MySQL InnoDB Cluster & MySQL Group Replication in a Nutshell Hands-On Tutorial


ORACLE®

Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purpose only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied up in making purchasing decisions. The development, release and timing of any features or functionality described for Oracle´s product remains at the sole discretion of Oracle.

Who I am ?

Frédéric Descamps

- @lefred
- MySQL Evangelist
- Managing MySQL since 3.23
- devops believer
- http://about.me/lefred

ORACLE

get more online

MySQL Group Replication

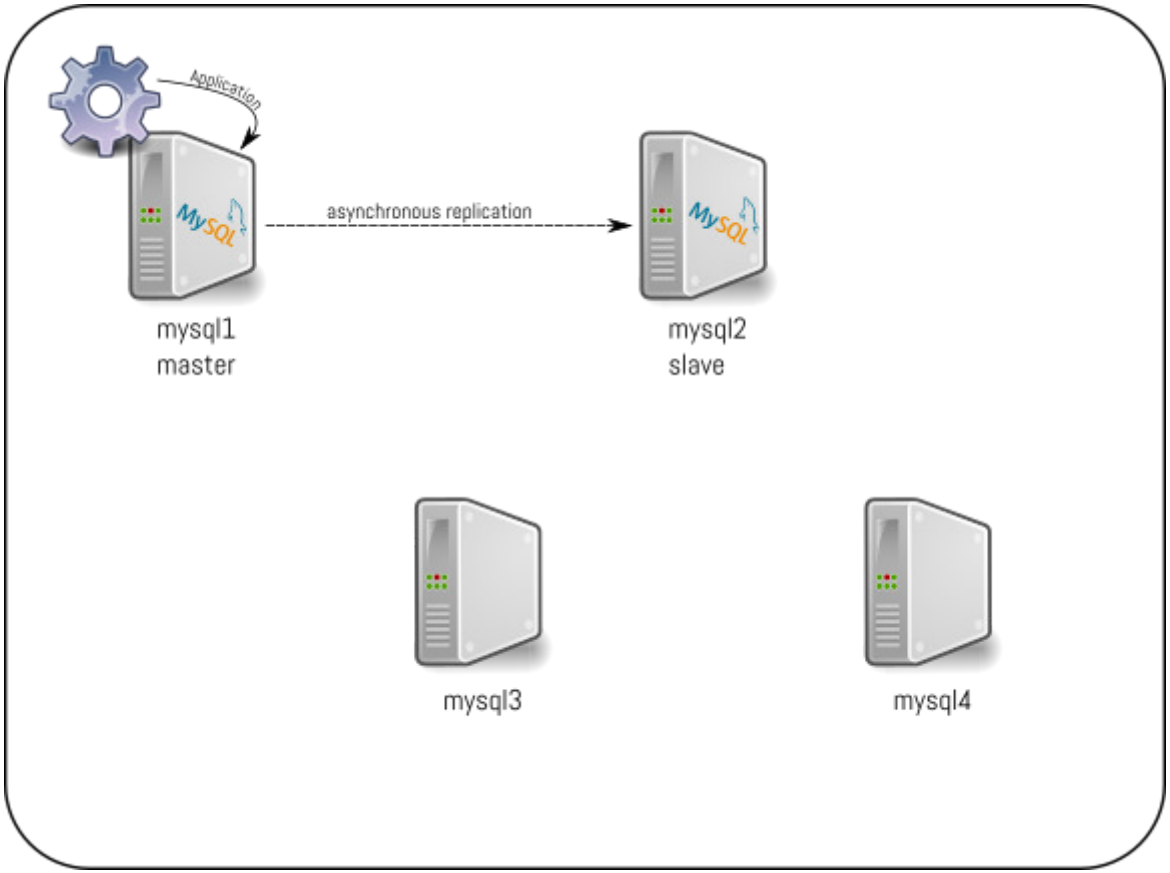
Blogs

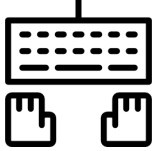
- http://lefred.be/
- http://mysqlhighavailability.com/
- https://thesubtlepath.com/blog/mysql/

Agenda

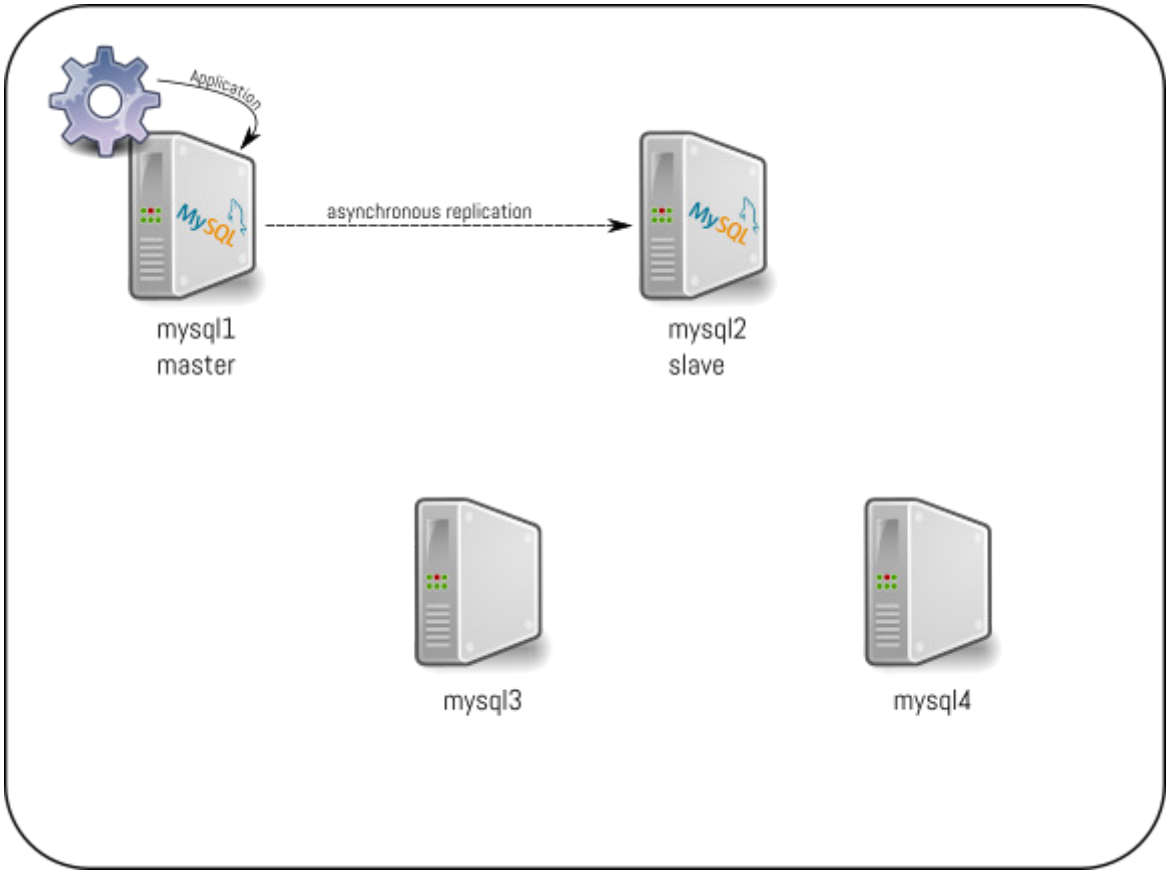
- MySQL InnoDB Cluster & Group Replication concepts
- Migration from Master-Slave to GR
- How to monitor ?
- Application interaction

LAB1: Current situation





LAB1: Current situation



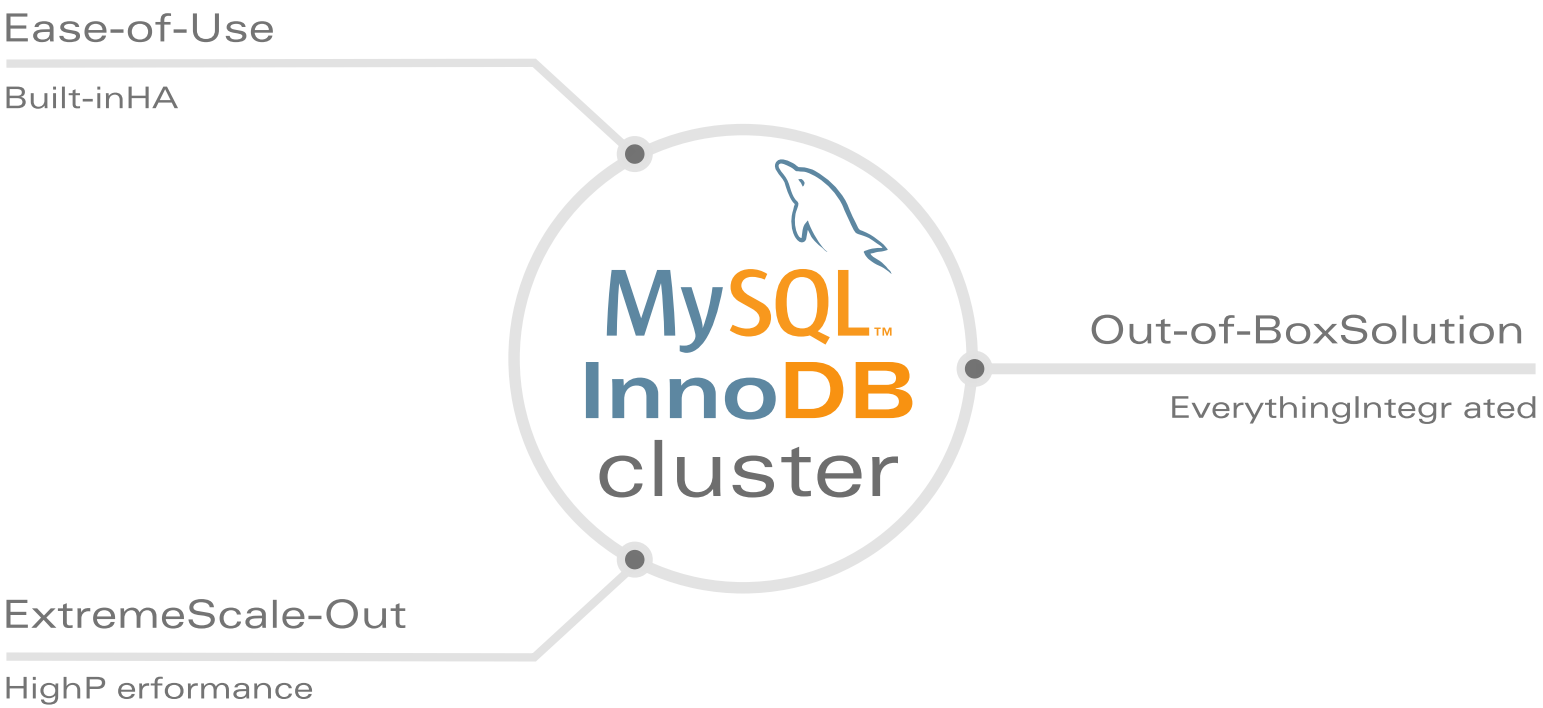
- launch `run_app.sh` on `mysql1` into a **`screen`** session
- verify that `mysql2` is a running slave

Summary

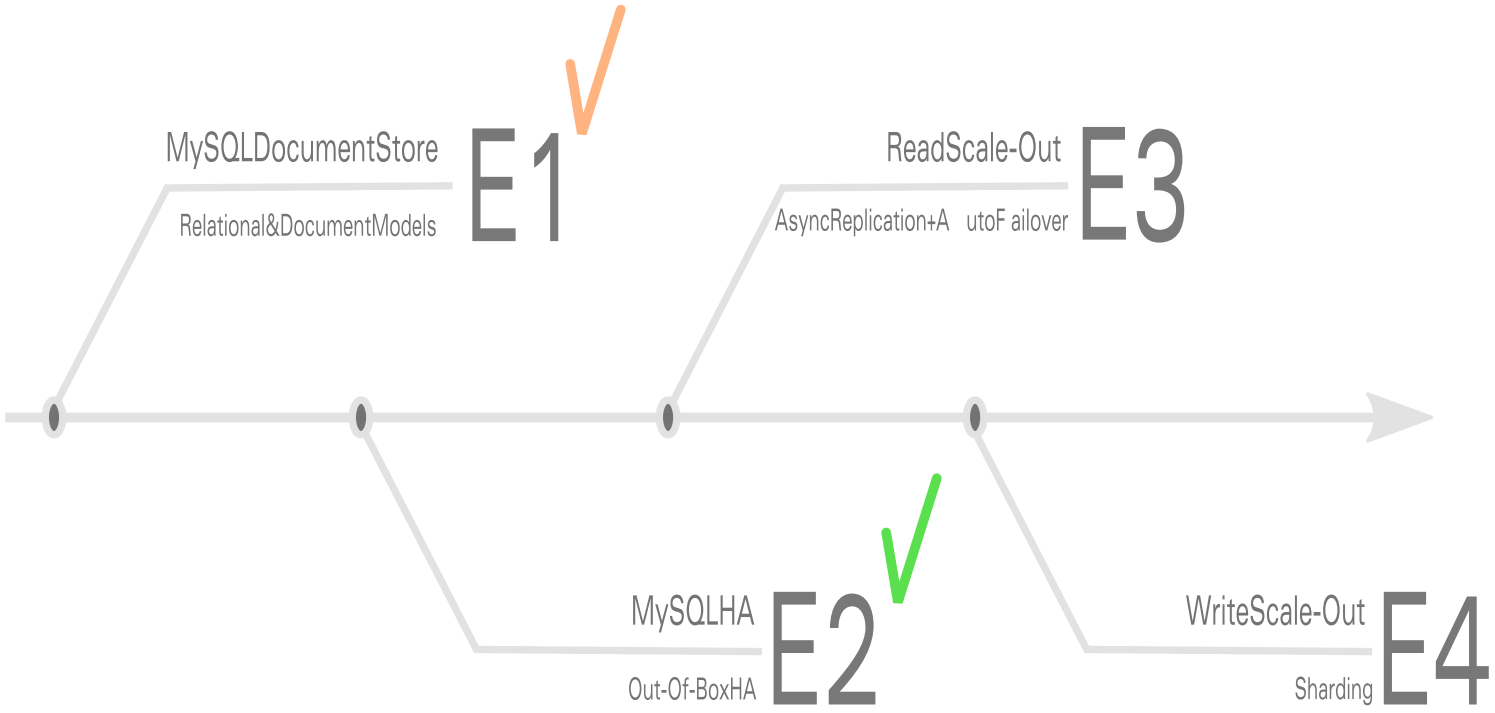
	ROLE	SSH PORT	INTERNAL IP
mysql1	master	8821	192.168.56.11
mysql2	slave	8822	192.168.56.12
mysql3	n/a	8823	192.168.56.13
mysql4	n/a	8824	192.168.56.14

