O'REILLY
ONLamp.com
LAMP: THE OPEN SOURCE WEB PLATFORM

Published on **ONLamp.com** (http://www.onlamp.com/)
http://www.onlamp.com/pub/a/onlamp/2006/04/20/advanced-mysql-replication.html
See this if you're having trouble printing code examples

Advanced MySQL Replication Techniques

by Giuseppe Maxia
04/20/2006

You may know about the MySQL Cluster, which is a complex architecture to achieve high availability and performance. One of the advantages of MySQL Cluster is that each node is a peer to the others, whereas in a normal replicating system you have a master and many slaves, and applications must be careful to write only to the master.

The main disadvantages of MySQL Cluster are (as of MySQL 5.0):

- The database is in memory only, thus requiring more resources than a normal MySQL database. (MySQL 5.1 introduces table spaces, with the capability of storing nonindexed data on disk.)
- Some normal features are not available, such as full-text searches, referential integrity, and transaction isolation levels higher than `READ COMMITTED`.

There are some cases where the MySQL Cluster is the perfect solution, but for the vast majority, replication is still the best choice.

Replication, too, has its problems, though:

- There is a fastidious distinction between master and slaves. Your applications must be replication-aware, so that they will write on the master and read from the slaves. It would be so nice to have a replication array where you could use all the nodes in the same way, and every node could be at the same time master and slave.
- There is the fail-over problem. When the master fails, it's true that you have the slaves ready to replace it, but the process of detecting the failure and acting upon it requires the administrator's intervention.

Fixing these two misfeatures is exactly the purpose of this article. Using features introduced in MySQL 5.0 and 5.1, it is possible to build a replication system where all nodes act as master and slave at the same time, with a built-in fail-over mechanism.

Setting Up a Multimaster Replication System

For those of you not well acquainted with the replication basics, I can refer to an earlier article explaining MySQL replication, and the demanding reader can integrate with the dry but extensive official MySQL replication manual.

Back to business. Consider the situation where you set up a replication system with more than one master. This has been a common scenario over the past several years. Chapters 7 and 8 of Jeremy Zawodny's High Performance MySQL describe such a solution. At the time of the book's publication, though, the necessary technology was not yet available.

One hard-to-solve problem in a multimaster replication is the conflict that can happen with self-generated keys. The `AUTO_INCREMENT` feature is quite convenient, but in a replication environment it will be disruptive. If node A and node B both insert an auto-incrementing key on the same table, conflicts arise immediately.

Rescue comes with recent versions. MySQL 5 introduces a couple of server variables for

replicated auto-increment that address this specific problem and allow for the creation of an array of peer-to-peer nodes with MySQL replication.

Quoting from the manual:

- `auto_increment_increment` controls the increment between successive `AUTO_INCREMENT` values.
- `auto_increment_offset` determines the starting point for `AUTO_INCREMENT` column values.

By choosing non-conflicting values for these variables on different masters, servers in a multiple-master configuration will not use conflicting `AUTO_INCREMENT` values when inserting new rows into the same table. To set up N master servers, set the variables like this:

- Set `auto_increment_increment` to N on each master.
- Set each of the N masters to have a different `auto_increment_offset`, using the values 1, 2, ... , N.

Using those two variables as described in the manual, you can ensure that all nodes in your replication array will use different sequences of auto-incrementing numbers. For example, using `auto_increment_increment = 10` and `auto_increment_offset=3`, the numbers generated when inserting three records will be 3, 13, 23. Using 10, 7, you'll get 7, 17, 27, and so on.

For my four-node array, I set `auto_increment_increment` to 10 for each node, and `auto_increment_offset` to 1 in the first node, 2 in the second, and so on.

This is theoretically clear, but it still isn't clear how I managed to transform these servers into peer-to-peer nodes.

The answer is a circular replication, where each node is master of the following node and slave of the preceding one.

Circular replication with two nodes

In its simplest form, circular replication has two nodes, where each one is at the same time master and slave of the other (Figure 1).

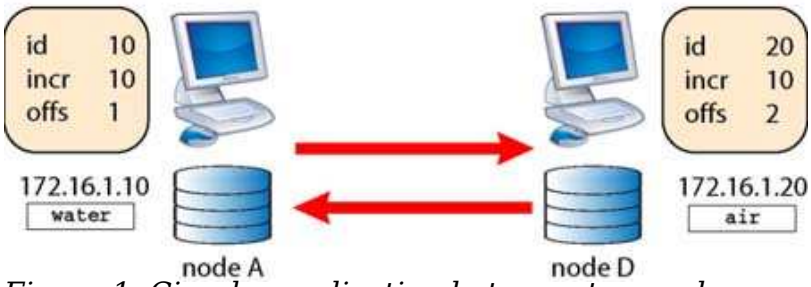

Figure 1. Circular replication between two nodes

For this test, I used two servers in my company (*water* and *air*; there will soon be two more, named *fire* and *earth*). Their basic configuration is:

```
# node A - (water) setup
[mysqld]
server-id                       = 10
# auto_increment_increment      = 10
# auto_increment_offset         = 1
master-host                     = air.stardata.it
master-user                     = nodeAuser
master-password                 = nodeApass

# node B - (air) setup
[mysqld]
server-id                       = 20
# auto_increment_increment      = 10
# auto_increment_offset         = 2
master-host                     = water.stardata.it
master-user                     = nodeBuser
```

```
master-password                 = nodeBpass
```

Notice the two magic variables in the configuration files. If you omit such variables (or comment them, as in this example), then something nasty may happen, and the unfortunate circumstances are easy to demonstrate. Remember that MySQL replication is asynchronous. It means that the replication process in the slave can happen at a different time than the one taking place in the master. This feature makes replication more resilient and ensures that even if you suffer a connection breakdown between master and slave, replication will continue when the slave connection resumes. However, this feature has a nasty side effect when you deal with auto-incremented values. Assume that you have a table like this:

```
CREATE TABLE x (
    id int(11) NOT NULL AUTO_INCREMENT,
    c char(10) DEFAULT NULL,
    PRIMARY KEY (id)
  ) ENGINE=MyISAM DEFAULT CHARSET=latin1
```

