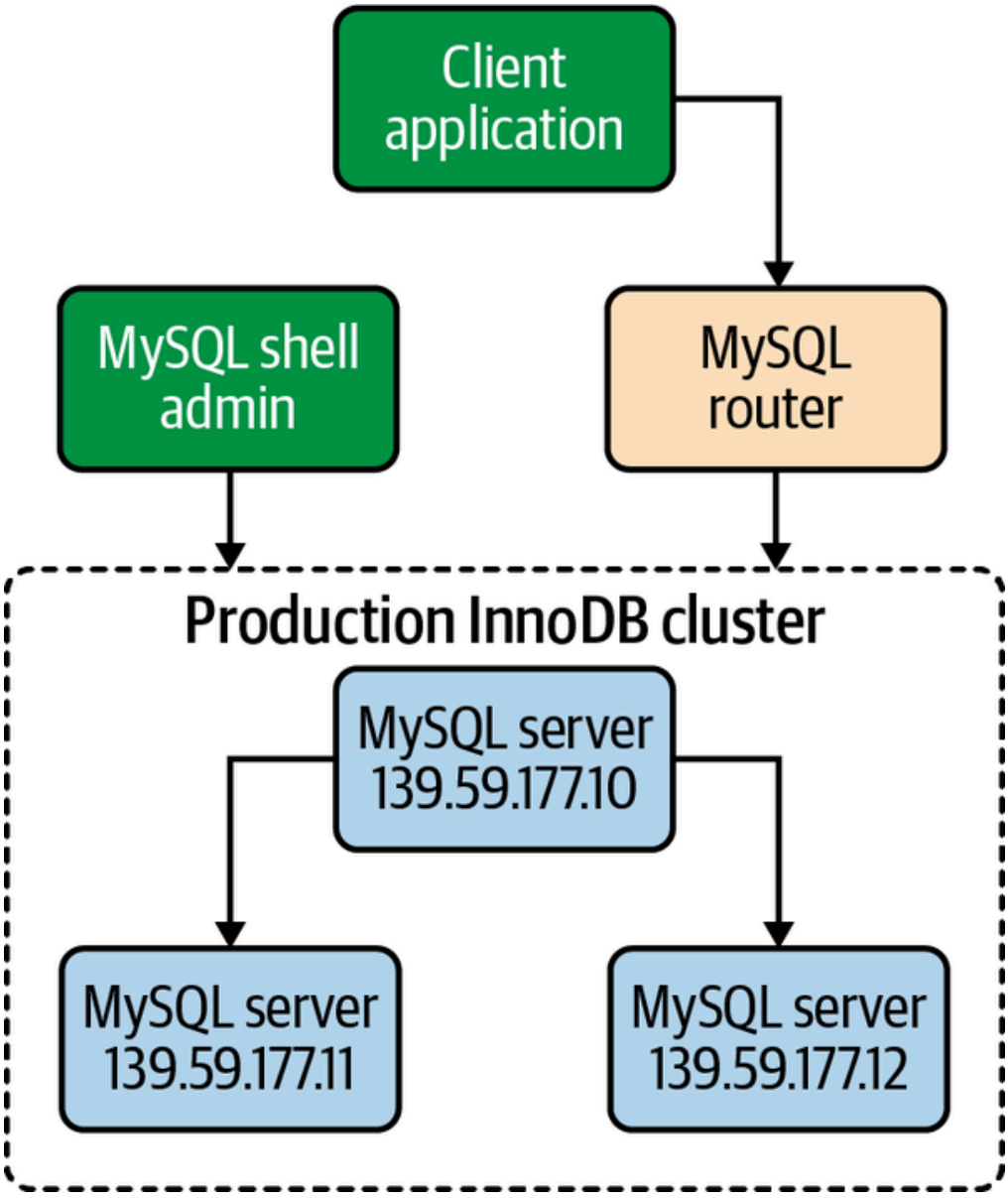

Figure 15-5. MySQL InnoDB Cluster production deployment

Now, to start our example, let's take a look at the MySQL members that are part of the InnoDB Cluster:

```
mysql> SELECT member_host, member_port, member_state, m
    -> FROM performance_schema.replication_group_member
```

```
+--------------+-------------+--------------+----------
| member_host  | member_port | member_state | member_rc
+--------------+-------------+--------------+----------
| 172.16.3.9   |        3306 | ONLINE       | SECONDARY
| 172.16.3.127 |        3306 | ONLINE       | SECONDARY
| 172.16.3.120 |        3306 | ONLINE       | PRIMARY
+--------------+-------------+--------------+----------
3 rows in set (0.00 sec)
```

```
mysql> SELECT cluster_name FROM mysql_innodb_cluster_me
```

```
+--------------+
| cluster_name |
+--------------+
| cluster1     |
+--------------+
1 row in set (0.00 sec)
```

Now that we have the configuration of the MySQL nodes and the cluster name, we can start configuring MySQL Router. For performance purposes it's recommended to set up MySQL Router in the same place as the application, supposing we have an instance per application server, so we will place our router on the application server. First, we are going to identify the version of MySQL Router compatible with our OS:

```
# cat /etc/*release
```

```
CentOS Linux release 7.9.2009 (Core)
```

Now, we will check the download page and install it using `yum` :

```
# yum install -y https://dev.mysql.com/get/Downloads/My
```

```
router-community-8.0.23-1.el7.x86_64.rpm
Loaded plugins: fastestmirror
mysql-router-community-8.0.23-1.el7.x86_64.rpm
Examining /var/tmp/yum-root-_ljdTQ/mysql-router-communi
86_64.rpm: mysql-router-community-8.0.23-1.el7.x86_64