Easy High Availability

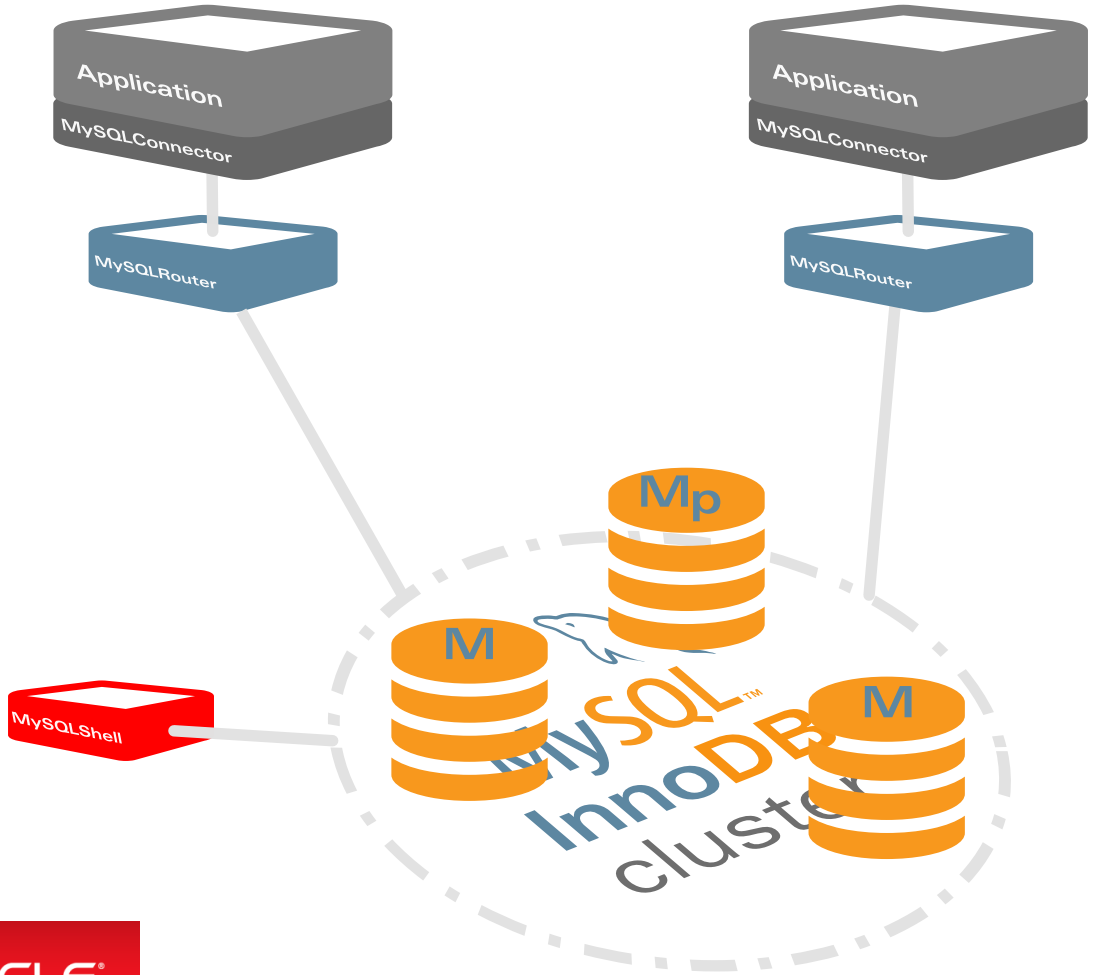
MySQL InnoDB Cluster



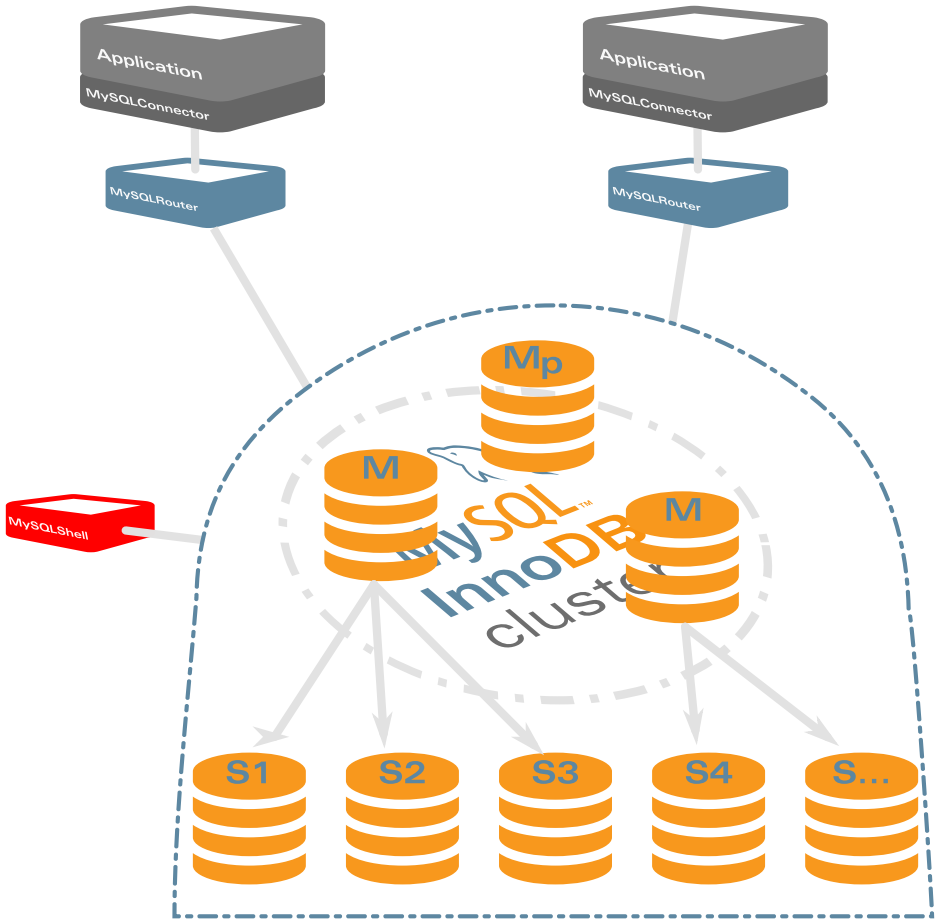
Our vision in 4 steps



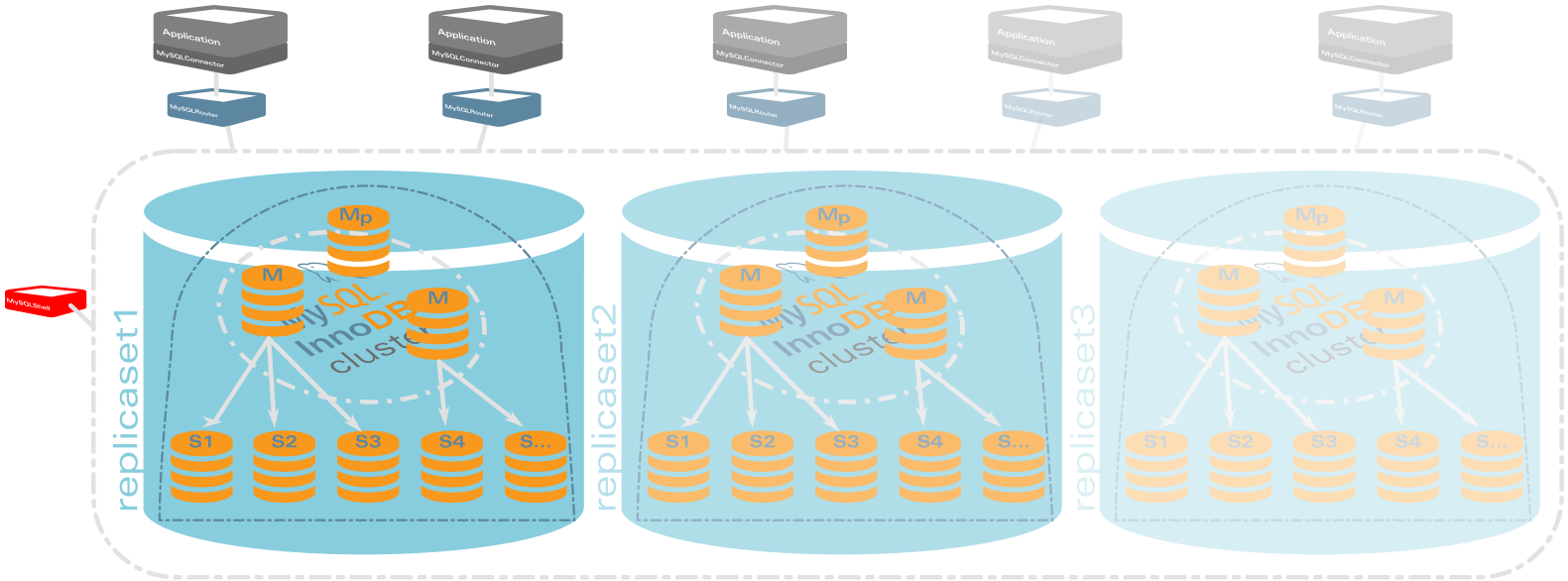
Step 2´s Architecture



Step 3´s Architecture



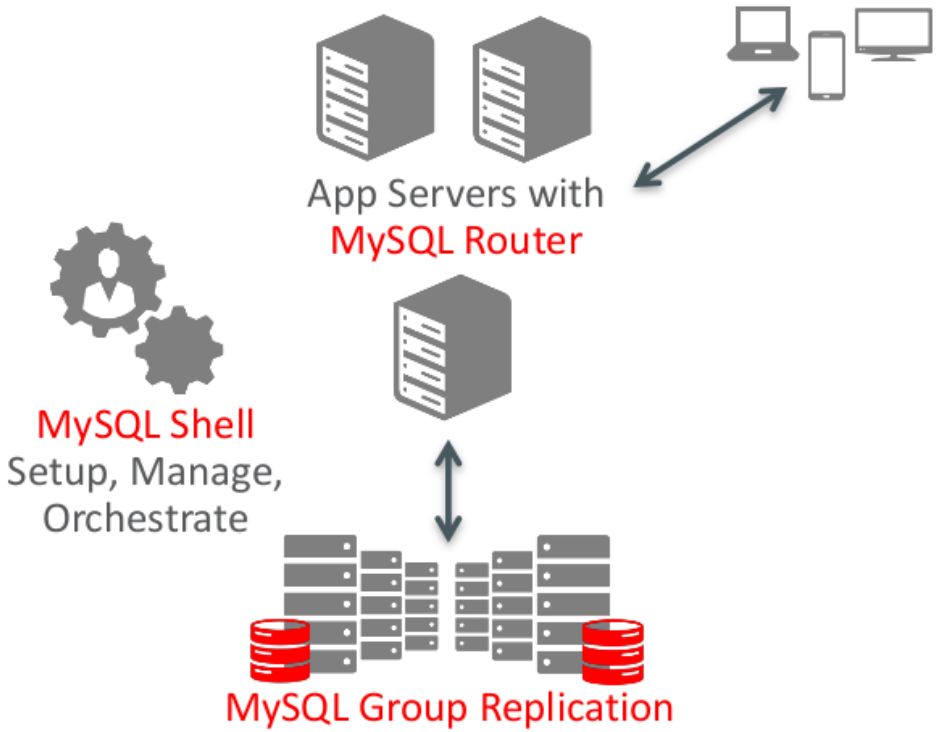
Step 4´s Architecture



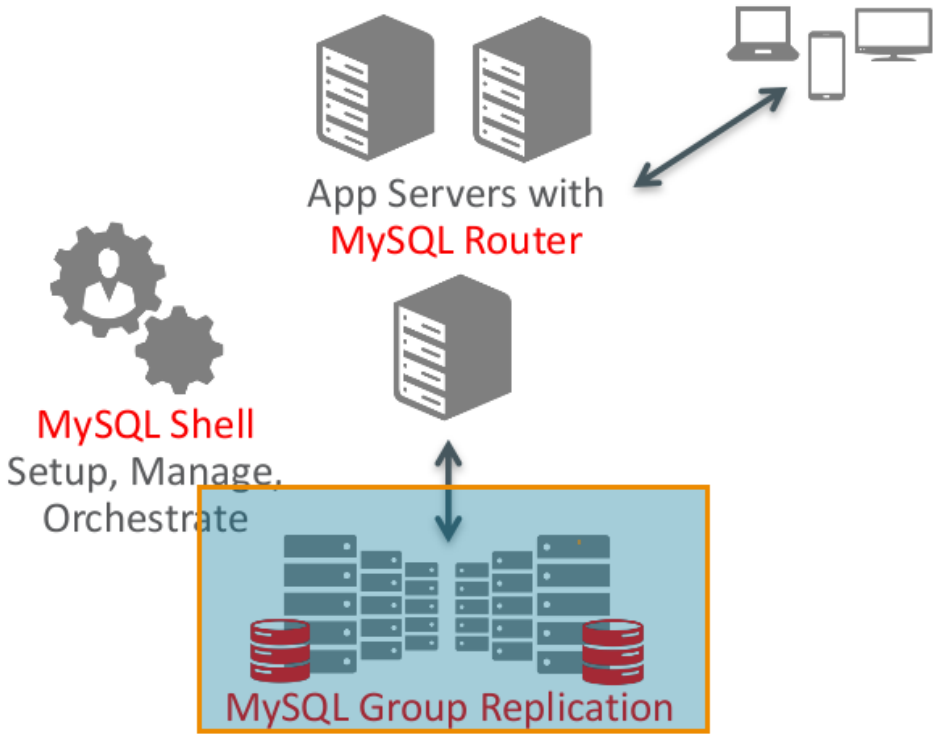
the magic explained

Group Replication Concept

Group Replication: heart of MySQL InnoDB Cluster



Group Replication: heart of MySQL InnoDB Cluster



MySQL Group Replication

but what is it ?!?

- GR is a plugin for MySQL, made by MySQL and packaged with MySQL
- GR is an implementation of Replicated Database State Machine theory
- GR allows to write on all Group Members (cluster nodes) simultaneously while retaining consistency
- GR implements conflict detection and resolution
- GR allows automatic distributed recovery
- Supported on **all MySQL platforms** !!
  - Linux, Windows, Solaris, OSX, FreeBSD

MySQL Group Communication System (GCS)

- MySQL Xcom protocol
- Replicated Database State Machine
- Paxos based protocol (similar to Mencius)
- Its task: Deliver messages accross the distributed system:
  - Atomically
  - in Total Order
- MySQL Group Replication receives the Ordered 'tickets' from this GCS subsystem.

And for users ?

- not longer necessary to handle server fail-over manually or with a complicated script
- GR provides fault tolerance
- GR enables update-everywhere setups
- GR handles crashes, failures, re-connects automatically
- **Allows an easy setup of a highly available MySQL service!**

OK, but how does it work ?

it´s just ...

OK, but how does it work ?

it´s just ...

