**4.3.1. Hot Standby**

The easiest of the topologies for duplicating servers is the hot standby topology. This topology is shown in [**Figure 4-2**](javascript:moveTo('master_with_a_hot_standby');) and consists of the master and a dedicated server called a hot standby that duplicates the main master. The hot standby server is connected to the master as a slave, and it reads and applies all changes.

The idea is that when the main master fails, the hot standby provides a faithful replica of the master, and all the clients and slaves can therefore be switched over to the hot standby and continue operating. As with many ideas, the reality is not always that rosy.

##### Figure 4-2. Master with a hot standby

Failure is inevitable, at least when you run a large deployment. It is not a question of *if* servers fail, but *when* and *how often* they fail. If the master fails for any reason, it should not bring the deployment to a halt. To ensure operations proceed, it is necessary to have a hot standby server available and to redirect all slaves to the hot standby when the main master fails. This will give you a chance to check what happened with the main master, and maybe fix it or replace it. After you have repaired the master, you have to bring it back on track and either set it to be the hot standby, or redirect the slaves to the original master again.

Sounds simple, doesn’t it? Ah, if only it was that easy—unfortunately, you have the following potential problems to ponder:

* When failing over to the hot standby, you are replicating from a new master, so it will be necessary to translate the binlog positions from those of the original master to those of the hot standby.
* When failing over a slave to a hot standby, the hot standby might actually not have all the changes that the slave has.
* When bringing the repaired master back into the configuration, the repaired master might have changes in the binary log that never left the server.

All these are relevant issues, but for starters, let’s just consider the simpler case illustrated in [**Figure 4-3**](javascript:moveTo('switching_over_from_a_running_master_to');): that of performing a *switchover* from a running master to a standby in order to, for example, perform maintenance on the original master. In this case, the master is still running, so the situation becomes a lot simpler, since we can control the master and make it work for us instead of against us. We will later consider how to handle the case when the master just goes down because its software crashed, a frustrated coworker decided to kick the server, or the janitor tripped over the power cord.

##### Figure 4-3. Switching over from a running master to a standby

By default, events executed by the slave thread are *not* logged to the binary log, which turns out to be a problem if the slave is a standby waiting to be used as a master. In this case, it is necessary to have all the changes sent by the master to the standby written to the binary log of the standby—if not, there will be nothing to replicate. To configure the standby server for this, add the option log-slave-updates to the *my.cnf* file. This option ensures that statements received from the master and executed are also written to the slave’s binary log.

[mysqld]

user = mysql

pid-file = /var/run/mysqld/mysqld.pid

socket = /var/run/mysqld/mysqld.sock

port = 3306

basedir = /usr

datadir = /var/lib/mysql

tmpdir = /tmp

log-bin = master-bin

log-bin-index = master-bin.index

server-id = 1

**log-slave-updates**

After updating the options file, restart the server.

The main problem with switching over to a standby in this case is to perform the switchover in such a way that replication starts at the standby precisely where it stopped replicating on the master. If the positions were easy to translate—for example, if the positions were the same on both the master and the standby—we would not have a problem. Unfortunately, the positions may be different on the master and the standby for a number of reasons. The most common case is when the standby was not attached to the master when the master was started, but even if that is done, events cannot be guaranteed to be written the same way to the binary log on the standby as they were written to the binary log on the master.

The basic idea for performing the switchover is to stop the slave and the standby at exactly the same position and then just redirect the slave to the standby. Because the standby hasn’t made any changes after the position where you stopped it, you can just check the binlog position on the standby and direct the slave to start at that position. However, just stopping the slave and the standby will not guarantee that they are synchronized, so you have to do this manually.

To do this, stop both the slave and the standby and compare the binlog positions. Since both positions refer to positions on the same master—the slave and standby are both connected to the same master—you can check the positions just by comparing the filename and the byte position lexicographically (in that order).

standby> **SHOW SLAVE STATUS\G**

...

Relay\_Master\_Log\_File: ***master-bin.000096***

...

Exec\_Master\_Log\_Pos: ***756648***

1 row in set (0.00 sec)

slave> **SHOW SLAVE STATUS\G**

...

Relay\_Master\_Log\_File: ***master-bin.000096***

...

