Chapter 13. High Availability

In the IT context, the term *high availability* defines a state of continuous operation for a specified length of time. The goal is not eliminating the risk of failure—that would be impossible. Rather, we are trying to guarantee that in a failure situation, the system remains available so that operation can continue. We often measure availability against a 100% operational or never-fails standard. A common standard of availability is known as *five 9s*, or 99.999% availability. Two 9s would be a system that guarantees 99% availability, allowing up to 1% downtime. Over the course of a year, this would translate to 3.65 days of unavailability.

Reliability engineering uses three principles of systems design to help achieve high availability: elimination of single points of failure (SPOFs), reliable crossover or failover points, and failure detection capabilities (including monitoring, discussed in Chapter 12).

Redundancy is required for many components to achieve high availability. A simple example is an airplane with two engines. If one engine fails while flying, the aircraft can still land at an airport. A more complex example is a nuclear power plant, where there are numerous redundant protocols and components to avoid catastrophic failures. Similarly, to achieve high availability of a database we need network redundancy, disk redundancy, different power supplies, multiple application and database servers, and much more.

This chapter will focus on the options to achieve high availability that MySQL databases offer.

Asynchronous Replication

Replication enables data from one MySQL database server (known as a *source*) to be copied to one or more other MySQL database servers (known as *replicas*). MySQL replication by default is asynchronous. With asynchronous replication, the source writes events to its binary log, and replicas request

them when ready. There is no guarantee that any event will ever reach any replica. It's a loosely coupled source/replica relationship, where the following are true:

- The source does not wait for the replica to catch up.
- The replica determines how much to read and from which point in the binary log.
- The replica can be arbitrarily far behind the source in reading or applying changes. This issue is known as *replication lag*, and we will look at ways of minimizing it.

Asynchronous replication provides lower write latency since a write is acknowledged locally by a source before being written to the replicas.

MySQL implements its replication capabilities using three main threads, one on the source server and two on the replicas:

Binary log dump thread

The source creates a thread to send the binary log contents to a replica when the replica connects. We can identify this thread in the output of `SHOW PROCESSLIST` on the source as the `Binlog Dump` thread.

The binary log dump thread acquires a lock on the source's binary log for reading each event sent to the replica. When the source reads the event, the lock is released, even before the source sends the event to the replica.

Replication I/O thread

When we execute the `START SLAVE` statement on a replica server, the replica creates an I/O thread connected to the source and asks it to send the updates recorded in its binary logs.

The replication I/O thread reads the updates that the source's `Binlog Dump` thread sends (see the previous item) and copies them to local files that comprise the replica's relay log.

MySQL shows the state of this thread as `Slave_IO_running` in the output of `SHOW SLAVE STATUS`.

Replication SQL thread

The replica creates a SQL thread to read the relay log written by the replication I/O thread and execute the transactions contained in it.

NOTE

As mentioned in Chapter 1, Oracle, Percona, and Maria DB are working to remove legacy terminology with negative connotations from their products. The documentation already uses the terms *source* and *replica*, as we do in this book, but because of the need to maintain backward compatibility and support for older versions, it would be impossible to completely change the terminology in one release. This is an ongoing effort.

There are ways to improve replication parallelization, as you'll see later in this chapter.

Figure 13-1 shows what the MySQL replication architecture looks like.

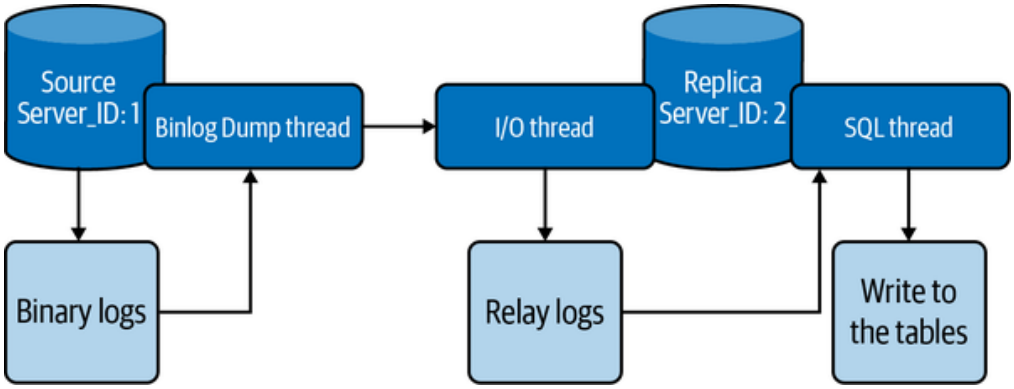

Figure 13-1. Asynchronous replication architecture flow

Replication works because events written to the binary log are read from the source and then processed on the replica, as shown in Figure 13-1. The events are recorded within the binary log in different formats according to the type of event. MySQL replication has three kinds of binary logging formats:

Row-based replication (RBR)

The source writes events to the binary log that indicate how individual table rows are changed. Replication of the source to the replica works by copying the events representing the replica's table rows' changes. For MySQL 5.7 and 8.0, this is the default replication format.

Statement-based replication (SBR)

The source writes SQL statements to the binary log. Replication of the source to the replica works by executing the SQL statements on the replica.