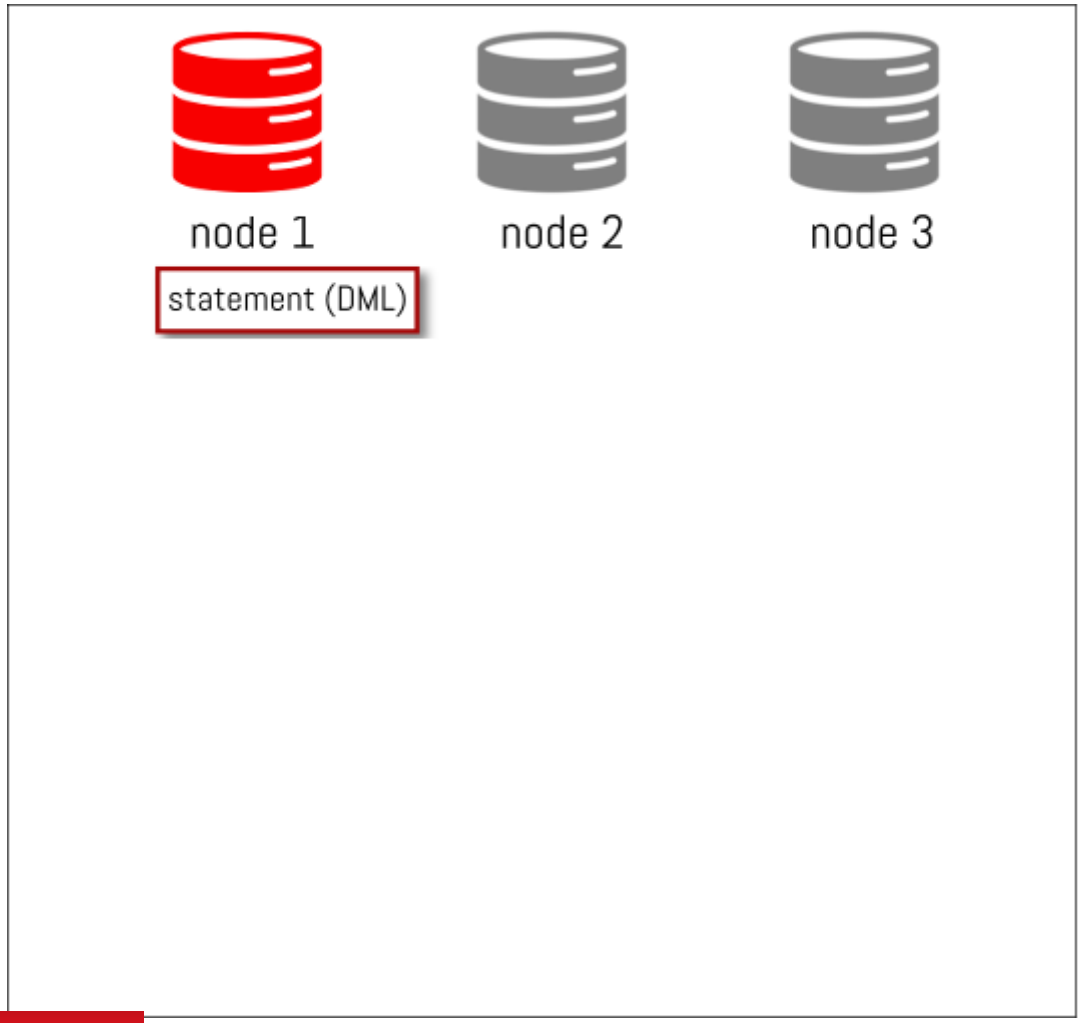
OK, but how does it work ?

it´s just ...

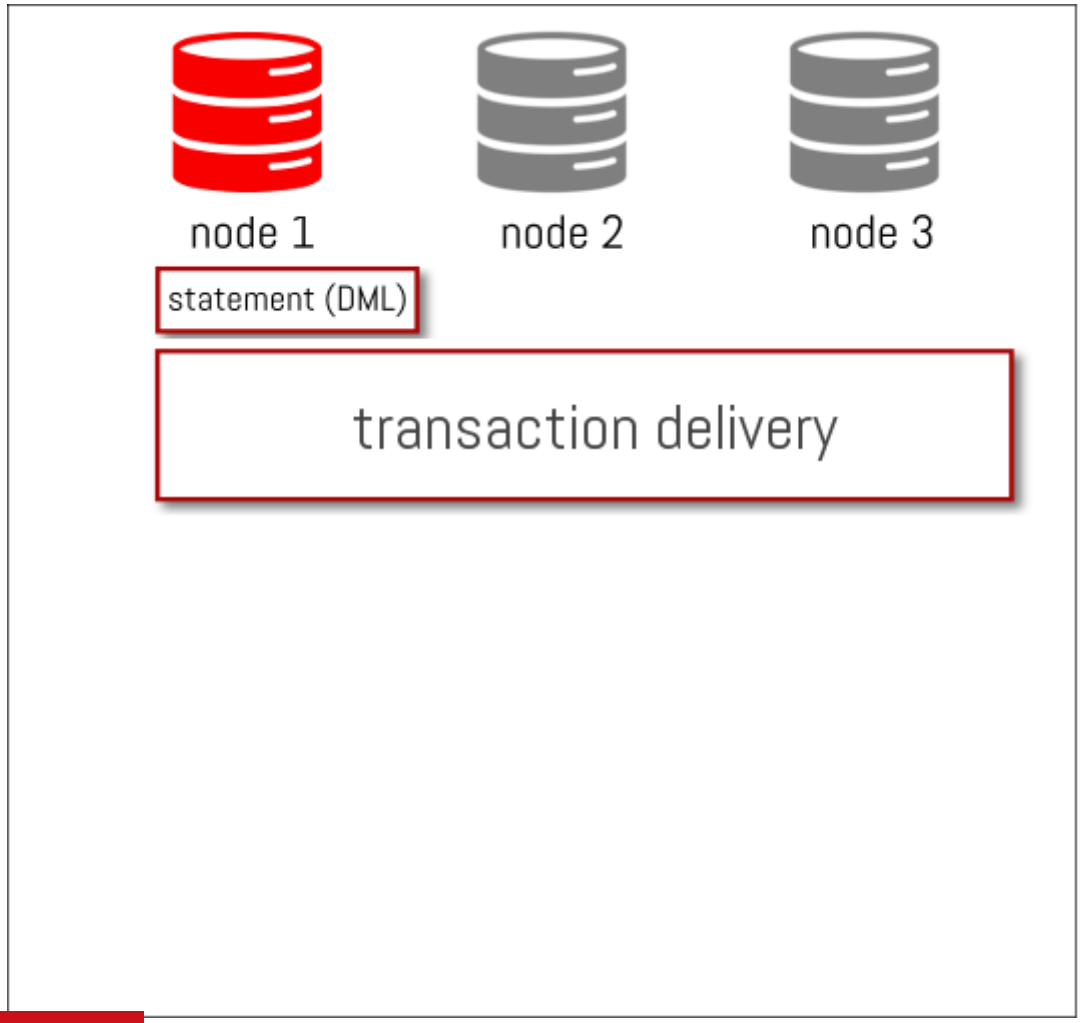
... no, in fact the writeset replication is **synchronous** and then certification and apply of the changes are local to each nodes and happen asynchronous.

not that easy to understand... right ? As a picture is worth a 1000 words, let´s illustrate this...

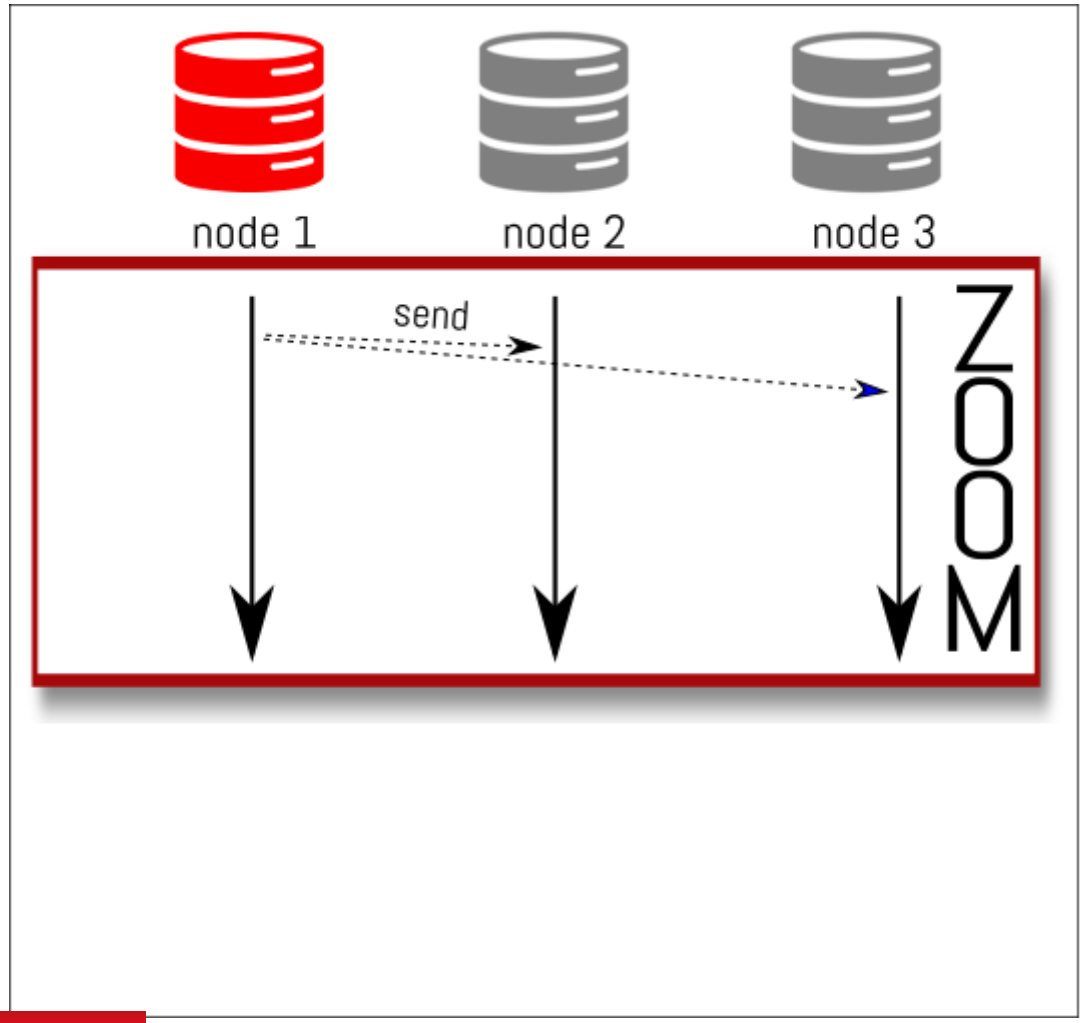
MySQL Group Replication (autocommit)



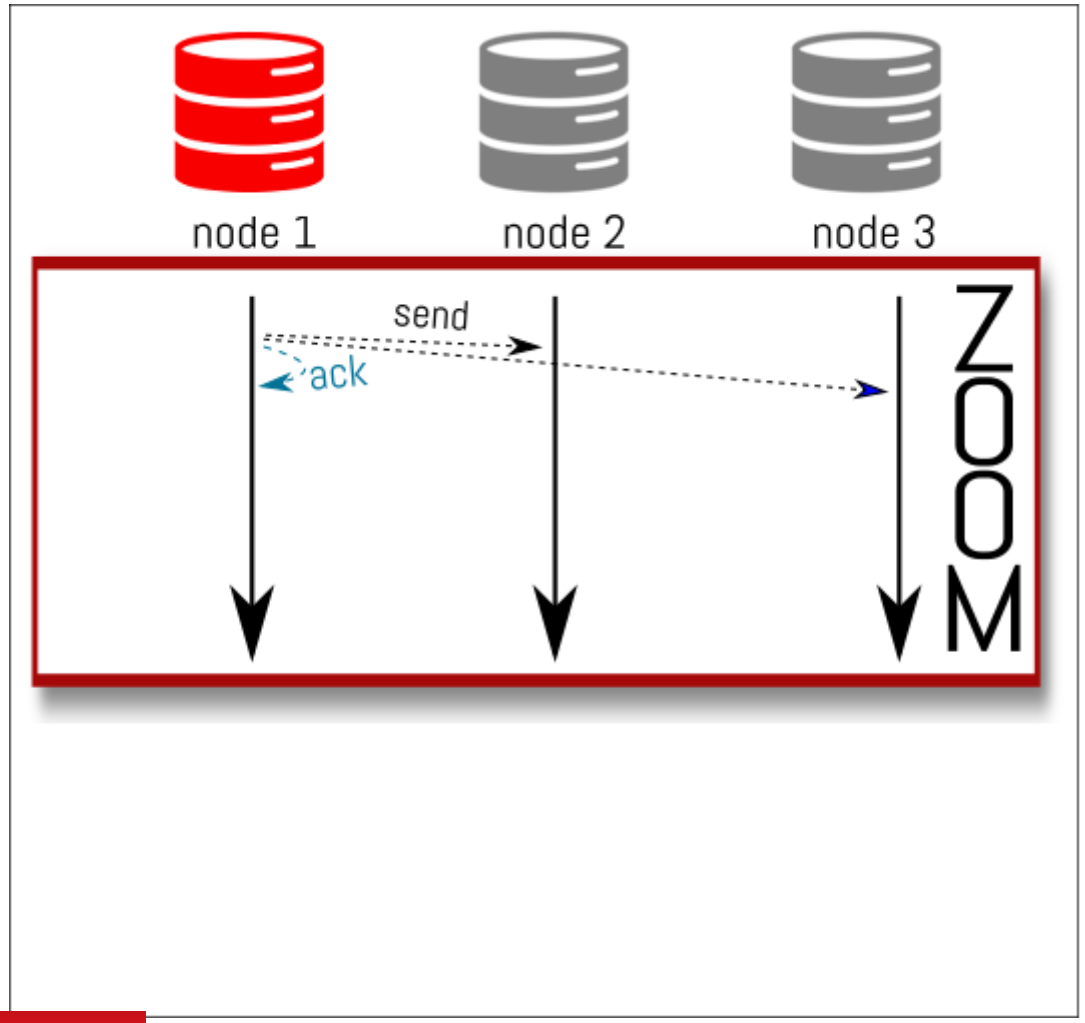
MySQL Group Replication (autocommit)