Assume also that the connection between node A and node B breaks for a moment. Suppose you issue an `INSERT` statement in both servers, while the replication is not working and the `auto_increment` variables are not set:

```
[node A]   insert into x values (null, 'aaa'), (null, 'bbb'), (null, 'ccc');

[node B]   insert into x values (null, 'xxx'), (null, 'yyy'), (null, 'zzz');
```

When replication resumes, you get a blocking error in both nodes:

```
Error 'Duplicate entry '1' for key 'PRIMARY'' on query. Default database:
    'test'. Query: 'insert into x values (null, 'aaa')'
```

The reason is easy to discover:

```
[node A] select * from x;
+----+------+
| id | c    |
+----+------+
| 1  | aaa  |
| 2  | bbb  |
| 3  | ccc  |
+----+------+


[node B] select * from x;
+----+------+
| id | c    |
+----+------+
| 1  | xxx  |
| 2  | yyy  |
| 3  | zzz  |
+----+------+
```

Both nodes have produced the same primary keys. Thus, when replication resumed, the DBMS justly complained that there was a mistake. Now activate those two variables to see what happens.

```
[node A] set auto_increment_increment = 10;
[node A] set auto_increment_offset    =  1;

[node B] set auto_increment_increment = 10;
[node B] set auto_increment_offset    =  2;
```

Clean the errors, delete all the records in the test table, and redo the insertion (after stopping the replication, to simulate a communication breakdown):

```
[node A] SET GLOBAL SQL_SLAVE_SKIP_COUNTER=1; start slave;
[node B] SET GLOBAL SQL_SLAVE_SKIP_COUNTER=1; start slave;
[node A] truncate x;
[node A] stop slave ;
[node B] stop slave ;

[node A]   insert into x values (null, 'aaa'), (null, 'bbb'), (null, 'ccc');

[node B]   insert into x values (null, 'xxx'), (null, 'yyy'), (null, 'zzz');
```

Now the situation is different.

```
[node A] select * from x;
+-----+------+
```

```
|  id | c    |
+-----+------+
|   1 | aaa  |
|  11 | bbb  |
|  21 | ccc  |
+-----+------+

[node B] select * from x;
+-----+------+
|  id | c    |
+-----+------+
|   2 | xxx  |
|  12 | yyy  |
|  22 | zzz  |
+-----+------+
```

Thus, when replication resumes, there is no conflicting error. This proves it. Choosing appropriate values for the `auto_increment_increment` and `auto_increment_offset` server variables prevents conflicts between auto-generated keys in this circular replication setup. QED.

Adding more nodes

Having just two nodes could be what you need today, but as your application grows and you need to scale your database environment, you will need to add more nodes into the mix. It's easy to extend this array to use four nodes (Figure 2).

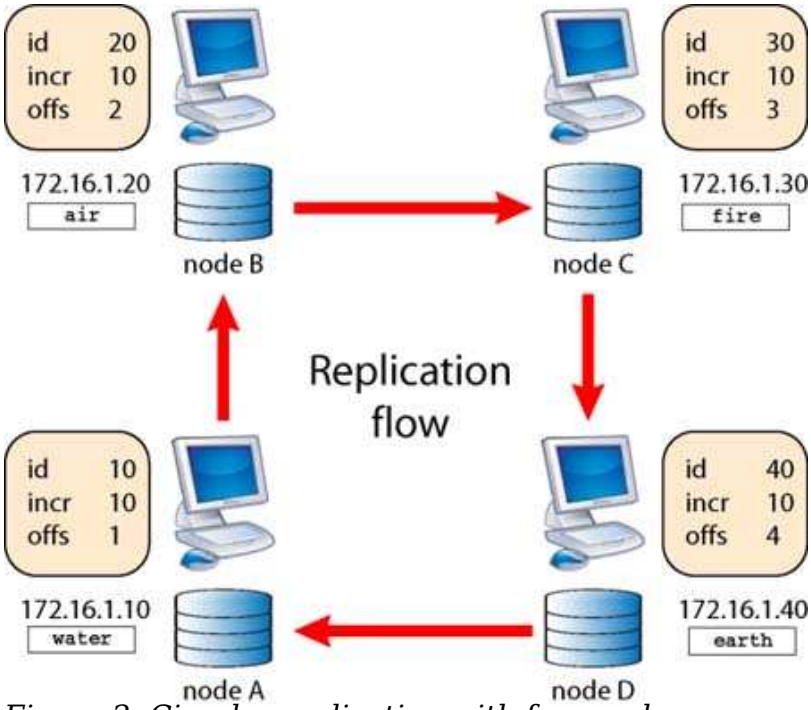

Figure 2. Circular replication with four nodes

In this broader schema, *water* is the master of *air* and slave of *earth*, which is slave of *fire*; and this last is in turn slave of *air*, thus completing the circle. The boxed numbers next to each server indicate the server ID, which must be different for each node; the `auto_increment_increment`, the same for all the nodes; and the `auto_increment_offset`, which guarantees the uniqueness of self-generated keys.