Mixed replication

You can also configure MySQL to use a mix of both statement-based and row-based logging, depending on which one is most appropriate to log the changes. With mixed-format logging, MySQL uses a statement-based log by default but switches to a row-based log for certain unsafe statements that have a nondeterministic behavior. For example, suppose we have the following statement:

```
mysql> UPDATE customer SET last_update=NOW() WHER
```

We know that the function `NOW()` returns the current date and time. Imagine that the source replicates the statement with 1 second of delay (there could be various reasons for this, such as the replica being on a different continent than the source). When the replica receives the statement and executes it, there will be a 1-second difference in the date and time returned by the function, leading to data inconsistency between the source and replica. When the mixed replication format is used, whenever MySQL parses a nondeterministic function like this, it will convert the statement to row-based replication. You can find a list of other functions that MySQL considers unsafe in the documentation.

Basic Parameters to Set on the Source and the Replica

There are some basic settings that we need to set on both the source server and the replica server in order to make replication work. They are required for all methods explained in this section.

On the source server, you must enable binary logging and define a unique server ID. You'll need to restart the server after making these changes (if you haven't already) because these parameters are not dynamic.

TIP

The server ID does not need to be incremental or be in any order, like having the source server ID be smaller than the replica server ID. The only requirement is that it be unique in each server that is part of the replication topology.

Here's what this will look like in the *my.cnf* file:

```
[mysqld]
log-bin=mysql-bin
server-id=1
```

You also need to establish a unique server ID for each replica. Like with the source, if you haven't done this yet, you'll need to restart the replica server after assigning it its ID. It is not mandatory to enable the binary log in the replica server, although it is a recommended practice:

```
[mysqld]
log-bin=mysql-replica-bin
server-id=1617565330
binlog_format = ROW
log_slave_updates
```

Using the `log_slave_updates` option tells the replica server that commands from a source server should be logged to the replica's own binary log. Again, this is not mandatory, but it is recommended as a good practice.

Each replica connects to the source using a MySQL username and password, so you'll also need to create a user account on the source server that the replica can use to connect (for a refresher on this, see "Creating and Using New Users" on page 317). Any account can be used for this operation, provided it has been granted the `REPLICATION SLAVE` privilege. Here's an example of how to create the user on the source server:

```
mysql> CREATE USER 'repl'@'%' IDENTIFIED BY 'P@ssw0rd!'
mysql> GRANT REPLICATION SLAVE ON *.* TO 'repl'@'%';
```

TIP

If you are using an automation tool like Ansible to deploy MySQL, you can use the following bash command to create server IDs:

```
# date '+%s'

1617565330
```

The command converts the current date and time to an integer value, so it increases monotonically. Note that the `date` command does not guarantee the values' uniqueness, but you may find it convenient to use as it provides a relatively good uniqueness level.

In the next sections, you will see different options to create a replica server.

Creating a Replica Using PerconaXtraBackup

As we saw in Chapter 10, the Percona XtraBackup tool provides a method of performing a hot backup of your MySQL data while the system is running. It also offers advanced capabilities like parallelization, compression, and encryption.

The first step is taking a copy of the current source so we can start our replica. The XtraBackup tool performs a physical backup of the source (see "Physical and Logical Backups" on page 376). We will use the commands provided in "Percona XtraBackup":

```
# xtrabackup --defaults-file=my.cnf -uroot -p_<passworc
    -H <host> -P 3306 --backup --parallel=4 \
    --datadir=./data/ --target-dir=./backup/
```

Alternatively, you can use `rsync`, NFS, or any other method that you feel comfortable with.

Once XtraBackup finishes the backup, we will send the files to a backup directory on the replica server. In this example, we will send the files using the `scp` command:

```
# scp -r ./backup/* <user>@<host>:/backup
```

At this point we're finished with the source. The following steps will run only on the replica server. The next step is to prepare our backup:

```
# xtrabackup --prepare --apply-log --target-dir=./
```

With everything set, we are going to move the backup to the data directory:

```
# xtrabackup --defaults-file=/etc/my.cnf --copy-back --
```

NOTE

Before proceeding, verify that your replica server does not have the same `server_id` as your source. If you followed the steps outlined in the previous section, you should have taken care of this already; if not, do so now.

On the replica, the content of the file *xtrabackup_binlog_info* will look something like this:

```
$ cat /backup/xtrabackup_binlog_info
mysql-bin.000003    156
```

This information is essential because it tells us where to start replicating. Remember that the source was still receiving operations when we took the backup, so we need to know what position MySQL was at in the binary log file when the backup finished.

With that information, we can run the command to start the replication. It will look something like this:

```
mysql> CHANGE MASTER TO MASTER_HOST='192.168.1.2', MAST
    -> MASTER_PASSWORD='P@ssw0rd!',
    -> MASTER_LOG_FILE='mysql-bin.000003', MASTER_LOG_F
mysql> START SLAVE;
```

Once you've started, you can run the `SHOW SLAVE STATUS` command to check if the replication is working:

```
mysql> SHOW SLAVE STATUS\G


               Slave_IO_Running: Yes
              Slave_SQL_Running: Yes
                    Last_Errno: 0
                    Last_Error:
                  Skip_Counter: 0
           Exec_Master_Log_Pos: 8332
               Relay_Log_Space: 8752
               Until_Condition: None
        Seconds_Behind_Master: 0
Master_SSL_Verify_Server_Cert: No
                Last_IO_Errno: 0
                Last_IO_Error:
               Last_SQL_Errno: 0
               Last_SQL_Error:
```

It is important to check that both threads are running (`Slave_IO_Running` and `Slave_SQL_Running`), whether there have been any errors (`Last_Error`), and how many seconds the replica is behind the source. For large databases with an intensive write workload, the replica may take a while to catch up.