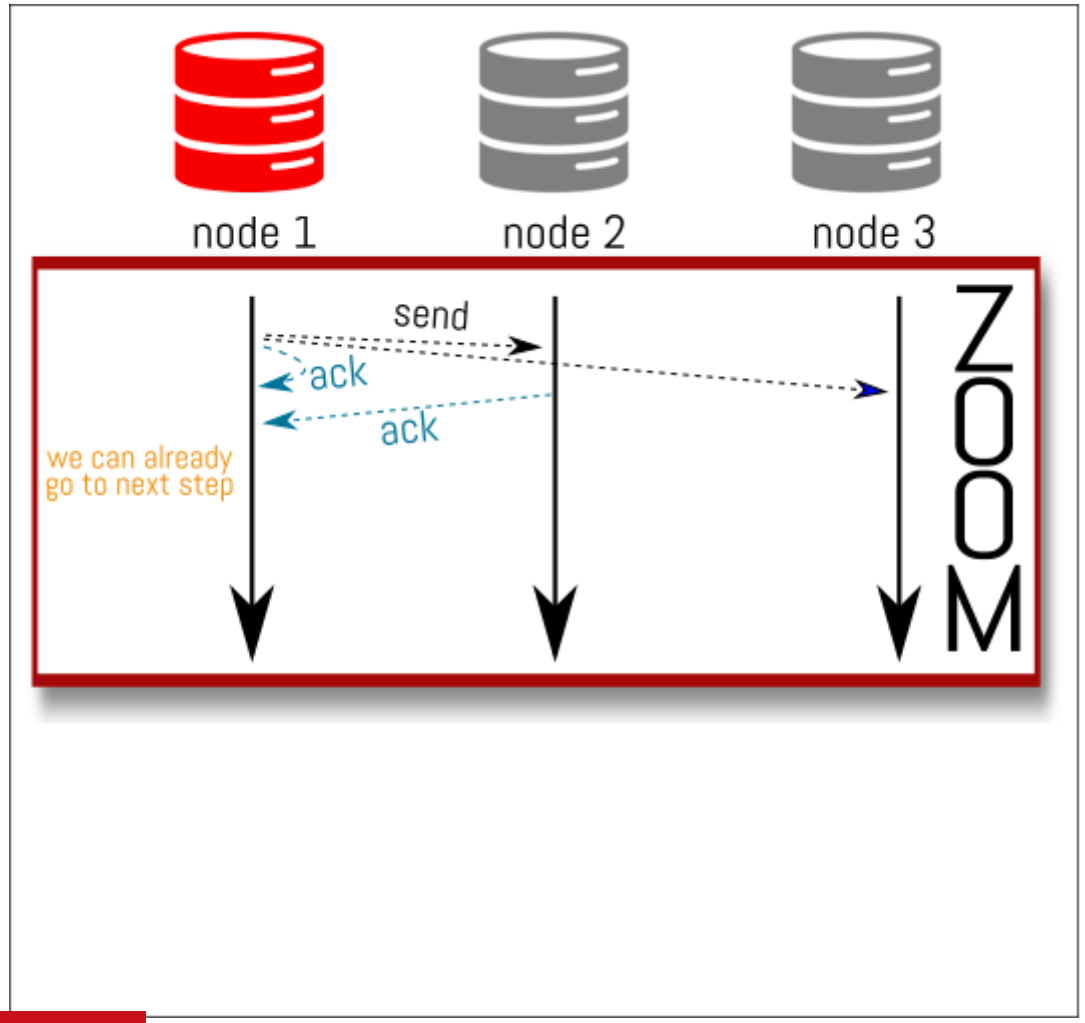
MySQL Group Replication (autocommit)



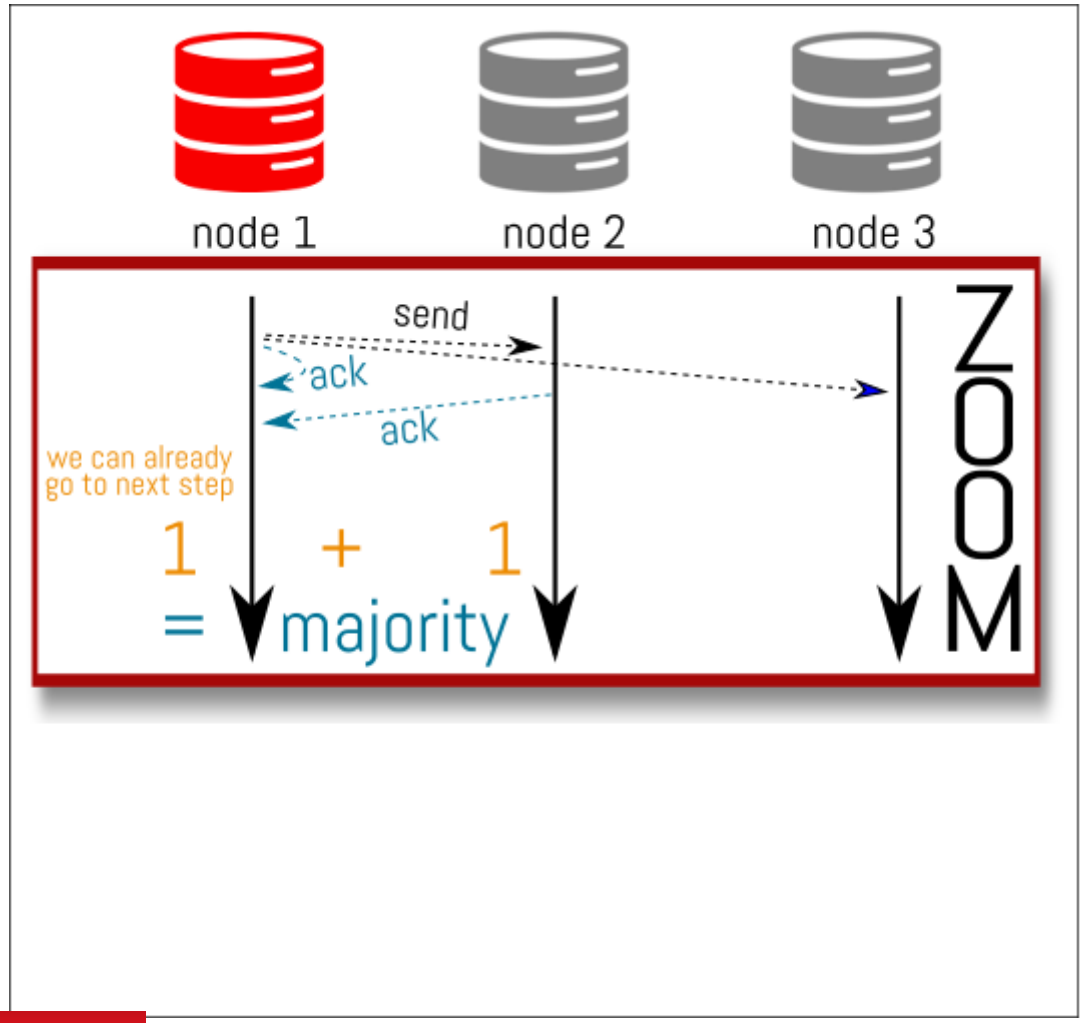
MySQL Group Replication (autocommit)



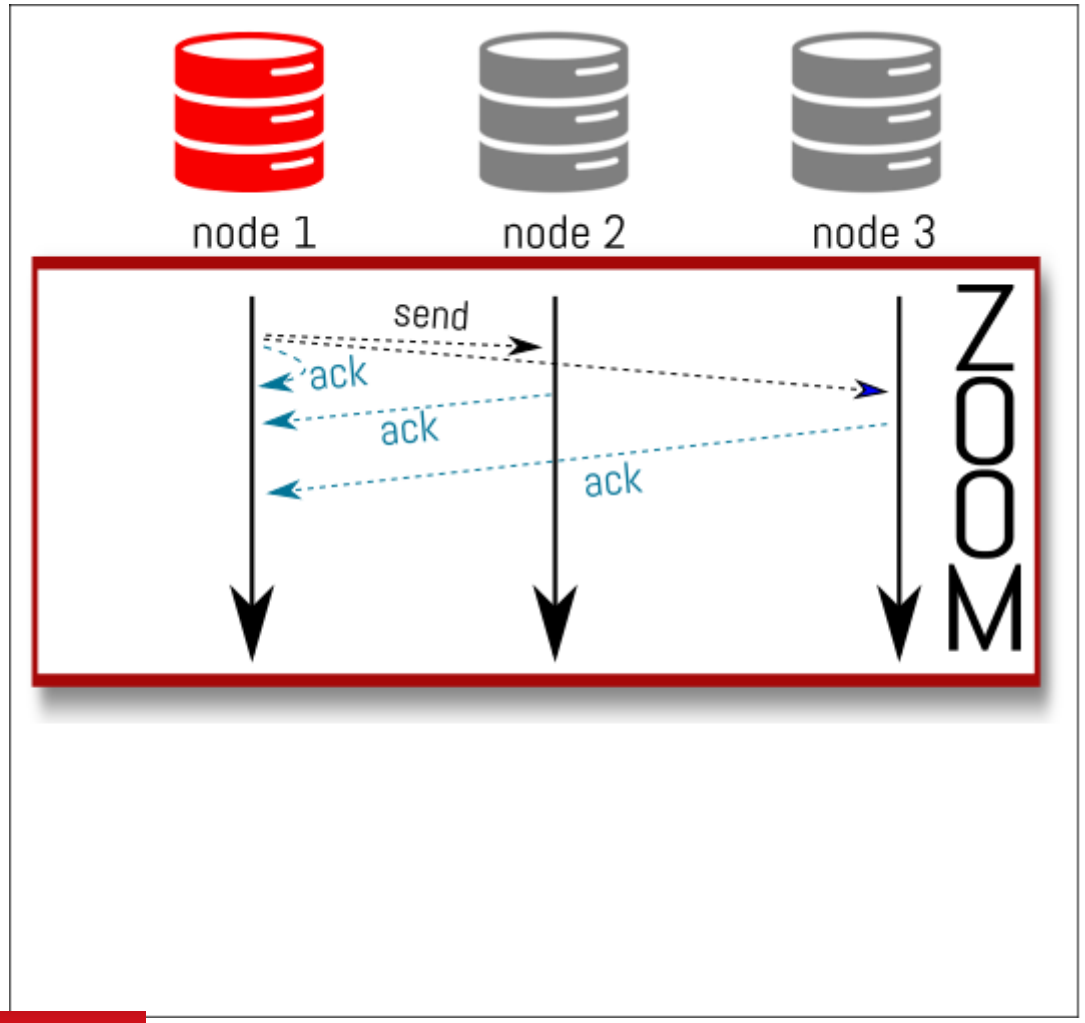
MySQL Group Replication (autocommit)



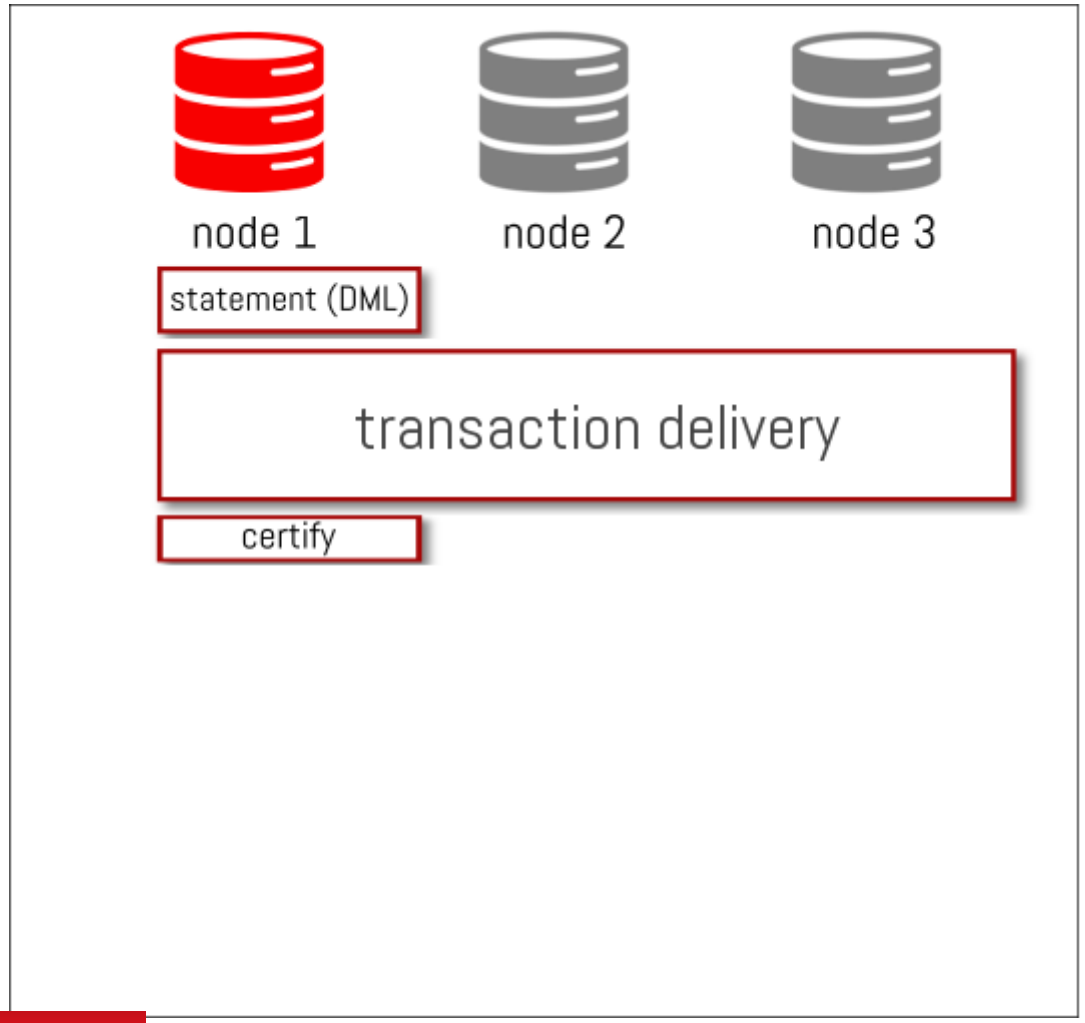
MySQL Group Replication (autocommit)



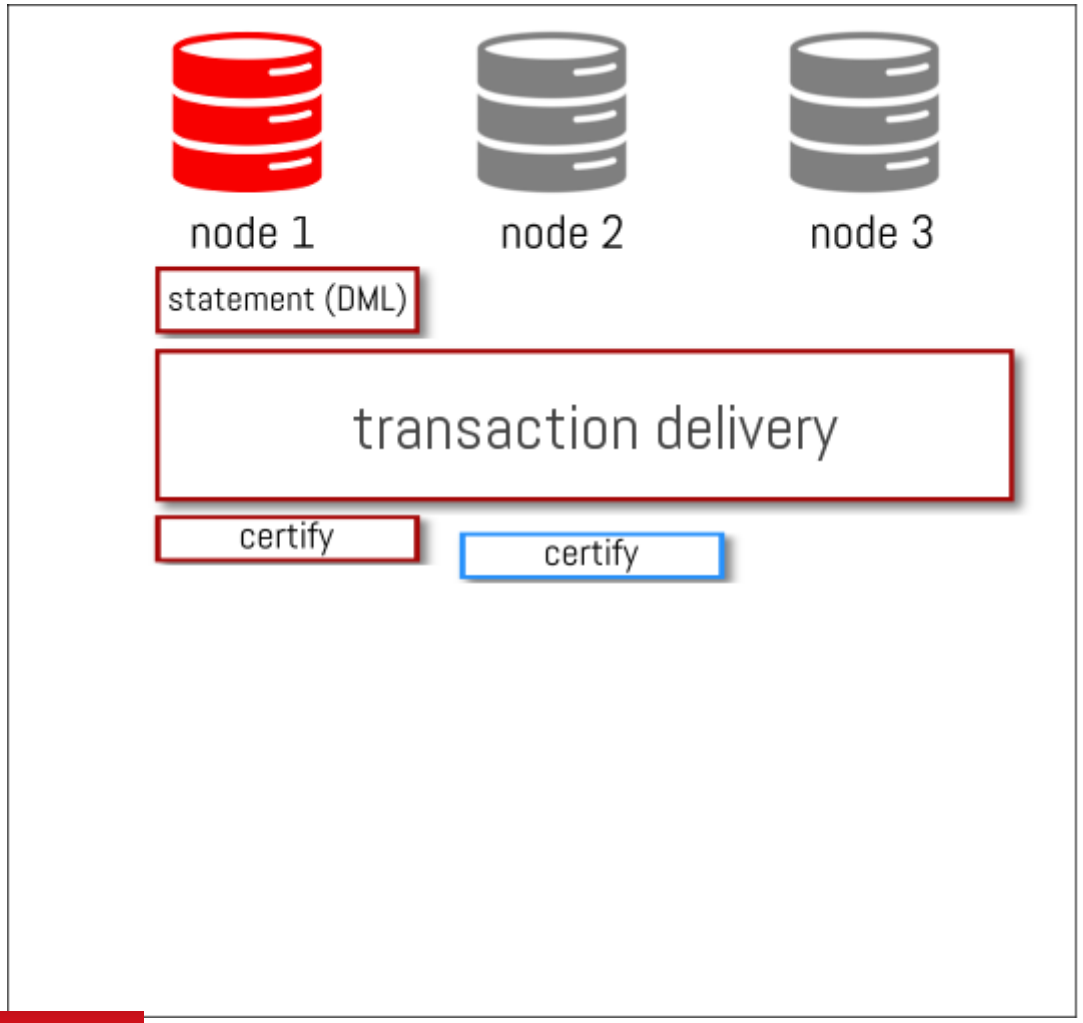
MySQL Group Replication (autocommit)