Here is the complete setup for all nodes:

```
# node A - water
[mysqld]
server-id                       = 10
log-bin                         = mysql-bin
log-slave-updates
replicate-same-server-id        = 0
auto_increment_increment        = 10
auto_increment_offset           = 1
master-host                     = earth.stardata.it
master-user                     = nodeAuser
master-password                 = nodeApass
report-host                     = nodeA

# Node B - air
[mysqld]
server-id                       = 20
log-bin                         = mysql-bin
log-slave-updates
replicate-same-server-id        = 0
auto_increment_increment        = 10
auto_increment_offset           = 2
master-host                     = water.stardata.it
master-user                     = nodeBuser
master-password                 = nodeBpass
report-host                     = nodeB

# Node C - fire
[mysqld]
server-id                       = 30
log-bin                         = mysql-bin
log-slave-updates
replicate-same-server-id        = 0
auto_increment_increment        = 10
auto_increment_offset           = 3
master-host                     = air.stardata.it
master-user                     = nodeCuser
master-password                 = nodeCpass
report-host                     = nodeC

# Node D - earth
[mysqld]
server-id                       = 40
log-bin                         = mysql-bin
```

```
log-slave-updates
replicate-same-server-id        = 0
auto_increment_increment        = 10
auto_increment_offset           = 4
master-host                     = fire.stardata.it
master-user                     = nodeDuser
master-password                 = nodeDpass
report-host                     = nodeD
```

A few variables are worth noting in these configuration files. The first is `log-slave-updates`. This option tells each server to write the changes that it receives from its master through the relay binary log to its own binary log. Without it, cascade replication doesn't work. The option `replicate-same-server-id` has the purpose of avoiding infinite replication loops, effectively telling each node to ignore from its master's binary log any statement that originated with its own server ID.

`auto_increment_increment` and `auto_increment_offset` have the appropriate values, as explained earlier. The rest is normal replication administration.

Here's an example of independent usage:

```
[node A] stop slave;
[node B] stop slave;
[node C] stop slave;
[node D] stop slave;
[node A] insert into test.x values (null, 'a'), (null, 'aa'), (null, 'aaa');
[node B] insert into test.x values (null, 'b'), (null, 'bb'), (null, 'bbb');
[node C] insert into test.x values (null, 'c'), (null, 'cc'), (null, 'ccc');
[node D] insert into test.x values (null, 'd'), (null, 'dd'), (null, 'ddd');
```

With the replication stopped, enter in each node three records with independently generated keys. The result is a set of nonconflicting records.

```
[node A] select * from test.x;
+----+------+
| id | c    |
+----+------+
| 1  | a    |
| 11 | aa   |
| 21 | aaa  |
+----+------+
[node B] select * from test.x;
+----+------+
| id | c    |
+----+------+
| 2  | b    |
| 12 | bb   |
| 22 | bbb  |
+----+------+
[node C] select * from test.x;
+----+------+
| id | c    |
+----+------+
| 3  | c    |
| 13 | cc   |
| 23 | ccc  |
+----+------+
[node D] select * from test.x;
+----+------+
| id | c    |
+----+------+
| 4  | d    |
| 14 | dd   |
| 24 | ddd  |
+----+------+
```

When you resume the replication flow, the table contents replicate everywhere:

```
[node A] start slave;
[node B] start slave;
[node C] start slave;
[node D] start slave;
[node A] select * from test.x;
+----+------+
| id | c    |
+----+------+
| 1  | a    |
| 11 | aa   |
| 21 | aaa  |
| 4  | d    |
| 14 | dd   |
| 24 | ddd  |
```

```
| 3  | c    |
| 13 | cc   |
| 23 | ccc  |
| 2  | b    |
| 12 | bb   |
| 22 | bbb  |
+----+------+

[node B] select count(*) from test.x;
+----------+
| count(*) |
+----------+
| 12       |
+----------+

[node C] select count(*) from test.x;
+----------+
| count(*) |
+----------+
| 12       |
+----------+

[node D] select count(*) from test.x;
+----------+
| count(*) |
+----------+
| 12       |
+----------+
```

Of course, if you want to break it, circular replication is as fragile as normal replication when comes to conflicting keys. Inserting the same non-auto-generated primary or unique key in two different nodes will disrupt replication just as well as it does it in normal master-slave replication. With asynchronous replication, this can happen, although you need to be particularly unlucky for this fact to occur. A policy of good programming practice will avoid most of the problems (short of communication failure, that is).

The two important issues here are that you can use circular replication almost seamlessly in any application that now uses a single database server, and that performance is adequate for our purposes.

Measuring circular replication performance

That usage is transparent, but how good is the performance? To answer this question, I had to make some measurements. Rather than clobbering this article unnecessarily, I refer to a recent entry in my blog dealing with this specific issue (Measuring replication speed, with its companion measuring replication speed source code). This tool shows that replicating 1,000 bytes along the whole replication array takes 0.001150 second. To get a reasonable comparison, consider that measuring on the same machine (that is, without replication at all) takes 0.000015 second, while moving the same data between two nodes takes only 0.000065 second.

I would also like to show some more tangible experience, something that makes sense to the average programmer. The method I used for measuring aims at getting a precise figure. However, replication speed can be so fast that it can accommodate even the most demanding programmer needs. Thus, consider an extreme test. Insert three records containing more than 1,000 bytes in one node, and immediately afterward fetch those records from that node's master (Figure 3). Those records will need to navigate through the whole array of nodes before being available.