Creating a Replica Using the Clone Plugin

MySQL 8.0.17 introduced the clone plugin, which can be used to make one MySQL server instance a *clone* of another. We refer to the server instance where the `CLONE` statement is executed as the *recipient* and to the source server instance from which the recipient will clone the data as the *donor*. The donor instance can be local or remote. The cloning process works by creating a physical snapshot of the data and metadata stored in the InnoDB storage engine on the donor, and transferring it to the recipient. Both local and remote instances perform the same clone operation; there is no difference related to the data between the two options.

Let's walk through a real example. We'll show you some additional details along the way, like how to monitor the progress of a long-running `CLONE`

command, the privileges required to clone, and more. The following example uses the classic shell. We'll talk about MySQL Shell, introduced in MySQL 8.0, in Chapter 16.

Choose the MySQL server to clone from and connect to it as the `root` user. Then install the clone plugin, create a user to transfer the data from the donor server, and grant that user the `BACKUP_ADMIN` privilege:

```
mysql> INSTALL PLUGIN CLONE SONAME "mysql_clone.so";
mysql> CREATE USER clone_user@'%' IDENTIFIED BY "clone_
mysql> GRANT BACKUP_ADMIN ON *.* to clone_user;
```

Next, to observe the progress of the cloning operation, we need to grant that user privileges to view the `performance_schema` database and execute functions:

```
mysql> GRANT SELECT ON performance_schema.* TO clone_us
mysql> GRANT EXECUTE ON *.* to clone_user;
```

Now we will move to the recipient server. If you are provisioning a new node, first initialize a data directory and start the server.

Connect to the recipient server as the `root` user. Then install the clone plugin, create a user to replace the current instance data with the cloned data, and grant that user the `CLONE_ADMIN` privilege. We'll also provide a list of valid donors that the recipient can clone (here, just one):

```
mysql> INSTALL PLUGIN CLONE SONAME "mysql_clone.so";
mysql> SET GLOBAL clone_valid_donor_list = "127.0.0.1:2
mysql> CREATE USER clone_user IDENTIFIED BY "clone_pass
mysql> GRANT CLONE_ADMIN ON *.* to clone_user;
```

We'll grant this user the same privileges we did on the donor side, to observe the progress on the recipient side:

```
mysql> GRANT SELECT ON performance_schema.* TO clone_us
mysql> GRANT EXECUTE ON *.* to clone_user;
```

We now have everything we need in place, so it's time to start the cloning process. Note that the donor server must be reachable from the recipient. The recipient will connect to the donor with the address and credentials provided and start cloning:

```
mysql> CLONE INSTANCE FROM clone_user@192.168.1.2:3306
    -> IDENTIFIED BY "clone_password";
```

The recipient must shut down and restart itself for the clone operation to succeed. We can monitor the progress with the following query:

```
SELECT STAGE, STATE, CAST(BEGIN_TIME AS TIME) as "START
CASE WHEN END_TIME IS NULL THEN
LPAD(sys.format_time(POWER(10,12) * (UNIX_TIMESTAMP(now
    UNIX_TIMESTAMP(BEGIN_TIME))), 10,' )
ELSE
LPAD(sys.format_time(POWER(10,12) * (UNIX_TIMESTAMP(END
    UNIX_TIMESTAMP(BEGIN_TIME))), 10, )
END AS DURATION,
LPAD(CONCAT(FORMAT(ROUND(ESTIMATE/1024/1024,0), 0)," ME
AS "Estimate",
CASE WHEN BEGIN_TIME IS NULL THEN LPAD('0%, 7, ' )
WHEN ESTIMATE > 0 THEN
LPAD(CONCAT(CAST(ROUND(DATA*100/ESTIMATE, 0) AS BINARY)
WHEN END_TIME IS NULL THEN LPAD('0%, 7, ' )
ELSE LPAD('100%, 7, ' ') END AS "Done(%)"
from performance_schema.clone_progress;
```

This will allow us to observe each state of the cloning process. The output will be similar to this:

```
+-----------+-----------+------------+-----------+-----
| STAGE     | STATE     | START TIME | DURATION  | Esti
+-----------+-----------+------------+-----------+-----
| DROP DATA | Completed | 14:44:46   |    1.33 s |
+-----------+-----------+------------+-----------+-----
| FILE COPY | Completed | 14:44:48   |    5.62 s | 1,51
+-----------+-----------+------------+-----------+-----
| PAGE COPY | Completed | 14:44:53   |  95.06 ms |
+-----------+-----------+------------+-----------+-----
| REDO COPY | Completed | 14:44:54   |  99.71 ms |
```

```
+-----------+-----------+-----------+-----------+-----
| FILE SYNC | Completed | 14:44:54  |    6.33 s |
+-----------+-----------+-----------+-----------+-----
| RESTART   | Completed | 14:45:00  |    4.08 s |
+-----------+-----------+-----------+-----------+-----
| RECOVERY  | Completed | 14:45:04  | 516.86 ms |
+-----------+-----------+-----------+-----------+-----
7 rows in set (0.08 sec)
```

As mentioned previously, there is a restart at the end. Note that replication has not started yet.