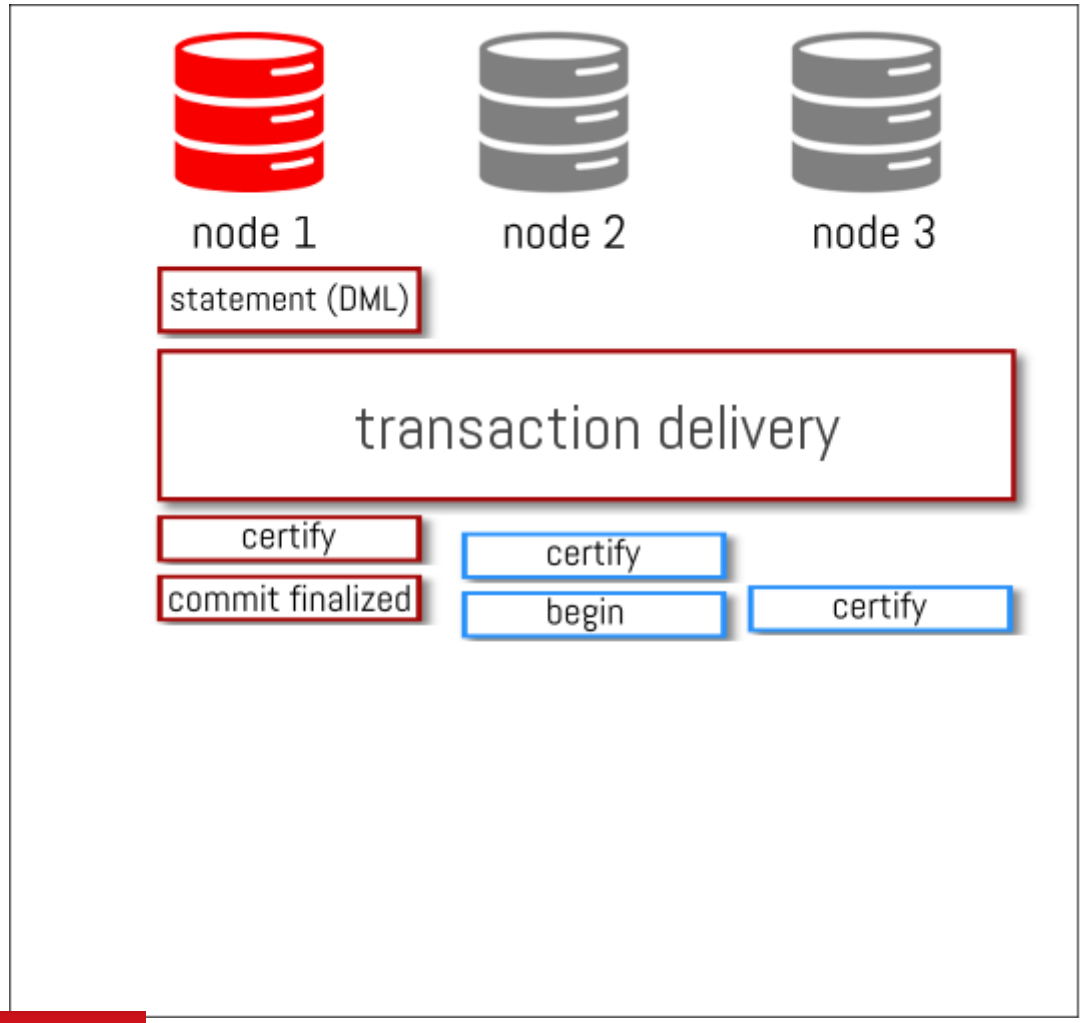
MySQL Group Replication (autocommit)



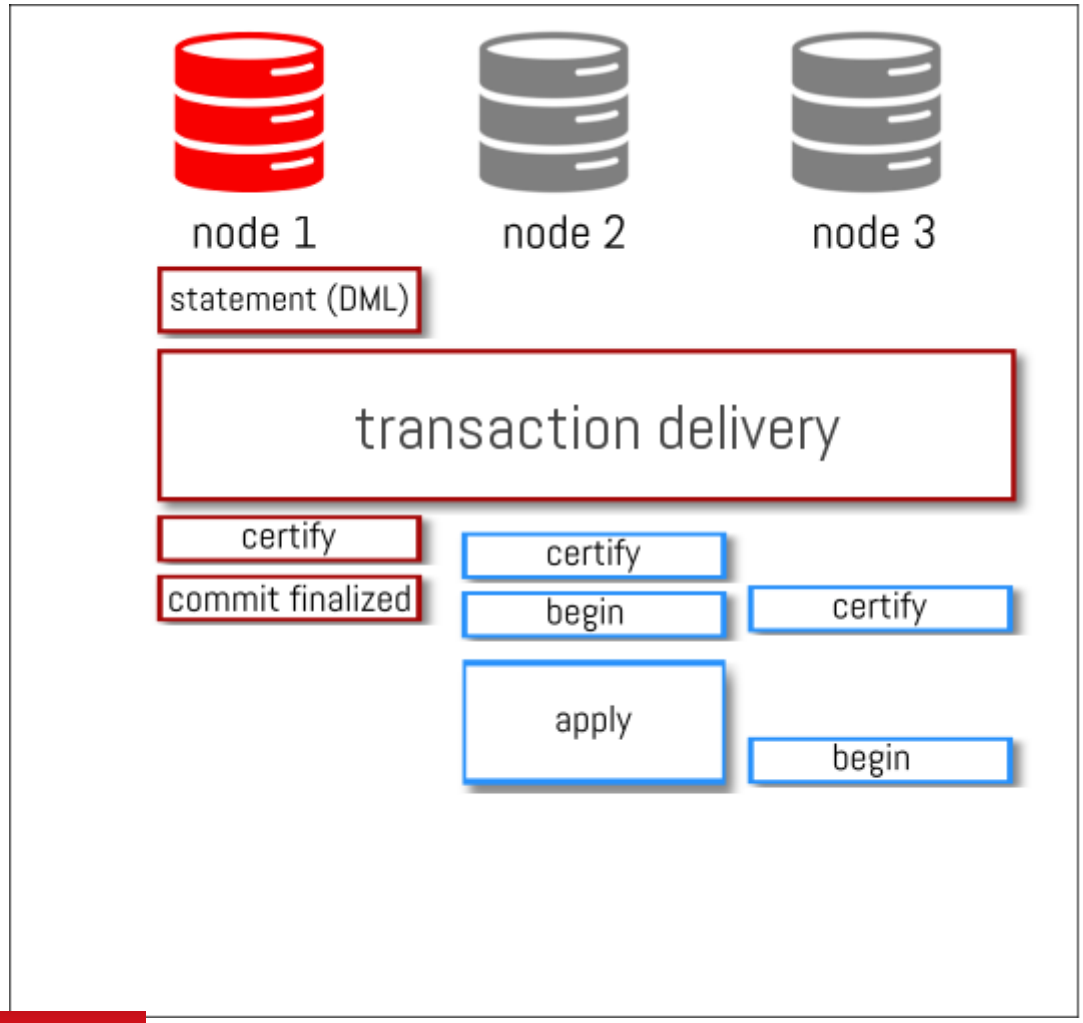
MySQL Group Replication (autocommit)



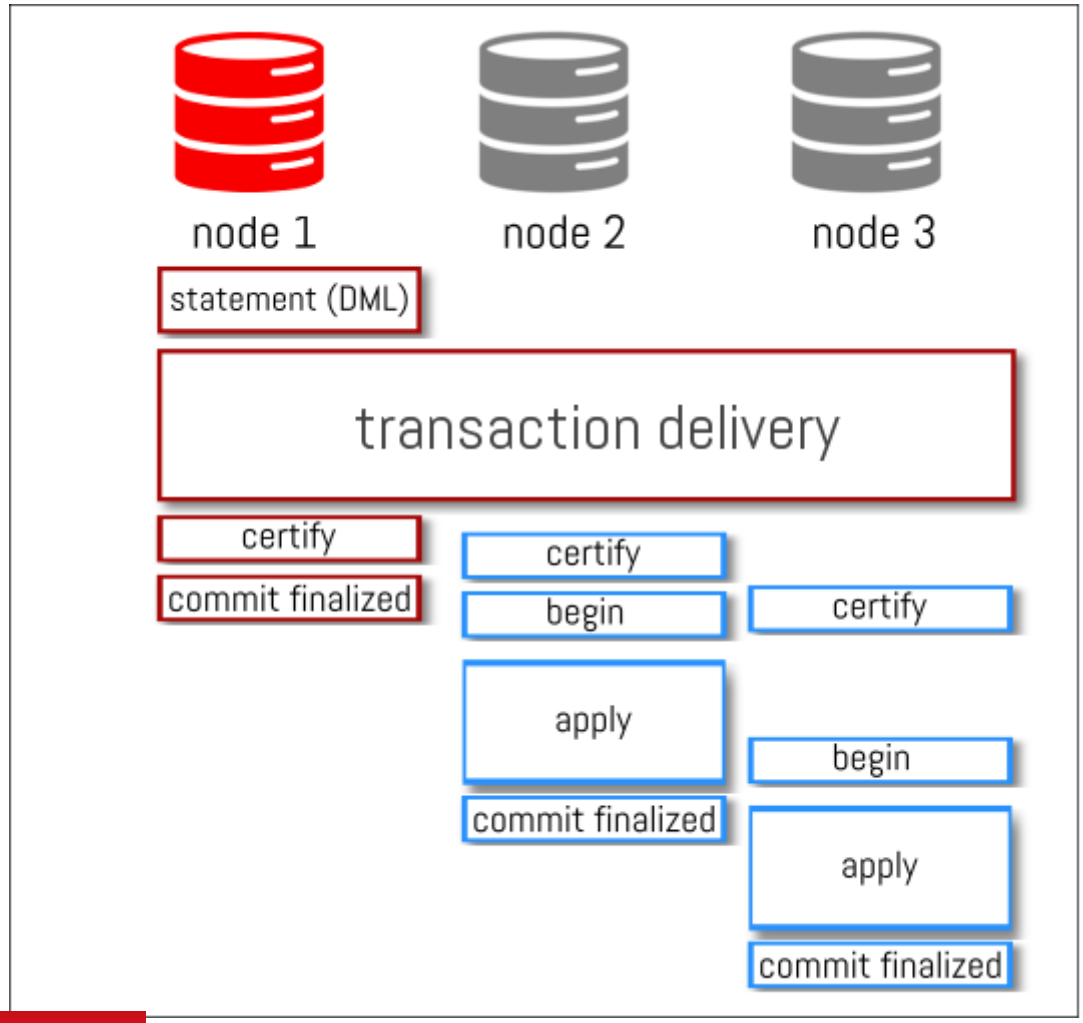
MySQL Group Replication (autocommit)



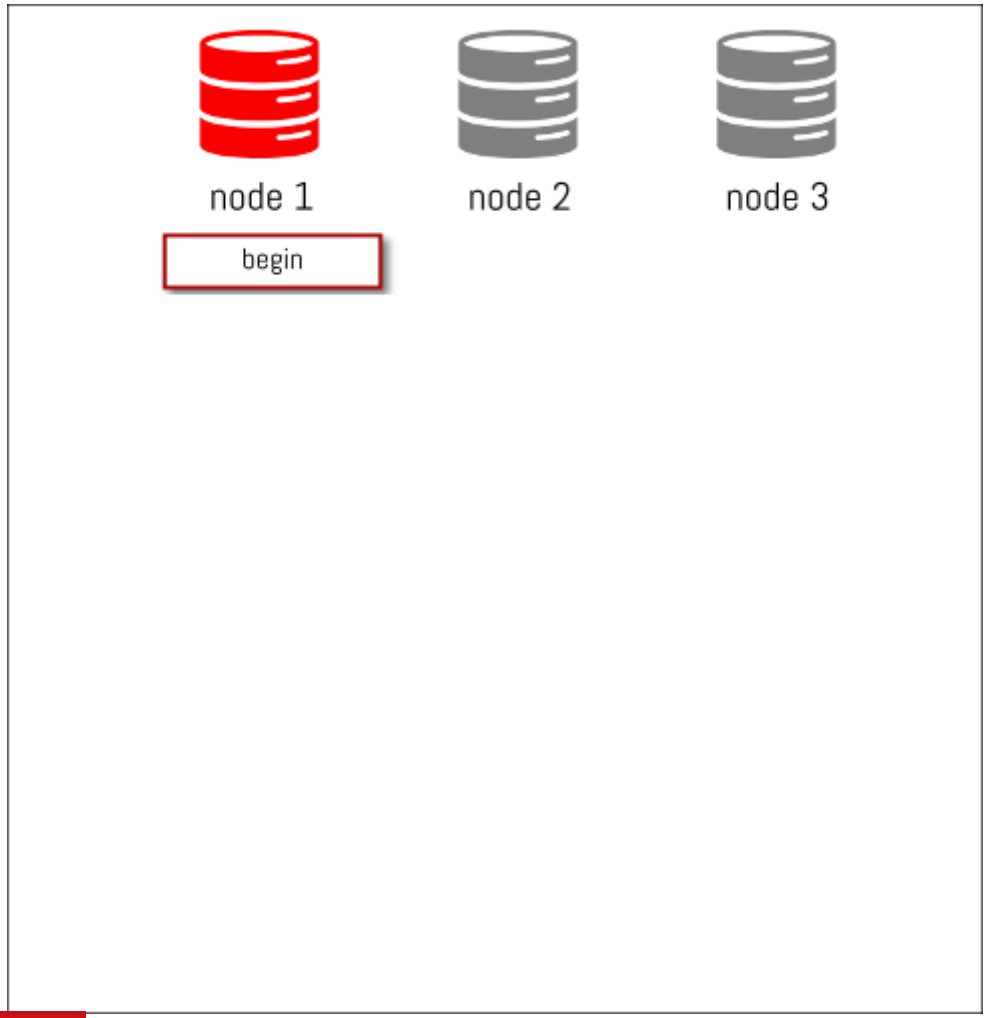
MySQL Group Replication (autocommit)