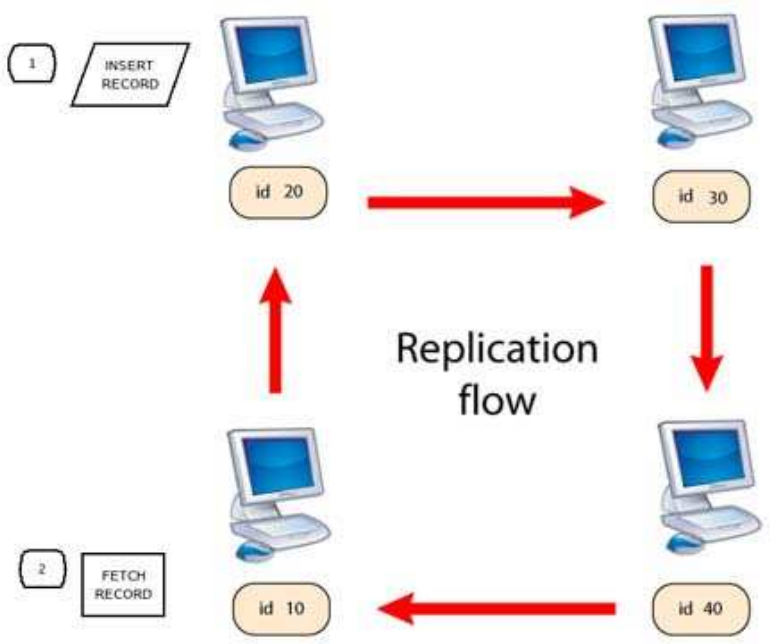

Figure 3. Testing replication speed

The testing code is a Perl script.

```
#!/usr/bin/perl

use strict;
use warnings;

use English qw( -no_match_vars );
use DBI;
use Time::HiRes qw/ usleep /;

my @configuration_files = ();
my $max_config_index    = 3;
my $current_config      = 0;

for ( 'A' .. 'D' ) {
    push @configuration_files,  "$ENV{HOME}/circular_replica/my.node$_.cnf";
}

sub get_connection {
    my ($config_files) = @_;
    my $config_file = $config_files->[$current_config];
    $current_config++;
    if ($current_config > $max_config_index) {
        $current_config = 0;
    }

    my $dbh;
    eval {

        $dbh=DBI->connect("dbi:mysql:test"
            . ";mysql_read_default_file=$config_file",
                undef,
                undef,
                {RaiseError => 1})
         or die "Can't connect: $DBI::errstr\n";
    };
    if ( $EVAL_ERROR ) {
        print STDERR $EVAL_ERROR;
        return;
    }
    return $dbh;
}

my $dbh = get_connection(\@configuration_files);
$dbh->do(qq{truncate x});
my $bigtext = 'a' x 1000;

for my $loop (1 .. 10)
{
    my $dbh1 = get_connection(\@configuration_files);
    my $dbh2 = get_connection(\@configuration_files);
    my ($server1) = $dbh1->selectrow_array(qq{select \@\@server_id});
```

```
    my ($server2) = $dbh2->selectrow_array(qq{select \@\@server_id});
    for (1 .. 3) {
        $dbh2->do( qq{insert into x values (null, concat("server ", ? ), ?)  } , undef , $server1, $bigtext );
    }
    usleep(1);
    my $count = $dbh1->selectrow_array(qq{ select count(*) from x });
    print "inserted a record from server $server2\n ",
          "retrieved $count records from $server1\n";
}
```

After removing all records from the test table, for which there is now a third column (`MEDIUMTEXT`), the code loops through the nodes, getting at each step a node (`$dbh2`) and its master (`$dbh1`). Immediately after inserting three records in the forward node, it calls the master to fetch a count of records, after a stop of one microsecond (`usleep`). A sample session follows:

```
inserted a record from server 30
 retrieved 3 records from 20
inserted a record from server 10
 retrieved 6 records from 40
inserted a record from server 30
 retrieved 9 records from 20
inserted a record from server 10
 retrieved 12 records from 40
inserted a record from server 30
 retrieved 15 records from 20
inserted a record from server 10
 retrieved 18 records from 40
inserted a record from server 30
 retrieved 21 records from 20
inserted a record from server 10
 retrieved 24 records from 40
inserted a record from server 30
 retrieved 27 records from 20
inserted a record from server 10
 retrieved 30 records from 40
```

With this simple demonstration, I believe I have shown that circular replication arrays in MySQL 5 are a viable alternative to expensive clusters. However, circular replication does not scale well. When the number of nodes grows to more than 10 nodes, the speed of replication may not be up to the expectations of normal business operations.

Keep this in mind, though. If a small number of servers are enough for your business, an array of replication nodes could be just what you need.

Automatic Fail-Over

Now I can tackle the second nuisance of current replication, namely the lack of fail-over mechanisms.

For an experienced administrator, changing the master to a group of slaves is a task that takes just a few minutes. However, while changing a master, you will of course realize that you need to replace the master. Such a realization may come to you at an unfortunate hour, and those few minutes of skilled work may become hours of lost sleep. Add to that the possibility of shaking social relations after leaving a party abruptly with your cell phone ringing, driving home dangerously, and eventually achieving a permanently anxious state of mind.

More experienced administrators have put in place monitoring systems that--on demand--can fire a script for changing the master without human intervention. Those scripts, though, depend on the operating system hosting the server and on the availability of the language and tools used for such scripts.

The news that MySQL 5.1 has introduced an event schedule feature embedded in the database server has surely prompted many people into researching a self-contained solution to the fail-over problem. This article is one such effort.

Start by considering a fail-over in normal (noncircular) replication, because it's easier to follow, and in many cases it may be just the missing piece to perfect happiness.

A simple replication scheme is very often a master, a couple of slaves, and a master candidate, which could simply be one of the normal slaves as well. For simplicity sake, assume the situation in Figure 4.