Exec\_Master\_Log\_Pos: ***743456***

1 row in set (0.00 sec)

In this case, the standby is ahead of the slave, so just write down the slave position of the standby and start the slave to run until it has caught up with the standby. To have the slave catch up with the standby and stop at the right position, use the START SLAVE UNTIL command as we did when stopping the reporting slave earlier in this chapter:

slave> **START SLAVE UNTIL**

-> **MASTER\_LOG\_FILE = 'master-bin.000096',**

-> **MASTER\_LOG\_POS = 756648;**

Query OK, 0 rows affected (0.18 sec)

slave> **SELECT MASTER\_POS\_WAIT('master-bin.000096', 756648);**

Query OK, 0 rows affected (1.12 sec)

The slave and standby have now stopped at exactly the same position, and everything is ready to do the switchover to the standby using CHANGE MASTER TO to direct the slave to the standby and start it. But what position should you specify? Since the file and position that the master recorded for its stopping point are different from the file and position recorded by the standby for the same point, it is necessary to fetch the position that the standby recorded while recording the changes as a master. To do this, execute SHOW MASTER STATUS on the standby:

standby> **SHOW MASTER STATUS\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

File: *standby-bin.000019*

Position: *56447*

Binlog\_Do\_DB:

Binlog\_Ignore\_DB:

1 row in set (0.00 sec)

Now you can redirect the slave to the standby using the correct position:

slave> **CHANGE MASTER TO**

-> **MASTER\_HOST = 'standby-1',**

-> **MASTER\_PORT = 3306,**

-> **MASTER\_USER = 'repl\_user',**

-> **MASTER\_PASSWORD = 'xyzzy',**

-> **MASTER\_LOG\_FILE = '**

***standby-bin.000019*',**

-> **MASTER\_LOG\_POS =** ***56447*;**

Query OK, 0 rows affected (0.18 sec)

slave> **START SLAVE;**

Query OK, 0 rows affected (0.25 sec)

If the opposite is true—that the slave is ahead of the standby—you can just switch the roles of the standby and the slave in the previous steps. This is possible since the master is running and can provide either the slave or the standby with the missing changes. In the next section, we will consider how to handle the situation in which the master has stopped unexpectedly and hence cannot provide either the slave or the standby with the missing changes.

##### 4.3.1.1. Handling a switchover in Python

[**Example 4-1**](javascript:moveTo('procedures_for_switching_to_new_master');) shows the Python code for switching a slave over to another master. The replicate\_to\_position function instructs a server to read from the master only to the given position. When the procedure returns, the slave will have stopped at exactly this position. The switch\_to\_master directs a slave to a new master. The procedure assumes that both the server on which it executes and the new master are connected to the same original master. If they are not, the positions are not comparable and the procedure will raise an exception.

##### Example 4-1. Procedure for switching to a new master

def replicate\_to\_position(server, pos):

server.sql("START SLAVE UNTIL MASTER\_LOG\_FILE=%s, MASTER\_LOG\_POS=%s",

(pos.file, pos.pos))

server.sql("SELECT MASTER\_POS\_WAIT(%s,%s)", (pos.file, pos.pos))

def switch\_to\_master(server, standby):

stop\_slave(server)

stop\_slave(standby)

server\_pos = fetch\_slave\_position(server)

standby\_pos = fetch\_slave\_position(standby)

if server\_pos < standby\_pos:

replicate\_to\_position(server, standby\_pos)

elif server\_pos > standby\_pos:

replicate\_to\_position(standby, server\_pos)

master\_pos = fetch\_master\_position(standby)

change\_master(server, standby, master\_pos)

start\_slave(standby)

start\_slave(server)

#### 4.3.2. Dual Masters

One frequently mentioned setup for high availability is the dual masters topology. In this setup, two masters replicate each other to keep both current. This setup is very simple to use since it is symmetric. Failing over to the standby master does not require any reconfiguration of the main master, and failing back to the main master again when the standby master fails in turn is very easy.

Servers can be either active or passive. If a server is active it means that the server accepts writes, which are likely to be propagated elsewhere using replication. If a server is passive, it does not accept writes and is just following the active master, usually to be ready to take over when it fails.

When using dual masters, there are two different setups, each serving a different purpose:

*Active-active*

In an active-active setup, writes go to both servers, which then transfer changes to the other master.

*Active-passive*

In this setup, one of the masters, called the active master, handles writes while the other server, called the passive master, just keeps current with the active master.

This is almost identical to the hot standby setup, but since it is symmetric, it is easy to switch back and forth between the masters, each taking turns being the active master.

Note that this setup does not necessarily let the passive master answer queries. For some of the solutions that you’ll see in this section, the passive master is a cold standby.

These setups do not necessarily mean that replication is used to keep the servers synchronized—there are other techniques that can serve that purpose. Some techniques can support active-active masters, while other techniques can only support active-passive masters.