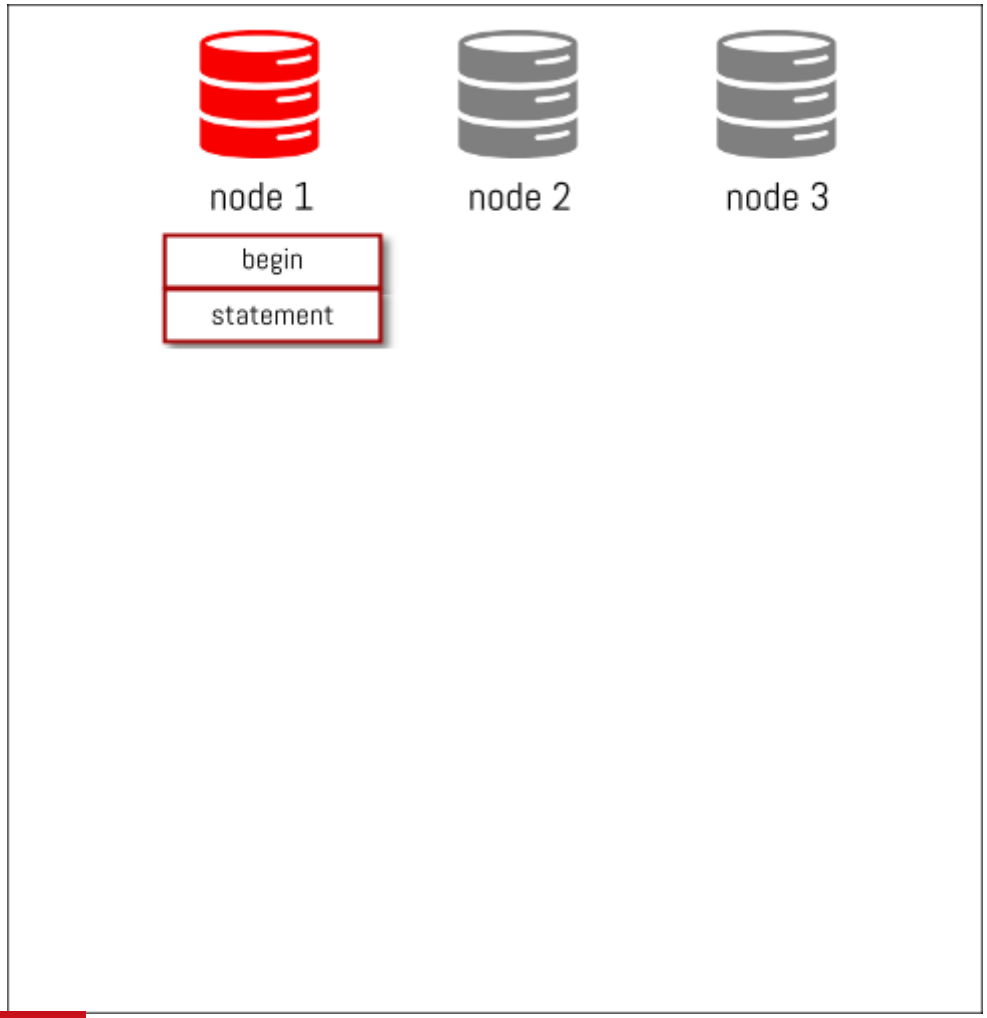
MySQL Group Replication (autocommit)



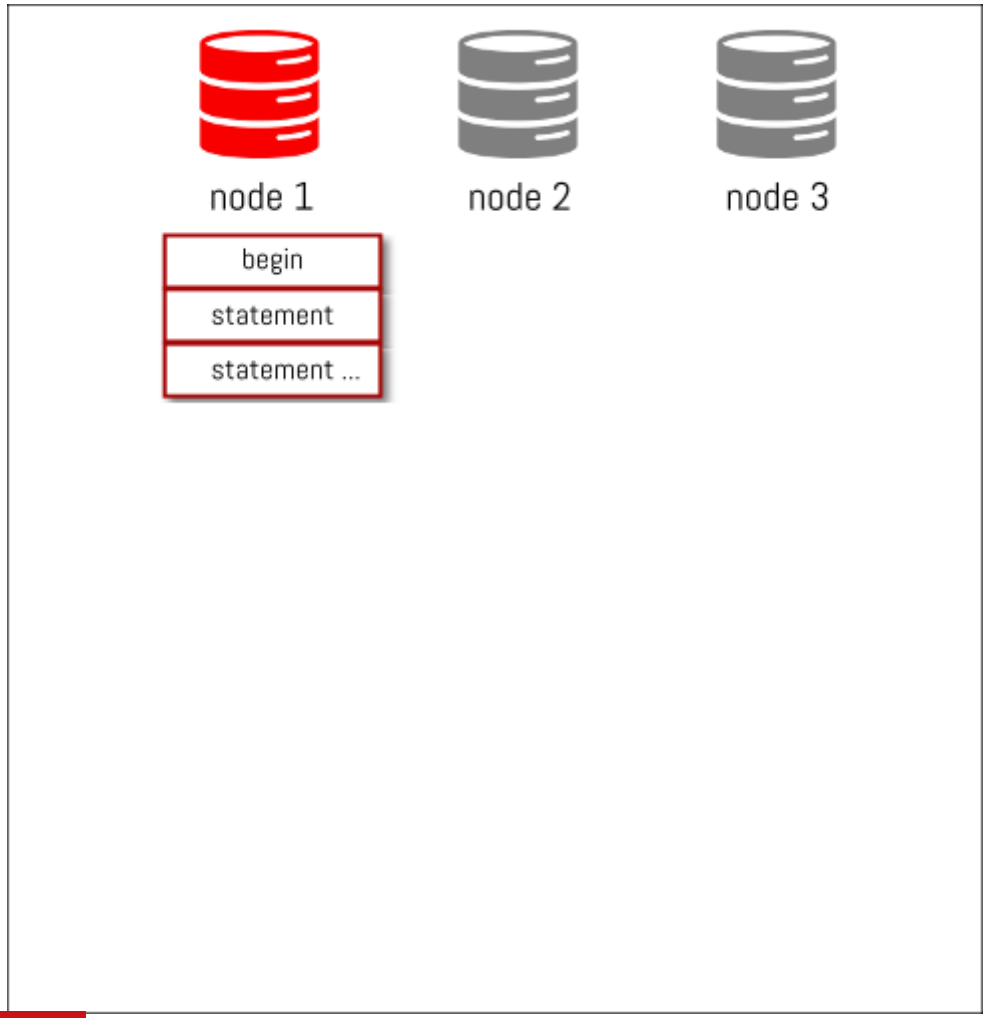
MySQL Group Replication (full transaction)



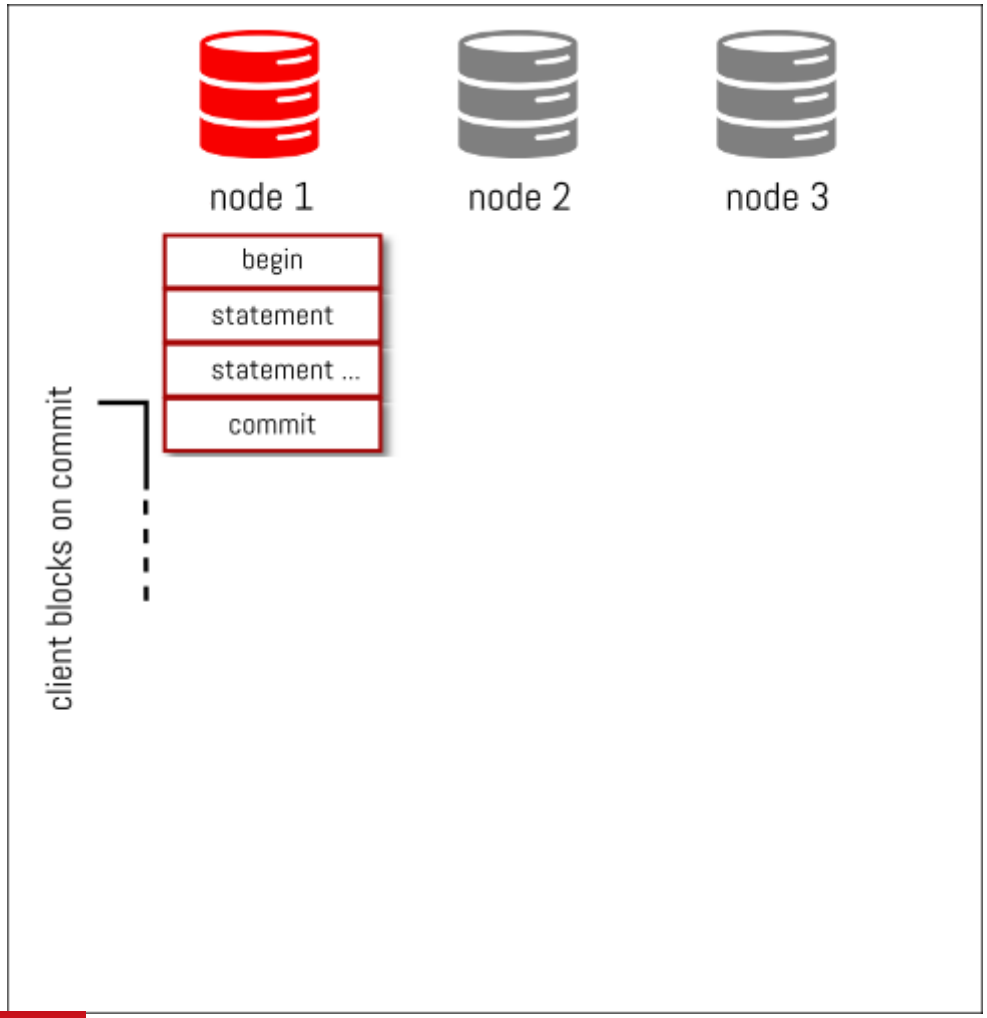
MySQL Group Replication (full transaction)



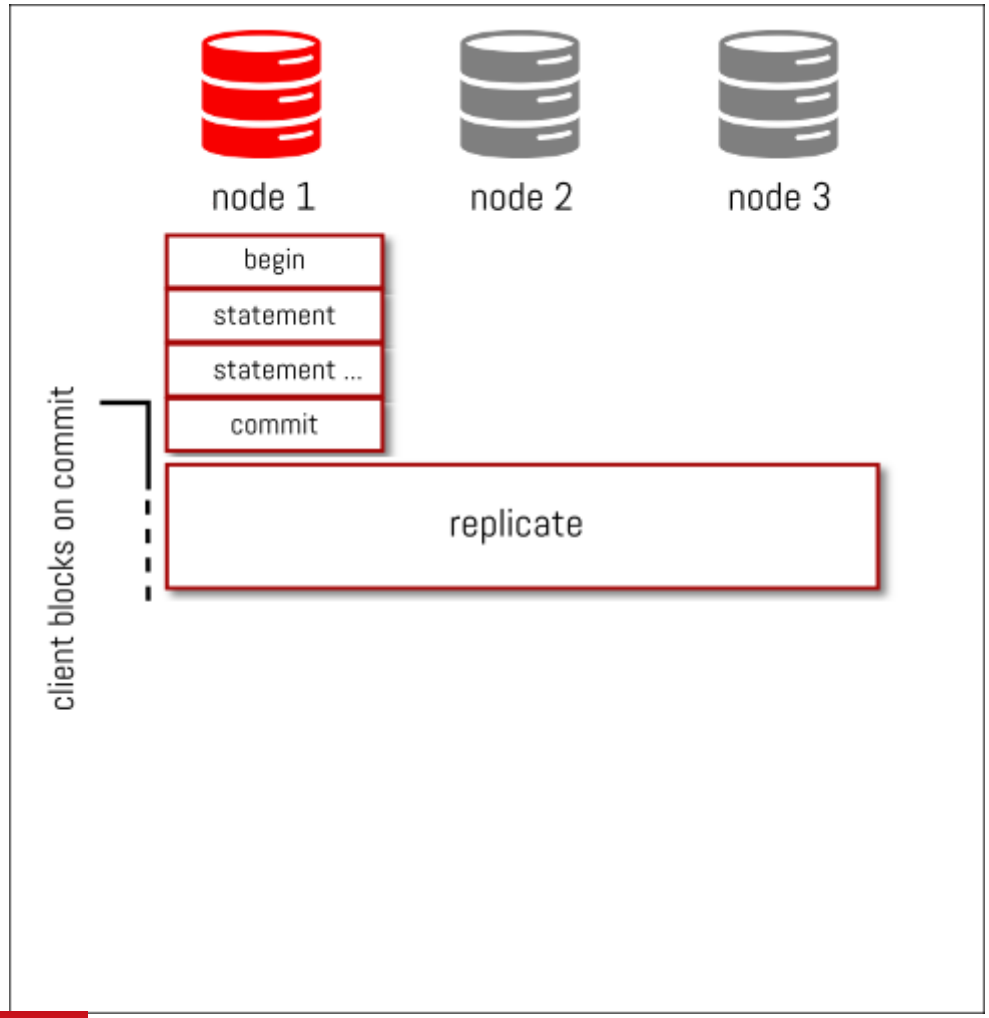
MySQL Group Replication (full transaction)



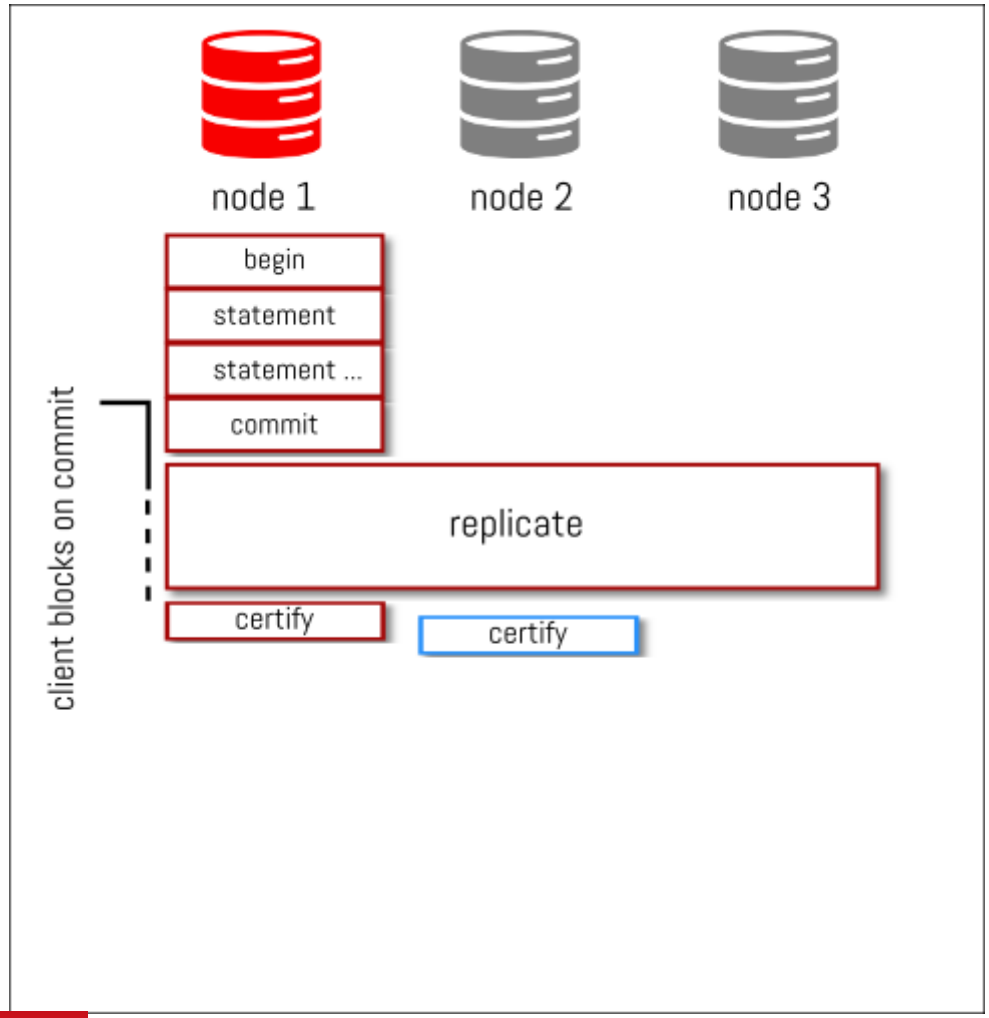
MySQL Group Replication (full transaction)