Marking /var/tmp/yum-root-_ljdTQ/mysql-router-community
_64.rpm to be installed
Resolving Dependencies
--> Running transaction check
...
Running transaction
  Installing : mysql-router-community-8.0.23-1.el7.x86_
  1/1
  Verifying  : mysql-router-community-8.0.23-1.el7.x86_
```

```
  1/1

Installed:
  mysql-router-community.x86_64 0:8.0.23-1.el7

Complete!
```

Now that MySQL Router is installed, we need to create a dedicated directory for its operation:

```
# mkdir /var/lib/mysqlrouter
```

Next, we are going to bootstrap MySQL Router. The bootstrap will configure the router for operation with a MySQL InnoDB Cluster:

```
# mysqlrouter --bootstrap root@172.16.3.120:3306 \
    --directory /var/lib/mysqlrouter --conf-use-sockets
    --account app_router --account-create always \
    --user=mysql
```

```
Please enter MySQL password for root:
# Bootstrapping MySQL Router instance at '/var/lib/mysc

Please enter MySQL password for app_router:
- Creating account(s)
- Verifying account (using it to run SQL queries that w
Router)
- Storing account in keyring
- Adjusting permissions of generated files
- Creating configuration /var/lib/mysqlrouter/mysqlrout

...

## MySQL Classic protocol

- Read/Write Connections: localhost:6446, /var/lib/mysc
- Read/Only Connections:  localhost:6447,
/var/lib/mysqlrouter/mysqlro.sock

## MySQL X protocol

- Read/Write Connections: localhost:64460,
```

```
/var/lib/mysqlrouter/mysqlx.sock
- Read/Only Connections:  localhost:64470,
/var/lib/mysqlrouter/mysqlxro.sock
```

In the command line, we are telling the router to connect with the user `root`, in our primary server (`172.16.3.120`), at port 3306. We are also telling the router to create a socket file so we can connect using it. Finally, we are creating a new user (`app_router`) to use in our application.