The most common use of an active-active dual masters setup is to have the servers geographically close to different sets of users, for example, in offices at different places in the world. The users can then work with the local server, and the changes will be replicated over to the other master so that both masters are kept in sync. Since the transactions are committed locally, the system will be perceived as more responsive. It is important to understand that the transactions are committed locally, meaning that the two masters are not consistent in the sense that they have the same information. The changes committed to one master will be propagated to the other master eventually, but until that has been done, the masters have inconsistent data.

This has two main consequences that you need to be aware of:

* If the same information is updated on the two masters—for example, a user is accidentally added to both masters—there will be a conflict between the two updates and it is likely that replication will stop.
* If a crash occurs while the two masters are inconsistent, some transactions will be lost.

To some extent, you can avoid the problem with conflicting changes by allowing writes to only one of the servers, thereby making the other master a passive master. This is called an active-passive setup—where the active server is called the *primary* and the passive server is called the *secondary*.

Losing transactions when the server crashes is an inevitable result of using asynchronous replication, but depending on the application, it does not necessarily have to be a serious problem. You can limit the number of transactions that are lost when the server crashes by using a new feature in MySQL 5.5 called *semisynchronous replication*. The idea behind semisynchronous replication is that the thread committing a transaction will block until at least one slave acknowledges that it has received the transaction. Since the events for the transaction are sent to the slave after the transaction has been committed to the storage engine, the number of lost transactions can be kept down to at most one per thread.

Similar to the active-active approach, the active-passive setup is symmetrical and therefore allows you to switch easily from the main master to the standby and back. Depending on the way you handle the mirroring, it may also be possible to use the passive master for administrative tasks such as upgrading the server and use the upgrade server as the active master once the upgrade is finished without any downtime at all.

One fundamental problem that has to be resolved when using an active-passive setup is the risk of both servers deciding that they are the primary master—this is called the *split-brain syndrome*. This can occur if network connectivity is lost for a brief period, long enough to have the secondary promote itself to primary, but then the primary is brought online again. If changes have been made to both servers while they are both in the role of primary, there may be a conflict. In the case of using a shared disk, simultaneous writes to the disks by two servers are likely to cause “interesting” problems with the database—that is, probably disastrous and difficult to pinpoint.

##### 4.3.2.1. Shared disks

A straightforward dual masters approach is shown in [**Figure 4-4**](javascript:moveTo('dual_masters_using_a_shared_disk');), where a pair of masters is connected using a shared disk architecture such as a SAN (storage area network). In this approach, both servers are connected to the same SAN and are configured to use the same files. Since one of the masters is passive, it will not write anything to the files while the active master is running as usual. If the main server fails, the standby will be ready to take over.

##### Figure 4-4. Dual masters using a shared disk

The advantage of this approach is that since the binlog files are stored on a shared disk, there is no need for translating binlog positions. The two servers are truly mirror images of each other, but they are running on two different machines. This means that switching over from the main master to the standby is very fast. There is no need for the slaves to translate positions to the new master; all that is necessary is to note the position where the slave stopped, issue a CHANGE MASTER command, and start replication again.

When you fail over using this technique, you have to perform recovery on the tables, since it is very likely updates were stopped midstream. Each storage engine behaves differently in this situation. For example, InnoDB has to perform a normal recovery from the transaction log, as it would in the event of a crash, whereas if you use MyISAM you probably have to repair the tables before being able to continue operation. Of these two choices, InnoDB is preferred because recovery is significantly faster than repairing a MyISAM table.

[**Example 4-2**](javascript:moveTo('procedure_to_remaster_a_slave_when_using');) shows a Python script for handling such a failover using the Replicant library. Notice that the position uses the server ID of the main server, but since both servers are using the same files, the standby server is really a mirror image of the main server. Since the position contains the server ID as well, this will also catch any mistakes made by the user, such as passing a master that is not a mirror image of the main master.

##### Example 4-2. Procedure to remaster a slave when using a shared disk

|  |
| --- |
| def remaster\_slave(slave, master):  position = fetch\_slave\_position(slave)  change\_master(slave, master, position) |

The ability to set up dual masters using shared disks is dependent on the shared storage solution used, a discussion that is beyond the scope of this book.

The problem with using shared storage is that since the two masters are using the same files for storing data, you have to be very careful when doing any administrative tasks on the passive master. Overwriting the configuration files, even by mistake, can be fatal.

The handling of split-brain syndrome depends on which shared disk solution is used and is beyond the scope of this book. One example, however, occurs when using SCSI, which has support for reserving disks by servers. This allows a server to detect that it is really not the primary anymore by noticing that the disks are reserved by another server.