In addition to cloning the data, the cloning operation extracts the binary log position and GTID from the donor server and transfers them to the recipient. We can execute the following queries on the donor to view the binary log position or the GTID of the last transaction that was applied:

```
mysql> SELECT BINLOG_FILE, BINLOG_POSITION FROM perform
```

```
+------------------+-----------------+
| BINLOG_FILE      | BINLOG_POSITION |
+------------------+-----------------+
| mysql-bin.000002 |       816804753 |
+------------------+-----------------+
1 row in set (0.01 sec)
```

```
mysql> SELECT @@GLOBAL.GTID_EXECUTED;
```

```
+------------------------+
| @@GLOBAL.GTID_EXECUTED |
+------------------------+
|                        |
+------------------------+
1 row in set (0.00 sec)
```

In this example we are not using GTIDs, so the query does not return anything. Next, we will run the command to start the replication:

```
mysql> CHANGE MASTER TO MASTER_HOST = '192.168.1.2', MA
    -> MASTER_USER = 'repl', MASTER_PASSWORD = 'P@ssw0r
    -> MASTER_LOG_FILE = 'mysql-bin.000002',
    -> MASTER_LOG_POSITION = 816804753;
mysql> START SLAVE;
```

As in the previous section, we can check that replication is working correctly by running the `SHOW SLAVE STATUS` command.

The advantage of this approach is that the clone plugin automates the whole process, and only at the end is it necessary to execute the `CHANGE MASTER` command. The disadvantage is that the plugin is available only for MySQL 8.0.17 and higher. While it's still relatively new, we believe that in years to come, this process may become the default.

Creating a Replica Using mysqldump

This is what we might call the classic approach. It's the typical option for those who are getting started with MySQL and still learning about the ecosystem. As usual, we assume here that you have performed the necessary setup in "Basic Parameters to Set on the Source and the Replica".

Let's see an example of using `mysqldump` to create a new replica. We will execute the backup from the source server:

```
# mysqldump -uroot -p<password> --single-transaction \
    --all-databases --routines --triggers --events \
    --master-data=2 > backup.sql
```

The dump succeeded if the message `Dump completed` appears at the end:

```
# tail -1f backup.sql

-- Dump completed on 2021-04-26 20:16:33
```

With the backup taken, we need to import it in the replica server. For example, you can use this command:

```
$ mysql < backup.sql
```

Once that's done, you'll need to execute the `CHANGE MASTER` command with the coordinates extracted from the dump (for more details about `mysqldump`, revisit "The mysqldump Program"). Because we used the `--master-data=2` option, the information will be written at the beginning of the dump. For example:

```
$ head -n 35 out
-- MySQL dump 10.13  Distrib 5.7.31-34, for Linux (x86_
--
-- Host: 127.0.0.1    Database:
-- ------------------------------------------------------
-- Server version   5.7.33-log

...

--
-- Position to start replication or point-in-time recov
--

-- CHANGE MASTER TO MASTER_LOG_FILE='mysql-bin.000001',
```

Or, if you're using GTIDs:

```
--
-- GTID state at the beginning of the backup
-- (origin: @@global.gtid_executed)
--

SET @@GLOBAL.GTID_PURGED=00048008-1111-1111-1111-111111
```

Next, we are going to execute the command to start the replication. For the GTID scenario, it looks like this:

```
mysql> CHANGE MASTER TO MASTER_HOST='192.168.1.2', MAST
    -> MASTER_PASSWORD = 'P@ssw0rd!', MASTER_AUTO_POSIT
mysql> START SLAVE;
```

For traditional replication, you can start replication from the previously extracted binary log file position as follows:

```
mysql> CHANGE MASTER TO MASTER_LOG_FILE='mysql-bin.0000
    -> MASTER_HOST='192.168.1.2', MASTER_USER='repl',
    -> MASTER_PASSWORD='P@ssw0rd!';
mysql> START SLAVE;
```

To verify that replication is working, execute the `SHOW SLAVE STATUS` command.

Creating a Replica Using mydumper and myloader

`mysqldump` is the most common tool used by beginners for performing backups and building replicas. But there is a more efficient method: `mydumper`. Like `mysqldump`, this tool generates a logical backup and can be used to create a consistent backup of your database. The main difference between `mydumper` and `mysqldump` is that `mydumper`, when paired with `myloader`, can dump and restore data in parallel, improving the dump and, especially, restore time. Imagine a scenario where your database has a dump of 500 GB. Using `mysqldump`, you will have a single huge file. With `mydumper`, you will have one file per table, allowing the restore process to be executed in parallel later.

Setting up the mydumper and myloader utilities

You can run `mydumper` directly on the source server or from another server, which in general is better since it will avoid the overhead in the storage system of writing the backup files on the same server.

To install `mydumper`, download the package specific to the operating system version you are using. You can find the releases in the `mydumper` GitHub repository. Let's see an example for CentOS:

```
# yum install https://github.com/maxbube/mydumper/relea
mydumper-0.10.3-1.el7.x86_64.rpm -y
```

Now you should have both the `mydumper` and `myloader` commands installed on the server. You can validate this with:

```
$ mydumper --version
mydumper 0.10.3, built against MySQL 5.7.33-36

$ myloader --version
myloader 0.10.3, built against MySQL 5.7.33-36
```

Extracting data from the source

The following command will execute a dump of all databases (except `mysql`, `test`, and the `sys` schema) with 15 simultaneous threads and will also include triggers, views, and functions:

```
# mydumper --regex '^(?!(mysql\.|test\.|sys\.))' --thre
--user=learning_user --password='learning_mysql' --host
    --port=3306 --trx-consistency-only --events --routi
    --compress --outputdir /backup --logfile /tmp/log.c
```

TIP

You will need to grant at least the `SELECT` and `RELOAD` permissions to the `mydumper` user.