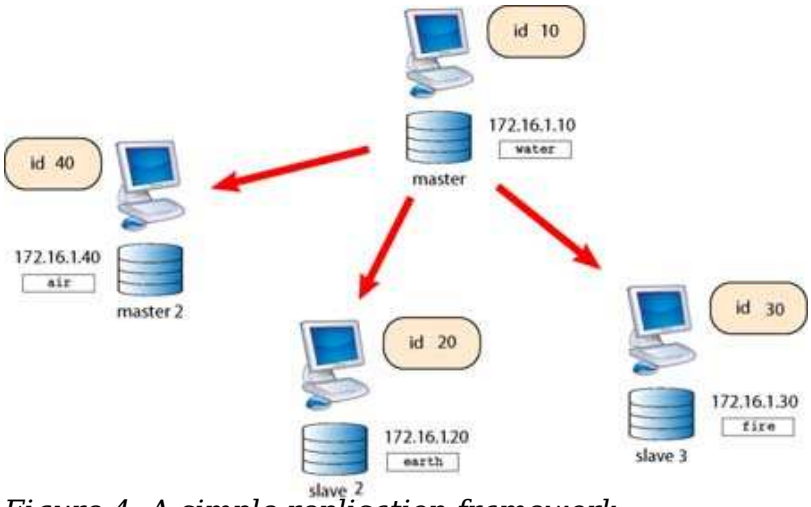

Figure 4. A simple replication framework

To achieve this purpose, you need two recently added features: federated tables, introduced in MySQL 5.0, and the MySQL event scheduler, available from MySQL 5.1.6. Describing these two features extensively would exceed the scope of this article. Fortunately, someone else has already written articles on these subjects, and thus the willing reader can find a gentle introduction to MySQL's federated engine and another about MySQL's event scheduler.

For the purposes of this article, it's enough to say that federated tables are just a link to an existing table in a remote server. The table in the local server has the same structure as the remote one. You must add a special connection string to the creation statement, so that the local database server knows where to look for the data.

The event scheduler is an engine that executes SQL statements in response to the occurrence of temporal events. An *event* is actually a database schema object that combines a *schedule* and an *action*. The schedule defines an optionally recurring point in time, and the action is a SQL statement that must execute according to schedule. Events can be created, altered, enabled, disabled, or dropped through DDL. The database server process maintains event scheduling and execution; it does not depend upon external utilities such as Linux cron jobs or Windows scheduled tasks.

Please note that MySQL 5.1 is, at the writing of this article, still in beta. While the techniques for circular replication are fit for production systems, the features upon which I based the ones I'm now going to show are still likely to have bugs. Before using this technique in production, wait until MySQL 5.1 is released as "generally available." Until then, feel free to experiment.

With these two tools at your disposal, you can build a mechanism to recover the replication system from a master failure. The paradigm is conceptually simple. In the master, in a database `replica`, is a table called `who` with just one column. Any other table would do, though. The only purpose of this table is to exist, so that the slaves can link to it through a federated table called `master_who` (Figure 5).

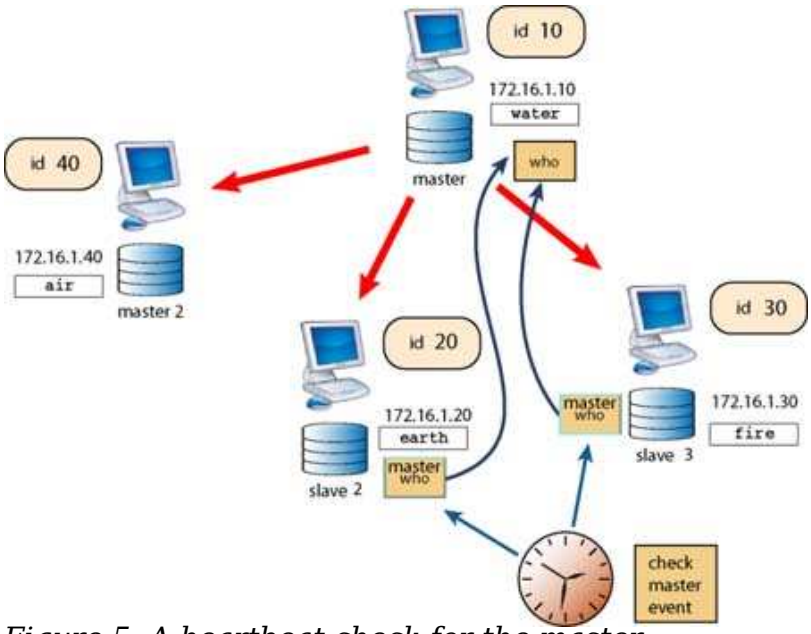

Figure 5. A heartbeat check for the master

```
# in the master
CREATE TABLE who (
  server_id int
) ENGINE=MyIsam

# in each slave
CREATE TABLE master_who (
  server_id int
) ENGINE=Federated
  CONNECTION='mysql://username:password@172.16.1.10:3306/replica/who';
```

The leading character in this scenario, as shown in Figure 5, is a recurring event, set to occur every 30 seconds, called `check master_conn`.

```
create event check_master_conn
    on schedule every 30 second
    enable
    do call check_master();
```

It's as simple as setting a rule in *crontab* (even easier, I daresay). Every 30 seconds, this event calls a stored procedure that tests the status of the master.

```
create procedure check_master()
deterministic
begin
    declare master_dead boolean default false;
    declare curx cursor for
        select server_id from replica.master_who;
    declare continue handler for SQLSTATE 'HY000'
        set master_dead = true;

    open curx;
    # a failure to open the cursor occurs here
    # setting the master_dead variable to true

    if (master_dead) then
        stop slave;
        change master to
            master_host='172.16.1.40',
            master_log_file='mysql-bin.000001',
            master_log_pos=0;
        start slave;
        alter event check_master_conn disable;
    end if;
end
```

To see if the master is alive, this procedure opens a cursor on `master_who` (a federated table pointing to a remote one). If the cursor fails to open, a handler enabled on the specific state related to such failure will set a variable `master_dead` to true. When this happens, the procedure will stop the replication, change the master to the waiting candidate, restart the replication, and finally disable the event, which has fulfilled its role (Figure 6).