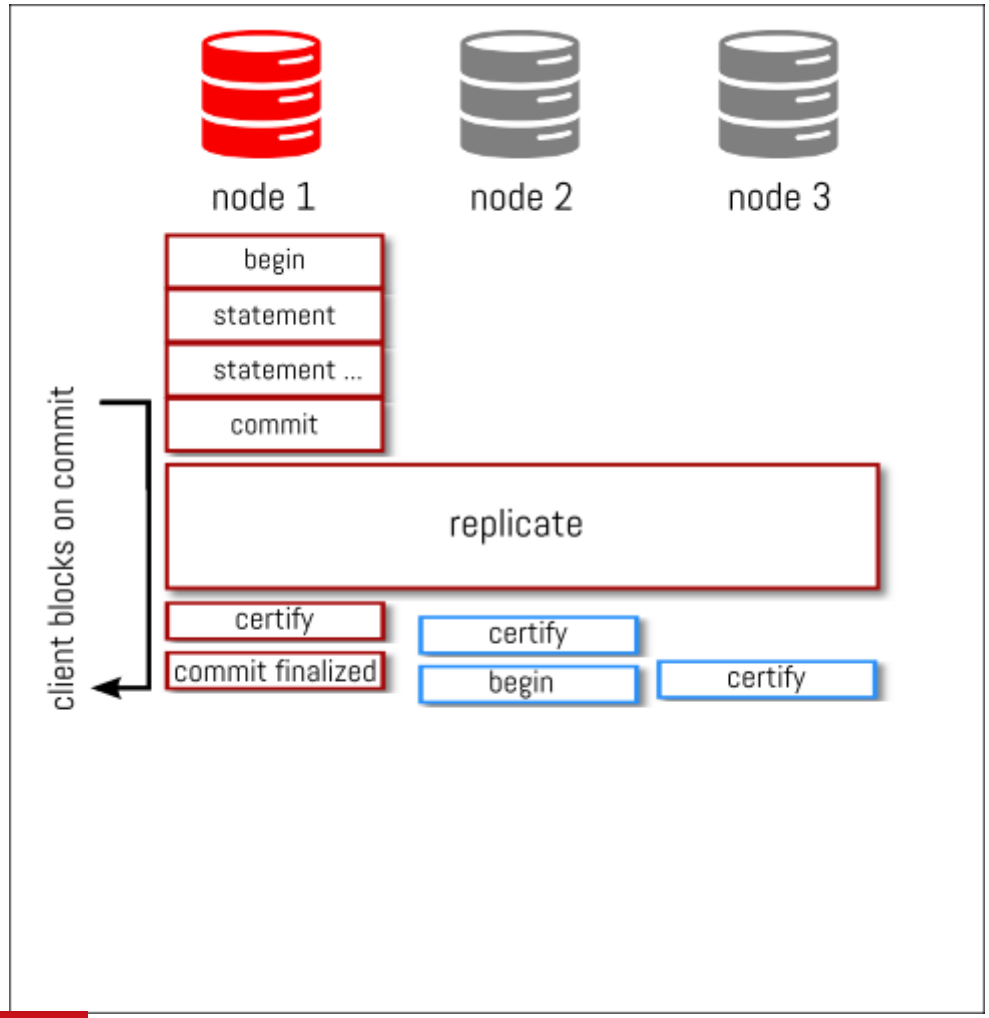
MySQL Group Replication (full transaction)



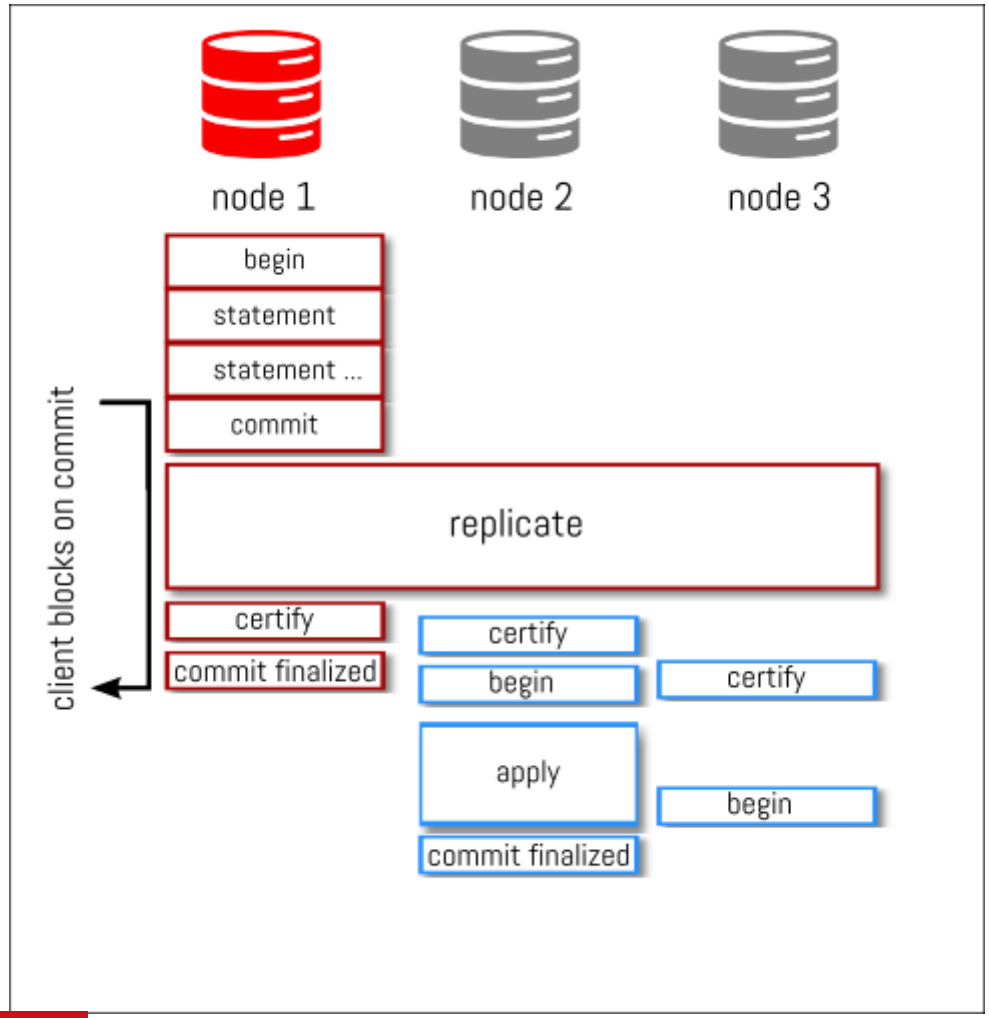
MySQL Group Replication (full transaction)



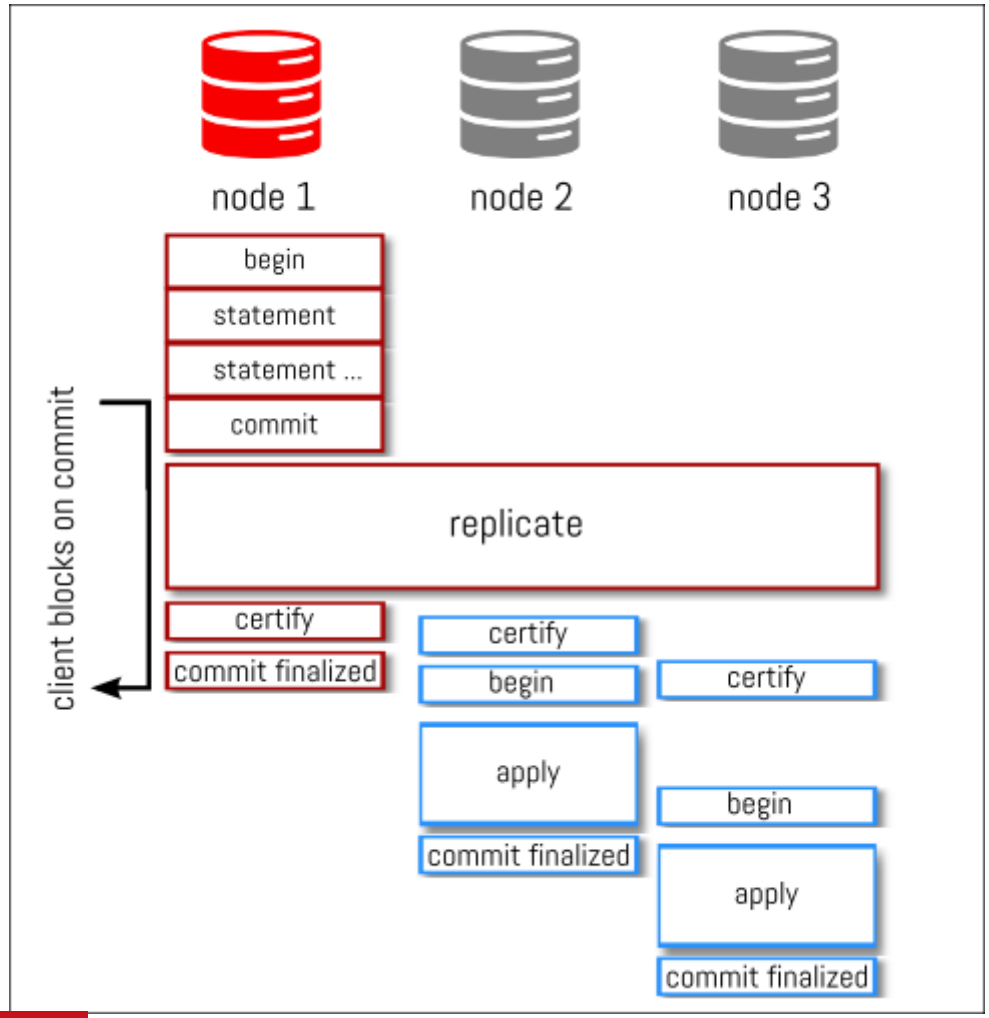
MySQL Group Replication (full transaction)



MySQL Group Replication (full transaction)

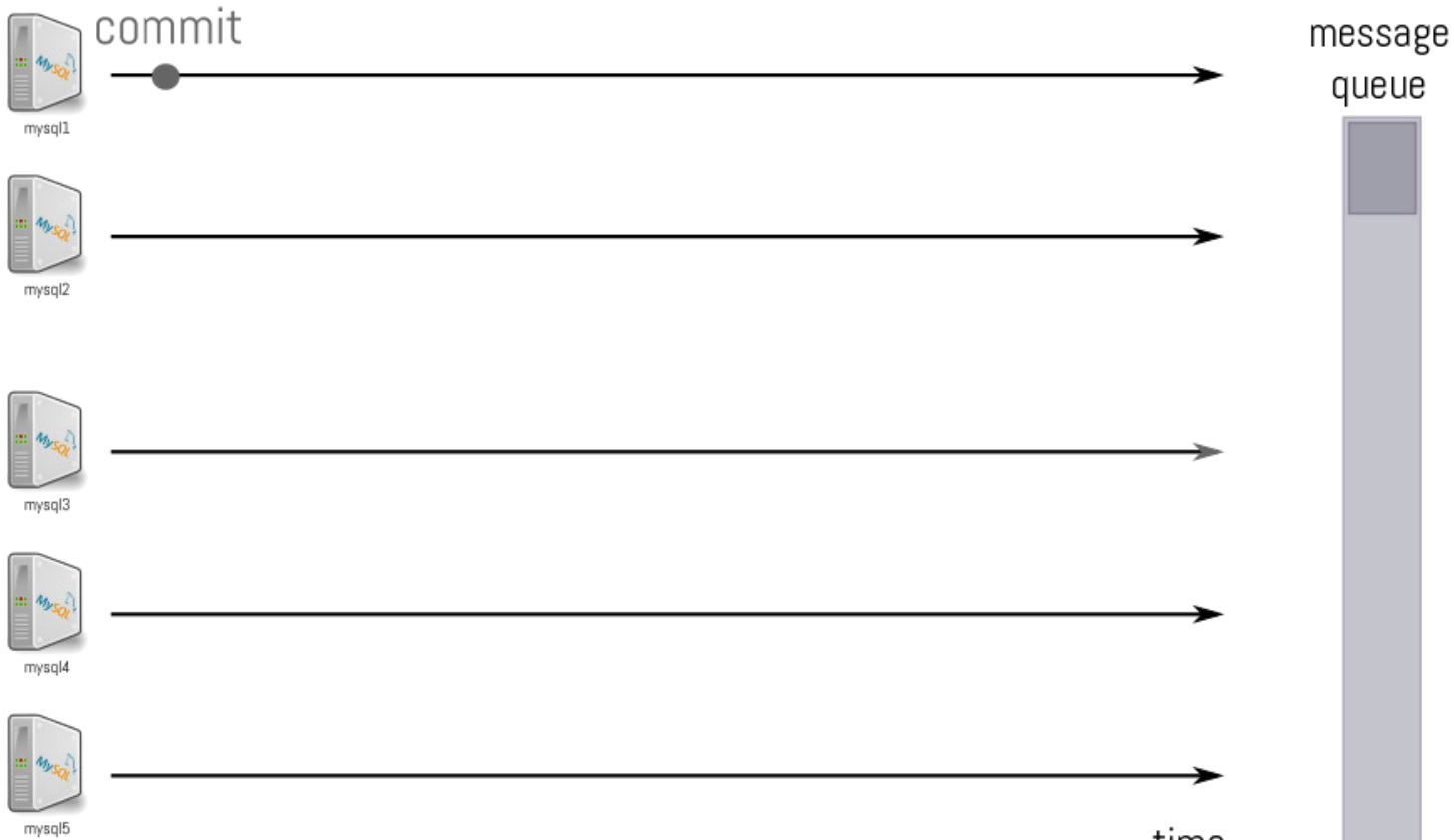


MySQL Group Replication (full transaction)