If you check the output directory (`outputdir`), you will see the compressed files. Here's the output on one of the authors' machines:

```
# ls -l backup/
total 5008
-rw...1 vinicius.grippa percona   182 May  1 19:30 meta
-rw...1 vinicius.grippa percona   258 May  1 19:30 sysb
-rw...1 vinicius.grippa percona 96568 May  1 19:30 sysb
-rw...1 vinicius.grippa percona   258 May  1 19:30 sysb
-rw...1 vinicius.grippa percona 96588 May  1 19:30 sysb
-rw...1 vinicius.grippa percona   258 May  1 19:30 sysb
...
```

TIP

Decide the number of threads based on the CPU cores of the database server and server load. Doing a parallel dump can consume a lot of server resources.

Restoring data in a replica server

Like with `mysqldump`, we need to have the replica MySQL instance already up and running. Once the data is ready to be imported, we can execute the following command:

```
# myloader --user=learning_user --password='learning_my
--threads=25 --host=192.168.1.3 --port=3306
--directory=/backup --overwrite-tables --verbose 3
```

Establishing the replication

Now that we've restored the data, we will set up replication. We need to find the correct binary log position at the start of the backup. This information is stored in the `mydumper` metadata file:

```
$ cat backup/metadata
Started dump at: 2021-05-01 19:30:00
SHOW MASTER STATUS:
    Log: mysql-bin.000002
    Pos: 9530779
    GTID:00049010-1111-1111-1111-111111111111:1-319

Finished dump at: 2021-05-01 19:30:01
```

Now, we simply execute the `CHANGE MASTER` command like we did previously for `mysqldump`:

```
mysql> CHANGE MASTER TO MASTER_HOST='192.168.1.2', MAST
    -> MASTER_PASSWORD='P@ssw0rd!',  MASTER_LOG_FILE='m
    -> MASTER_LOG_POS=9530779, MASTER_PORT=49010;
mysql> START SLAVE;
```

Group Replication

It might be a bit controversial to include Group Replication in the asynchronous replication group. The short explanation for this choice is that Group Replication is asynchronous. The confusion here can be explained by the comparison with Galera (discussed in "Galera/PXC Cluster"), which claims to be synchronous or virtually synchronous.

The more detailed reasoning is that it depends on how we define replication. In the MySQL world, we define replication as the process that enables changes made in one database (the source) to be automatically duplicated in another (the replica). The entire process involves five different steps:

1. Locally applying the change on the source
2. Generating a binlog event
3. Sending the binlog event to the replica(s)
4. Adding the binlog event to the replica's relay log
5. Applying the binlog event from the relay log on the replica

In MySQL Group Replication and Galera (even if the Galera cache primarily replaces the binlog and relay log files), only step 3 is synchronous—the streaming of the binary log event (or write set in Galera) to the replica(s).

Thus, while the process of sending (replicating/streaming) the data to the other servers is synchronous, the *applying* of these changes is still wholly asynchronous.

TIP

Group Replication has been available since MySQL 5.7. However, when the product was released, it was not mature enough, leading to constant performance issues and crashes. We highly recommend using MySQL 8.0 if you want to test Group Replication.

Installing Group Replication

The first advantage of Group Replication compared to Galera is that you don't have to install different binaries. MySQL Server provides Group Replication

as a plugin. It's also available for Oracle MySQL and Percona Server for MySQL; for details on installing those, see Chapter 1.

To confirm that the Group Replication plugin is enabled, run the following query:

```
mysql> SELECT PLUGIN_NAME, PLUGIN_STATUS, PLUGIN_TYPE
    -> FROM INFORMATION_SCHEMA.PLUGINS
    -> WHERE PLUGIN_NAME LIKE 'group_replication';
```

The output should show `ACTIVE`, as you see here:

```
+-------------------+---------------+------------------
| PLUGIN_NAME       | PLUGIN_STATUS | PLUGIN_TYPE
+-------------------+---------------+------------------
| group_replication | ACTIVE        | GROUP REPLICATION
+-------------------+---------------+------------------
1 row in set (0.00 sec)
```

If the plugin is not installed, run the following command to install it:

```
mysql> INSTALL PLUGIN group_replication SONAME 'group_r
```

With the plugin active, we will set the minimum parameters required on the servers to start Group Replication. Open *my.cnf* on server 1 and add the following:

```
[mysqld]
server_id=175907211
log-bin=mysqld-bin
enforce_gtid_consistency=ON
gtid_mode=ON
log-slave-updates
transaction_write_set_extraction=XXHASH64
master_info_repository=TABLE
relay_log_info_repository=TABLE
binlog_checksum=NONE
```

Let's go over each of those parameters:

server_id

Like with classic replication, this parameter helps to identify each member in the group using a unique ID. You must use a different value for each server participating in Group Replication.

log_bin

In MySQL 8.0 this parameter is enabled by default. It is responsible for recording all the changes in the database in binary log files.

enforce_gtid_consistency

This value must be set to `ON` to instruct MySQL to execute transaction-safe statements to ensure consistency when replicating data.

gtid_mode

This directive enables global transaction identifier-based logging when set to `ON`. This is required for Group Replication.

log_slave_updates

This value is set to `ON` to allow members to log updates from each other. In other words, the directive chains the replication servers together.

transaction_write_set_extraction

This instructs the MySQL server to collect write sets and encode them using a hashing algorithm. In this case, we are using the XXHASH64 algorithm. Write sets are defined by primary keys on each record.

master_info_repository

When set to `TABLE`, this directive allows MySQL to store details about source binary log files and positions into a table rather than a file to enable faster replication and guarantee consistency using InnoDB's ACID properties. In MySQL 8.0.23 this is the default, and the `FILE` option is deprecated.

relay_log_info_repository

When set to `TABLE`, this configures MySQL to store replication information as an InnoDB table. In MySQL 8.0.23 this is the default, and the `FILE` option is deprecated.