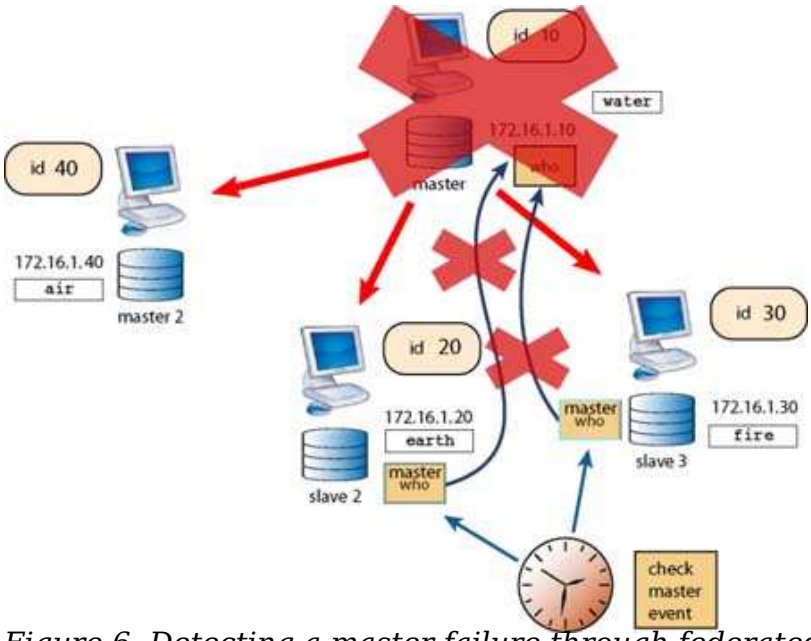

Figure 6. Detecting a master failure through federated tables

Within 30 seconds of the master failure (or less, if you decide to set a lower interval), the old master is forgotten and the new one has happily taken its place (Figure 7).

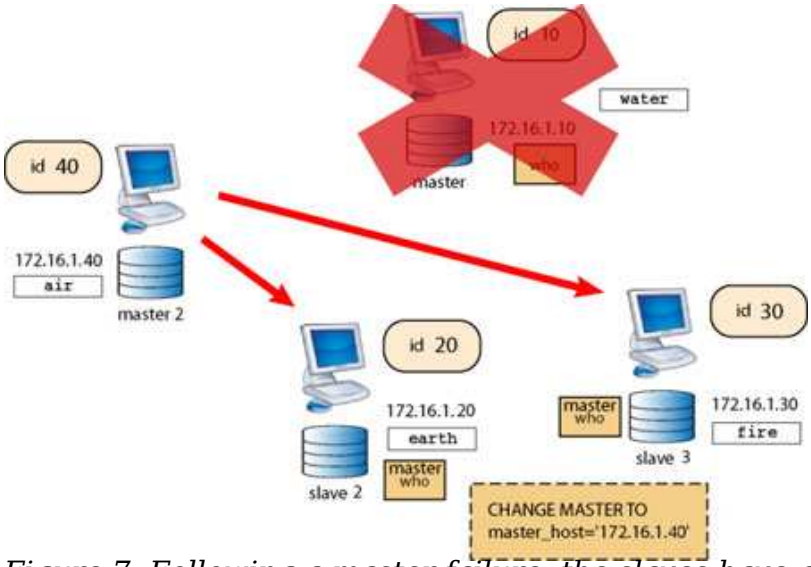

Figure 7. Following a master failure, the slaves have attached to a new master

The administrator is thus spared from social embarrassment, high-speed driving, and a nervous breakdown.

I can see some hands rising from the audience. *What about a secondary fail-over mechanism? What if the candidate master collapses as well?* Take a breath and accept the reality. This is a proof of concept that such a mechanism can work and could spare you pain and fatigue. The cleanup after the failure still needs some manual work. You won't need to rush any longer, that's all.

As for secondary (or chain) failures, while they are highly unlikely, you should not dismiss them. My personal take is to put the best effort into reacting to the primary failure and leave secondary failure to human intervention--but there are cases where it is advisable or even necessary to take such failures into account and to put a plan of automatic recovery in place as well. It's just a matter of complexity and, ultimately, of money, because the more time you invest in dealing with redundant fail-over plans, the more it is going to cost.

You, or your company (or your customer, if you are a consultant), will decide how paranoid you can afford to be. The path is set, and you can take the simple solution explained in this

article and double (or triple) its level of accuracy according to your policy. Everything is feasible. I know by experience that systems with redundant recovery solutions become complex and expensive. They are also beyond the scope of this article.

Circular replication failure recovery

It's important to cover something else: how to recover from a failure in a circular topology, thus achieving a replication array with the major features of a full-fledged cluster. The principle involved is the same, but multiply the practice by the number of nodes (Figure 8).