Let's have a look at the contents that the bootstrap process created in our configuration directory (*/var/lib/mysqlrouter*):

```
# ls -l | awk '{print $9}'


data
log
mysqlrouter.conf
mysqlrouter.key
run
start.sh
stop.sh
```

A generated MySQL Router configuration file (*mysqlrouter.conf*) looks similar to this:

```
# cat mysqlrouter.conf


# File automatically generated during MySQL Router boot
[DEFAULT]
user=mysql
logging_folder=/var/lib/mysqlrouter/log
runtime_folder=/var/lib/mysqlrouter/run

...

[rest_routing]
require_realm=default_auth_realm

[rest_metadata_cache]
require_realm=default_auth_realm
```

In this example, MySQL Router configured four ports (two ports to read/write using the regular MySQL protocol, and two to read/write using the X Protocol) and four sockets. Ports are added by default, and sockets were added because we passed in `--conf-use-sockets`. The InnoDB Cluster named `cluster1` is the source of the metadata, and the destinations are using the InnoDB Cluster metadata cache to dynamically configure host information.

By executing the *start.sh* script we can start the MySQL router daemon:

```
# ./start.sh


# PID 1684 written to '/var/lib/mysqlrouter/mysqlrouter
logging facility initialized, switching logging to logg
configuration
```

Now, we can observe the process running:

```
# ps -ef | grep -i mysqlrouter


root      1683     1  0 17:36 pts/0    00:00:00 sudo
ROUTER_PID=/var/lib/mysqlrouter/mysqlrouter.pid /usr/bi
/var/lib/mysqlrouter/mysqlrouter.conf --user=mysql
mysql     1684  1683  0 17:36 pts/0    00:00:17 /usr/bi
/var/lib/mysqlrouter/mysqlrouter.conf --user=mysql
root      1733  1538  0 17:41 pts/0    00:00:00 grep --
mysqlrouter
```

And the ports open:

```
# netstat -tulnp | grep -i mysqlrouter


tcp   0   0 0.0.0.0:64470   0.0.0.0:*   LISTEN   1684/m
tcp   0   0 0.0.0.0:8443    0.0.0.0:*   LISTEN   1684/m
tcp   0   0 0.0.0.0:64460   0.0.0.0:*   LISTEN   1684/m
tcp   0   0 0.0.0.0:6446    0.0.0.0:*   LISTEN   1684/m
tcp   0   0 0.0.0.0:6447    0.0.0.0:*   LISTEN   1684/m
```

We've configured MySQL Router with the InnoDB Cluster, so now we can test this with read and read/write connections. First, we will connect to the writer port (6446):

```
# mysql -uroot -psecret -h 127.0.0.1 -P 6446 \
    -e "create database learning_mysql;"
# mysql -uroot -psecret -h 127.0.0.1 -P 6446 \
    -e "use learning_mysql; select database()"
```

```
+----------------+
| database()     |
+----------------+
| learning_mysql |
+----------------+
```

As you can see, it is possible to execute both reads and writes in the writer port.

Now we will check the read port (6447) using a `SELECT` statement:

```
# mysql -uroot -psecret -h 127.0.0.1 -P 6447 \
    -e "use learning_mysql; select database()"
```

```
+----------------+
| database()     |
+----------------+
| learning_mysql |
+----------------+
```

That's working, but let's try to execute a write:

```
# mysql -uroot -psecret -h 127.0.0.1 -P 6447 \
    -e "create database learning_mysql_write;"
```

```
ERROR 1290 (HY000) at line 1: The MySQL server is runni
--super-read-only option so it cannot execute this stat
```

So, the read port only accepts reads. It is also possible to see the router load-balancing the reads:

```
# mysql -uroot -psecret -h 127.0.0.1 -P 6447 -e "select
```

```
+-----------------------+
| @@hostname            |
+-----------------------+
| vinicius-grippa-node1 |
+-----------------------+
```

```
# mysql -uroot -psecret -h 127.0.0.1 -P 6447 -e "select
```

```
insecure.
+-----------------------+
| @@hostname            |
+-----------------------+
| vinicius-grippa-node2 |
+-----------------------+
```

In this way, if any downtime occurs in one of the MySQL nodes, MySQL Router will route the queries to the remaining active nodes.