`binlog_checksum`

Setting this to `NONE` tells MySQL not to write a checksum for each event in the binary log. The server will instead verify events when they are written by checking their length. In versions of MySQL up to and including 8.0.20, Group Replication cannot make use of checksums. If you're using a later release and want to use checksums, you can omit this setting and use the default, `CRC32`.

Next, we are going to add some specific parameters for Group Replication:

```
[mysqld]

loose-group_replication_group_name="8dc32851-d7f2-4b63-
loose-group_replication_start_on_boot=OFF
loose-group_replication_local_address="10.124.33.139:33
loose-group_replication_group_seeds="10.124.33.139:3306
10.124.33.90:33061, 10.124.33.224:33061"
loose-group_replication_bootstrap_group=OFF
bind-address = "0.0.0.0"
report_host = "10.124.33.139"
```

NOTE

We are using the `loose-` prefix to instruct the server to start even when the MySQL Group Replication plugin is not installed and configured. This avoids encountering server errors before you finish configuring all the settings.

Let's see what each parameter does:

`group_replication_group_name`

This is the name of the group that we are creating. We are going to use the built-in Linux `uuidgen` command to generate a universally unique identifier (UUID). It produces output like this:

```
$ uuidgen
```

```
8dc32851-d7f2-4b63-8989-5d4b467d8251
```

group_replication_start_on_boot

When set to `OFF`, the value instructs the plugin not to start working automatically when the server starts. You may set this value to `ON` once you are through with configuring all the group members.

loose-group_replication_local_address

This is the internal IP address and port combination used for communicating with other MySQL server members in the group. The recommended port for Group Replication is 33061.

group_replication_group_seeds

This configures the IP addresses or hostnames of members participating in Group Replication, together with their communication port. New members use the value to establish themselves in the group.

group_replication_bootstrap_group

This option instructs the server whether to create a group or not. We will only enable this option on demand on server 1, to avoid creating multiple groups. So, it will remain off for now.

bind_address

The value of `0.0.0.0` tells MySQL to listen to all networks.

report_host

This is the IP address or hostname the group members reports to each other when they are registered in the group.

Setting up MySQL Group Replication

First, we will set up the `group_replication_recovery` channel. MySQL Group Replication uses this channel to transfer transactions between members. Because of this, we must set up a replication user with `REPLICATION SLAVE` permission on each server.

So, on server 1, log in to the MySQL console and execute the following commands:

```
mysql> SET SQL_LOG_BIN=0;
mysql> CREATE USER replication_user@'%' IDENTIFIED BY '
mysql> GRANT REPLICATION SLAVE ON *.* TO 'replication_u
mysql> FLUSH PRIVILEGES;
mysql> SET SQL_LOG_BIN=1;
```

We first set `SQL_LOG_BIN` to `0` to prevent the new user's details from being logged to the binary log then we reenable it at the end.

To instruct the MySQL server to use the replication user we have created for the `group_replication_recovery` channel, run this command:

```
mysql> CHANGE MASTER TO MASTER_USER='replication_user',
    -> MASTER_PASSWORD='P@ssw0rd!' FOR CHANNEL
    -> 'group_replication_recovery';
```

These settings will allow members joining the group to run the distributed recovery process to get to the same state as the other members (donors).

Now we will start the Group Replication service on server 1. We will bootstrap the group using these commands:

```
mysql> SET GLOBAL group_replication_bootstrap_group=ON;
mysql> START GROUP_REPLICATION;
mysql> SET GLOBAL group_replication_bootstrap_group=OFF
```

To avoid starting up more groups, we set `group_replication_bootstrap_group` back to `OFF` after successfully starting the group.

To check the status of the new member, use this command:

```
mysql> SELECT * FROM performance_schema.replication_grc
```

```
+---------------------------+-------------------------
| CHANNEL_NAME              | MEMBER_ID
+---------------------------+-------------------------
| group_replication_applier | d58b2766-ab90-11eb-ba00-0
+------------+------------+-------------+-----------
...+---------------+------------+-------------+------
...| MEMBER_HOST   | MEMBER_PORT | MEMBER_STATE | MEMBE
...+---------------+------------+-------------+------
...| 10.124.33.139 |        3306 | ONLINE       | PRIMA
...+---------------+------------+-------------+------
1 row in set (0.00 sec)
```

Great. So far we've bootstrapped and initiated one group member. Let's proceed to the second server. Make sure you have installed the same MySQL version as on server 1, and add the following settings to the *my.cnf* file:

```
[mysqld]
loose-group_replication_group_name="8dc32851-d7f2-4b63-
loose-group_replication_start_on_boot=OFF
loose-group_replication_local_address="10.124.33.90:330
loose-group_replication_group_seeds="10.124.33.139:3306
10.124.33.90:33061, 10.124.33.224:33061"
loose-group_replication_bootstrap_group=OFF
bind-address = "0.0.0.0"
```

All we've changed is the `group_replication_local_address` ; the other settings remain the same. Note that the other MySQL configurations are required for server 2, and we strongly recommend keeping them the same across all nodes.