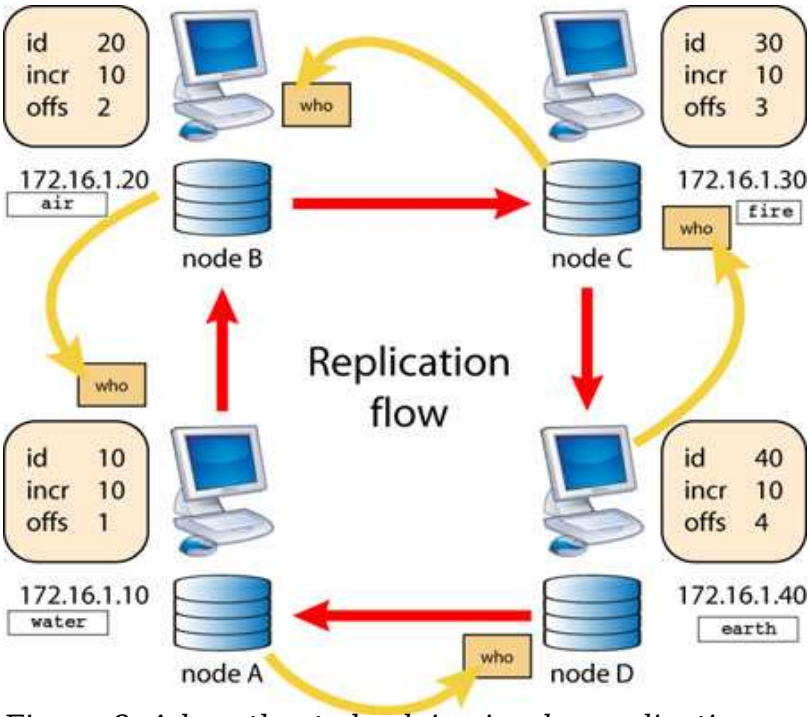

Figure 8. A heartbeat check in circular replication

Instead of having only one table `who`, add one per node and one federated table `master_who` pointing to its corresponding master. The structure of the table is the same as what you have seen before. What changes is the connection string. To set the appropriate table in each node, use a convenient stored procedure.

```
create procedure make_connections()
begin
    drop table if exists master_who;
    case @@server_id
        when 10 then
            CREATE TABLE master_who
                (
                    server_id int not null primary key
                ) ENGINE = federated
                CONNECTION = 'mysql://username:password@earth:3306/replica/who';
        when 20 then
            CREATE TABLE master_who
                (
                    server_id int not null primary key
                ) ENGINE = federated
                CONNECTION = 'mysql://username:password@water:3306/replica/who';
        when 30 then
            CREATE TABLE master_who
                (
                    server_id int not null primary key
                ) ENGINE = federated
                CONNECTION = 'mysql://username:password@air:3306/replica/who';
        when 40 then
            CREATE TABLE master_who
                (
                    server_id int not null primary key
                ) ENGINE = federated
                CONNECTION = 'mysql://username:password@fire:3306/replica/who';
    else
        select "unhandled server id " as "error";
    end case;
```

```
end
```

The event is exactly the same in the normal replication scheme. The called procedure is different:

```
create procedure check_master()
deterministic
begin
    declare master_dead boolean default false;
    declare curx cursor for
        select server_id from replica.master_who;
    declare continue handler for SQLSTATE 'HY000'
        set master_dead = true;
    open curx;

    if (master_dead) then
        stop slave;
        case @@server_id
        when 10 then
            change master to
                master_host     = 'fire';
        when 20 then
            change master to
                master_host     = 'earth';
        when 30 then
            change master to
                master_host     = 'water';
        when 40 then
            change master to
                master_host     = 'air';
           else
               --  report the error in a log table
               insert into check_master_log values (now(), @@server_id,
                       "not handled server id");
        end case;
        start slave;
        alter event check_master_conn disable;
    end if;
end
```

The failure scenario is similar to the previous one. Unlike the previous example, where all slaves had the same failure mechanism and acted together to resume the replication, in this case only one node has to do something. The others will continue as usual (Figure 9).

Note that in the examples throughout this article, I use the server hostnames for clarity. In production servers, though, I always use the server IP address, for performance reasons (it will save some lookup time) and also because I often can use a dedicated high-speed line to connect servers for replication.

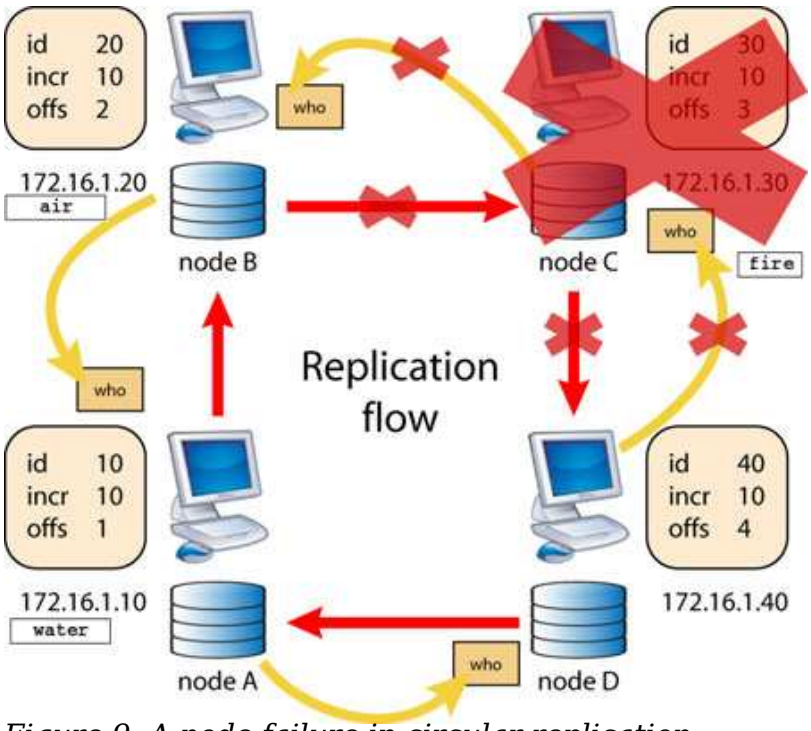

Figure 9. A node failure in circular replication

Node 40 detects that node 30 is no longer active. Thus its `check_master` will switch the master to node 20, *air*, for which business continued as usual (Figure 10).

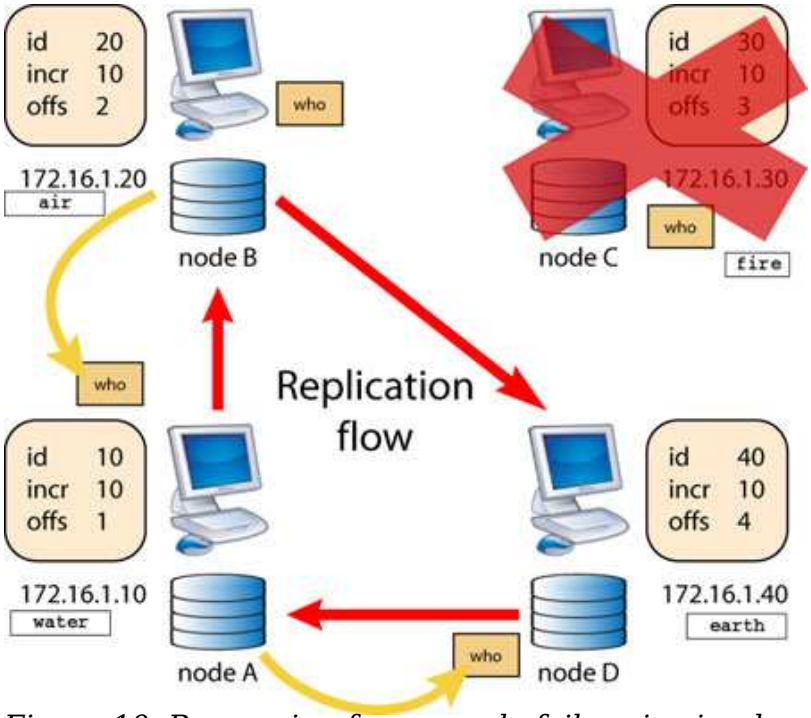

Figure 10. Recovering from a node failure in circular replication

Notice that, after recovering from failure, master checking does not resume in node 40. It's the same principle I have mentioned before, when discussing chain failures in normal replication. In both cases it's possible to set a second level of events and procedures to be called after the first recover, so that the system will survive to a second failure as well. I leave this task as an exercise to the industrious reader.

Fail-Over, Client Side

So far, I have shown how a server replaces a failed one. All is well when seen from within the replication array, but what about the poor client that was pointing at the now defunct master and keeps getting connection time-out errors?

Here comes the bad news: there is no silver bullet for this problem. Because it's on the application side, you must find an application solution. There are a few server-side solutions, but they are either limited to only two nodes or they depend heavily on a specific operating system (such as CARP, UCARP, or Linux-HA).

The consoling news is that MySQL Cluster is no better in this field. It may come as a surprise to many, but the answer for MySQL high availability is to use multiple data sources in Java. Unless some integration between operating system and MySQL comes up, the situation is that you are pretty much on your own when you need a high-availability system.

The good news is that if circular replication is satisfactory for your needs, a simple load balancer will be enough to guarantee at the same time a good spread of resources and a high-availability system. You can buy a hardware load balancer, or you can implement a software one within your application. You may also monitor the logs of the fail-over events we have seen in this article to exclude a dead node from the ones through which your balancer should loop. If you use such an approach and get random connections coming from the balancer, be sure to use them by transaction, not by query. Even if you don't use transactions, use only one connection within the same unit, be it a web page, a module function, or an administration program.

Let me try to explain further. You are using a device that gets you a random database connection to one of the nodes. You could just use such device for each query in your

application, thus spreading the load among all nodes. However, this extreme randomization would not be healthy. It may happen that you do an INSERT query in one node, and then a SELECT query in another node, where the effects of your insert may not yet be replicated. To avoid this, your application should get a connection and use it to execute all queries that are logically related.

Missing pieces

What I have shown in this article is, I believe, a big step forward in the direction of a more useful replication system. There are still several missing pieces to extend this proof of concept into a well-behaved engine.

One, I have to mention that MySQL AB is planning an extension of the current MySQL replication so that it will include some devices to resolve conflicts (such as the clash of unique keys). It is still in early stages, but feel free to explore.

I also need to mention some half-baked features in MySQL 5.*x*. You know that one of MySQL 5.0's major enhancements was the information schema database, which should offer a coherent view of database metadata objects. Unfortunately, a missing piece in this collection of metadata is all the data concerning replication. Therefore, because you can access replication status only through SHOW *SOMETHING* statements (as of today, in MySQL 5.1.9), stored procedures cannot access this information. A further problem is that the parameters of CHANGE MASTER TO must be text literals. Variables are not accepted, thus reducing the flexibility of stored procedures. This inconsistency has been reported through the appropriate channels, and we hope somebody will act on it. For the time being, it all means that you can achieve fine-grained replication administration only through external procedures.

(Actually, that is not exactly true. There are some undocumented and unofficial--even deprecated--practices that can overcome these limitations. For the brave, I will illustrate these techniques in Higher Order MySQL, a talk at the MySQL Users Conference 2006.)

Enhanced circular replication offers additional features that I have not shown here. It is possible, for instance, to exchange messages between servers. That is quite useful in the event of master replacement, when the slave could ask the master to perform a RESET MASTER before resuming replication. I leave these amenities to some other article, to avoid burdening this one too much.

However, let me remind you that the code in this article is just a proof of concept, which needs some hardening before being used for production. A real-world application will need to double-check whether the master is really dead before switching to a new master, and the new master must be questioned before the switch to ensure that it's ready to take over; the other nodes should be informed; and so on. You can perform all these actions using the currently available technology, although they will be more effective and easier to implement when the currently planned improvements on the data dictionary are available--according to the information at hand, in MySQL version 5.1.

Playing with the System

Experimenting with replication is not so easy. You need to have several servers to play with master and slaves. For a test of fail-over, you need at least three servers.

Because not everyone can enjoy an abundance of hardware, I offer you MySQL 5 Replication Playground, where all the nodes are in the same box, and they just use different ports and sockets to simulate separate machines.

Should you wish to play with this system, all you need to do is install one instance of MYSQL 5.1 and then the replication playground in your home directory. (You don't need root access.) Download it, peruse the readme, run the installation script, and play with it.

Acknowledgments

Thanks to Patrizio Tassone and Alberto Coduti for the graphics. Thanks to Roland Bouman, Jay Pipes, and Lars Thalmann for catching my mistakes and for providing useful comments and corrections. I am also indebted to them for some rephrasing and definitions in this text.

Thanks to Massimiliano Stucchi for providing a test environment for FreeBSD and for catching some more mistakes.

Giuseppe Maxia is a freelance database consultant and CTO of StarData.it, an Italian company specializing in open source solutions.

Return to ONLamp.com.