With the configurations in place, restart the MySQL service:

```
# systemctl restart mysqld
```

Issue the following commands to configure the credentials for the recovery user on server 2:

```
mysql> SET SQL_LOG_BIN=0;
mysql> CREATE USER 'replication_user'@'%' IDENTIFIED BY
```

```
mysql> GRANT REPLICATION SLAVE ON *. TO 'replication_us
mysql> SET SQL_LOG_BIN=1;
mysql> CHANGE MASTER TO MASTER_USER='replication_user',
MASTER_PASSWORD='PASSWORD' FOR CHANNEL
'group_replication_recovery';
```

Next, add server 2 to the group that we bootstrapped earlier:

```
mysql> START GROUP_REPLICATION;
```

And run the query to check the member's state:

```
mysql> SELECT * FROM performance_schema.replication_grc
```

```
+---------------------------+--------------------------
| CHANNEL_NAME              | MEMBER_ID
+---------------------------+--------------------------
| group_replication_applier | 9e971ba0-ab9d-11eb-afc6-0
| group_replication_applier | d58b2766-ab90-11eb-ba00-0
+------------+------------+-------------+-----------
...+---------------+-------------+--------------+...
...| MEMBER_HOST   | MEMBER_PORT | MEMBER_STATE |...
...+---------------+-------------+--------------+...
...| 10.124.33.90  |        3306 | ONLINE       |...
...| 10.124.33.139 |        3306 | ONLINE       |...
...+---------------+-------------+--------------+...
...+-------------+----------------+
...| MEMBER_ROLE | MEMBER_VERSION |
...+-------------+----------------+
...| SECONDARY   | 8.0.22         |
...| PRIMARY     | 8.0.22         |
...+-------------+----------------+
2 rows in set (0.00 sec)
```

Now you can follow the same steps for server 3 as we used for server 2, again updating the local address. When you're done, you can validate whether all the servers are responsive by inserting some dummy data:

```
mysql> CREATE DATABASE learning_mysql;
```

```
Query OK, 1 row affected (0.00 sec)

mysql> USE learning_mysql

Database changed

mysql> CREATE TABLE test (i int primary key);

Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO test VALUES (1);

Query OK, 1 row affected (0.00 sec)
```

Then connect to the other servers to see whether you can visualize the data.

Synchronous Replication

Synchronous replication is used by Galera Clusters, where we have more than one MySQl server, but they act as a single entity for the application. Figure 13-2 illustrates a Galera Cluster with three nodes.

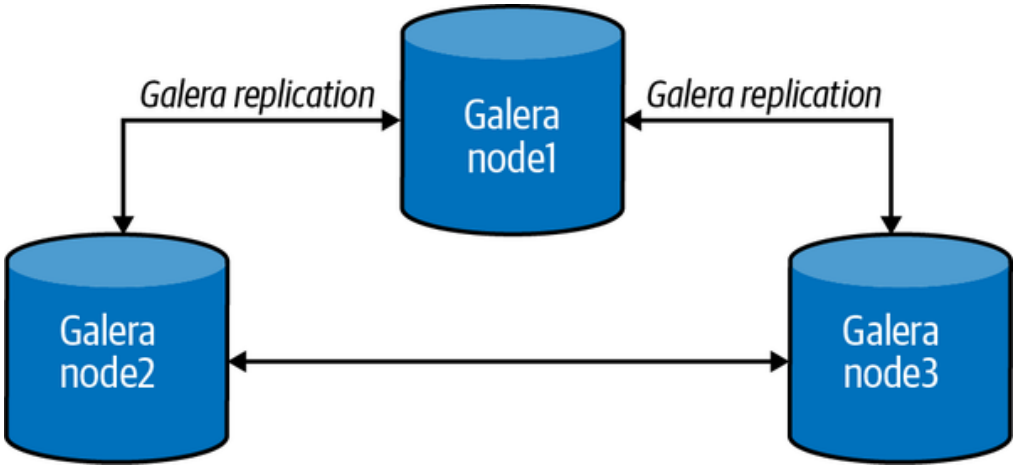

Figure 13-2. In a Galera Cluster, all nodes communicate with each other

The primary difference between synchronous and asynchronous replication is that synchronous replication guarantees that if a change happens on one node

in the cluster, then the change will happen on the other nodes in the cluster synchronously, or at the same time. Asynchronous replication gives no guarantees about the delay between applying changes on the source node and propagating those changes to replica nodes. The delay with asynchronous replication can be short or long. This also implies that if the source node crashes in an asynchronous replication topology, some of the latest changes may be lost. This concepts of source and replica do not exist in a Galera Cluster. All nodes can receive reads and writes.

Theoretically, synchronous replication has several advantages over asynchronous replication:

- Clusters utilizing synchronous replication are always highly available. If one of the nodes crashes, then there will be no data loss. Additionally, all cluster nodes are always consistent.
- Clusters utilizing synchronous replication allow transactions to be executed on all nodes in parallel.
- Clusters utilizing synchronous replication can guarantee causality across the whole cluster. This means that if a `SELECT` is executed on one cluster node after a transaction is executed on another cluster node, it should see the effects of that transaction.

However, there are disadvantages to synchronous replication as well. Traditionally, eager replication protocols coordinate nodes one operation at a time, using a two-phase commit or distributed locking. Increasing the number of nodes in the cluster leads to growth in the transaction response times and the probability of conflicts and deadlocks among the nodes. This is because all nodes need to certify the transaction and reply with an OK message.

For this reason, asynchronous replication remains the dominant replication protocol for database performance, scalability, and availability. Not understanding or underestimating the impact of synchronous replication is one reason companies sometimes give up using Galera Clusters and go back to using asynchronous replication.

At the time of writing, two companies support the Galera Cluster: Percona and MariaDB. The following example shows how to set up a Percona XtraDB Cluster.

Galera/PXC Cluster

Installing Percona XtraDB Cluster (PXC) is similar to installing Percona Server (the difference is the packages), so we won't dive into details for all platforms. You may want to revisit Chapter 1 to review the installation process. The configuration process we'll follow here assumes there are three PXC nodes.

Table 13-1. The three PXC nodes

Node	Host	IP
Node 1	pxc1	172.16.2.56
Node 2	pxc2	172.16.2.198
Node 3	pxc3	172.16.3.177

Connect to one of the nodes and install the repository:

```
# yum install https://repo.percona.com/yum/percona-rele
ch.rpm -y
```

With the repository installed, install the binaries:

```
# yum install Percona-XtraDB-Cluster-57 -y
```

Next, you can apply the typical configurations that you would use for a regular MySQL process (see Chapter 11). With the changes made, start the `mysqld` process and get the temporary password:

```
# 'systemctl start mysqld'
# 'grep temporary password'/var/log/mysqld.log'
```

Use the previous password to log in as `root` and change the password:

```
$ mysql -u root -p
```

```
mysql> ALTER USER 'root'@'localhost' IDENTIFIED BY 'P@s
```

Stop the `mysqld` process:

```
# systemctl stop mysql
```

Repeat the previous steps for the other two nodes.

With the binaries and basic configuration in place, we can start working on the cluster parameters.

We need to add the following configuration variables to */etc/my.cnf* on the first node:

```
[mysqld]
wsrep_provider=/usr/lib64/galera3/libgalera_smm.so
wsrep_cluster_name=pxc-cluster
wsrep_cluster_address=gcomm://172.16.2.56,172.16.2.198

wsrep_node_name=pxc1
wsrep_node_address=172.16.2.56

wsrep_sst_method=xtrabackup-v2
wsrep_sst_auth=sstuser:P@ssw0rd!

pxc_strict_mode=ENFORCING

binlog_format=ROW
default_storage_engine=InnoDB
innodb_autoinc_lock_mode=2
```

Use the same configuration for the second and third nodes, except the `wsrep_node_name` and `wsrep_node_address` variables.

For the second node, use:

```
wsrep_node_name=pxc2
wsrep_node_address=172.16.2.198
```

For the third node, use:

```
wsrep_node_name=pxc3
wsrep_node_address=172.16.3.177
```

Like regular MySQL, Percona XtraDB Cluster has many configurable parameters, and the ones we've shown are the minimal settings to start the cluster. We are configuring the node's name and IP address, the cluster address, and the user that will be used for internal communication among the nodes. You can find more detailed information in the documentation.

We have all the nodes configured at this point, but the `mysqld` process is not running on any node. PXC requires you to start one node in a cluster as a reference point for the others before the other nodes can join and form the cluster. This node must be started in *bootstrap* mode. Bootstrapping is an initial step to introduce one server as a primary component so the others can use it as a reference point to sync up data.

Start the first node with the following command:

```
# systemctl start mysql@bootstrap
```

Before adding other nodes to your new cluster, connect to the node that you just started, create a user for State Snapshot Transfer (SST), and provide the necessary privileges for it. The credentials must match those specified in the `wsrep_sst_auth` configuration that you set previously:

```
mysql> CREATE USER 'sstuser'@'localhost' IDENTIFIED BY
mysql> GRANT RELOAD, LOCK TABLES, PROCESS, REPLICATION
    -> TO 'sstuser'@'localhost';
mysql> FLUSH PRIVILEGES;
```

NOTE

The SST process is used by the cluster to provision nodes by transferring a full data copy from one node to another. When a new node joins the cluster, the new node initiates an SST to synchronize its data with a node that is already part of the cluster.

After this, you can initialize the other nodes regularly:

```
# systemctl start mysql
```

To verify that the cluster is up and running fine, we can perform a few checks, like creating a database on the first node, creating a table and inserting some data on the second node, and retrieving some rows from that table on the third node. First, let's create the database on the first node (`pxc1`):

```
mysq> CREATE DATABASE learning_mysql;
```

```
Query ok, 1 row affected (0.01 sec)
```

On the second node (`pxc2`), create a table and insert some data:

```
mysql> USE learning_mysql;
```

```
Database changed
```

```
mysql> CREATE TABLE example (node_id INT PRIMARY KEY, r
```

```
Query ok, 0 rows affected (0.05 sec)
```

```
mysql> INSERT INTO learning_mysql.example VALUES (1, "\
```

```
Query OK, 1 row affected (0.02 sec)
```

Then retrieve some rows from that table on the third node:

```
mysql> SELECT * FROM learning_mysql.example;
```

```
+---------+-----------+
| node_id | node_name |
```

```
+---------+-----------+
|       1 | Vinicius1 |
+---------+-----------+
1 row in set (0.00 sec)
```

Another, more elegant solution is checking the `wsrep_%` global status variables, in particular `wsrep_cluster_size` and `wsrep_cluster_status`:

```
mysql> SHOW GLOBAL STATUS LIKE 'wsrep_cluster_size';
```

```
+--------------------+-------+
| Variable_name      | Value |
+--------------------+-------+
| wsrep_cluster_size | 3     |
+--------------------+-------+
1 row in set (0.00 sec)
```

```
mysql> SHOW GLOBAL STATUS LIKE 'wsrep_cluster_status';
```

```
+----------------------+---------+
| Variable_name        | Value   |
+----------------------+---------+
| wsrep_cluster_status | Primary |
+----------------------+---------+
1 row in set (0.00 sec)
```

The output of these commands tells us that the cluster has three nodes and is in the primary state (it can receive reads and writes).

You might consider using ProxySQL in addition to the Galera Cluster to ensure transparency for the application (see Chapter 15).

The goal of this chapter was just to familiarize you with the different topologies so you know they exist. Cluster maintenance and optimization are advanced topics that are beyond the scope of this book.