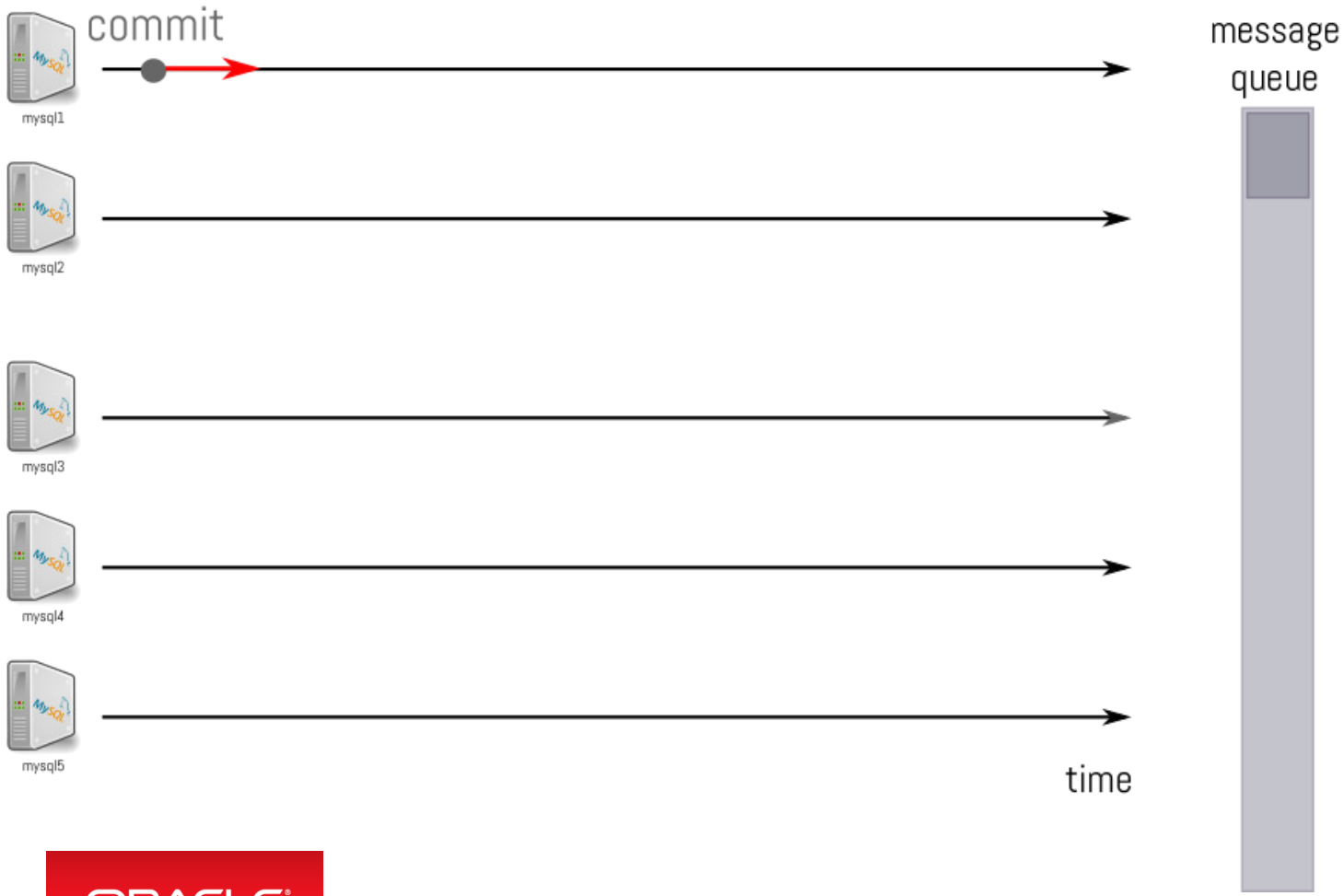


Group Replication : Total Order Delivery - GTID

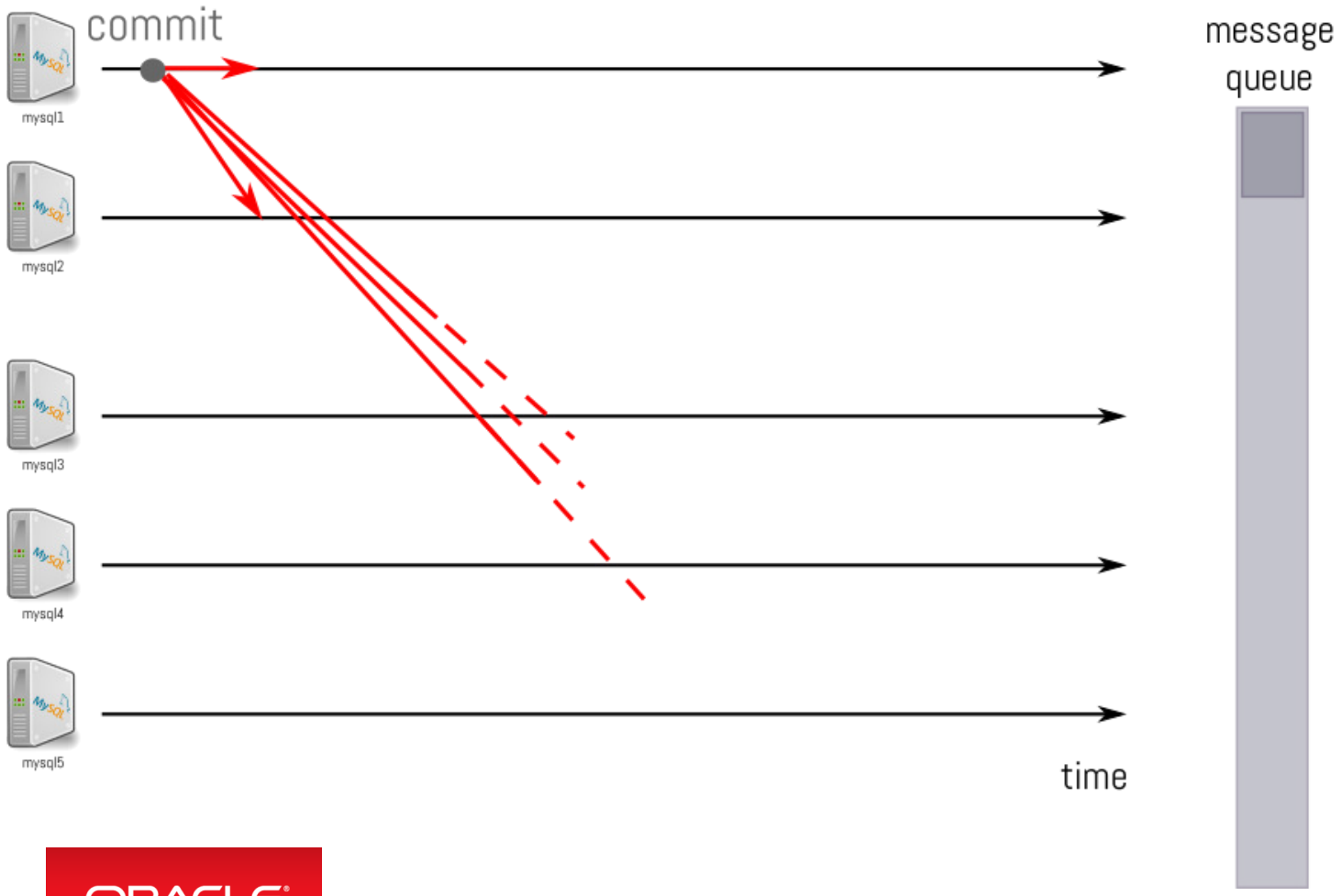
Group Replication : Total Order Delivery - GTID



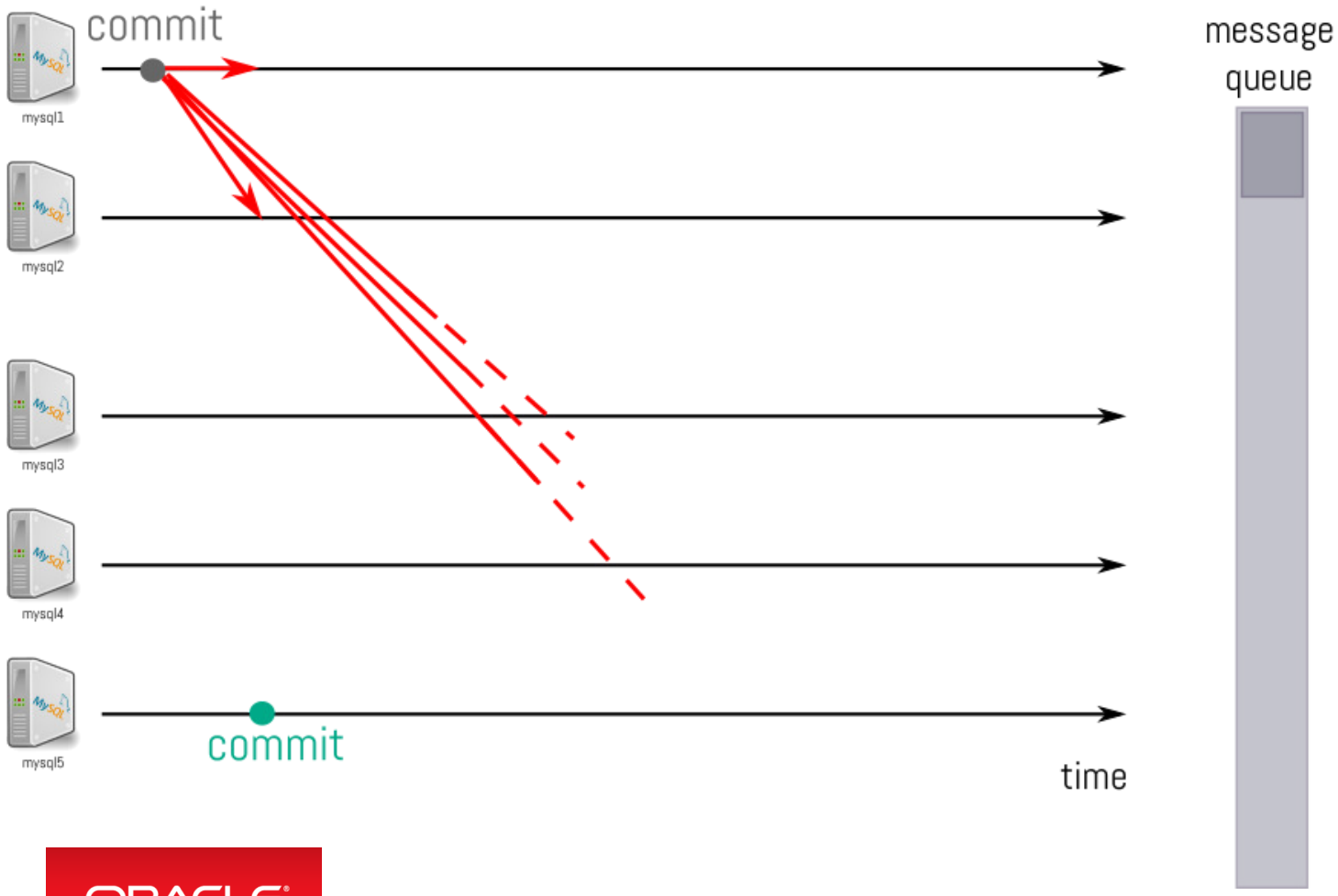
Group Replication : Total Order Delivery - GTID



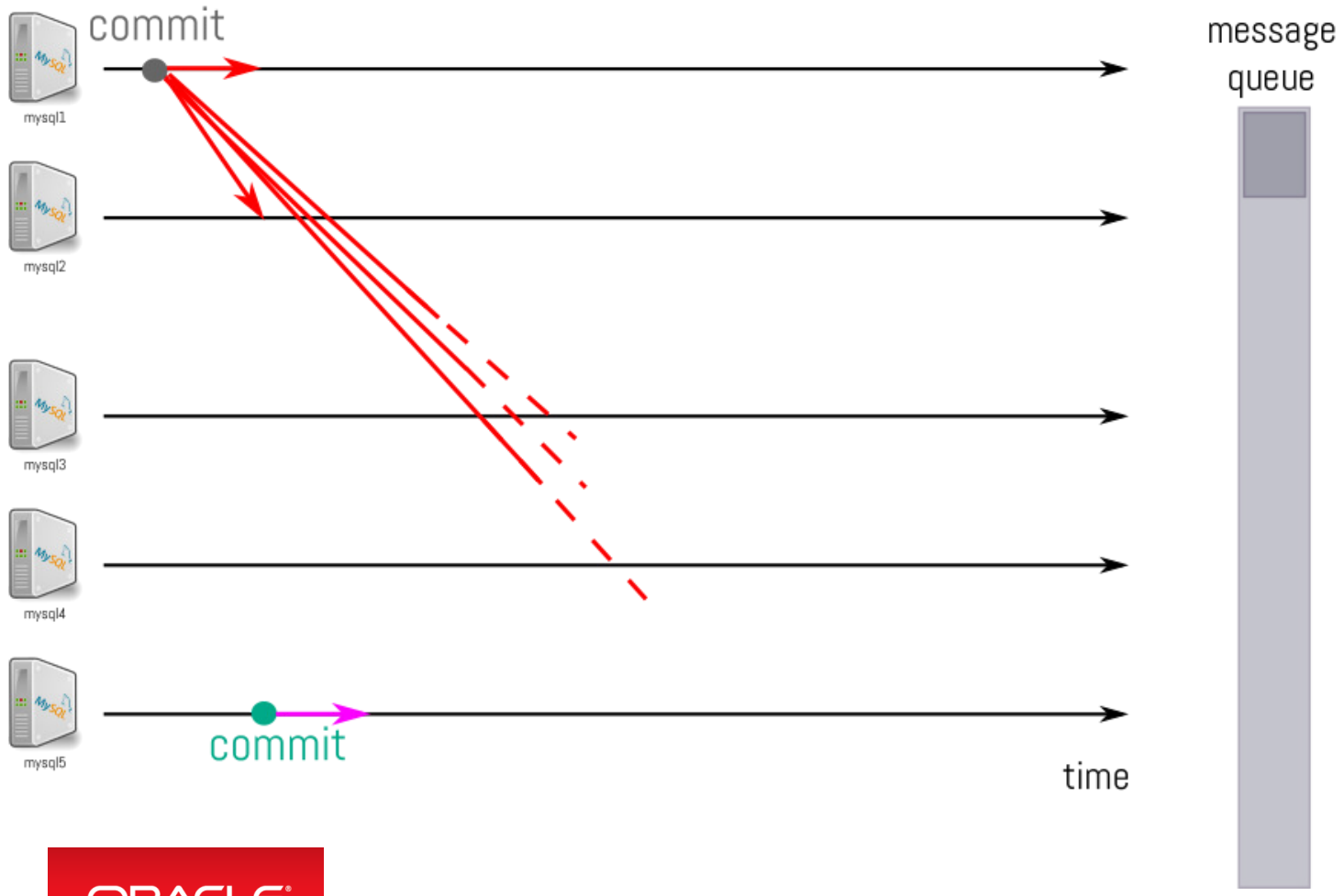
Group Replication : Total Order Delivery - GTID



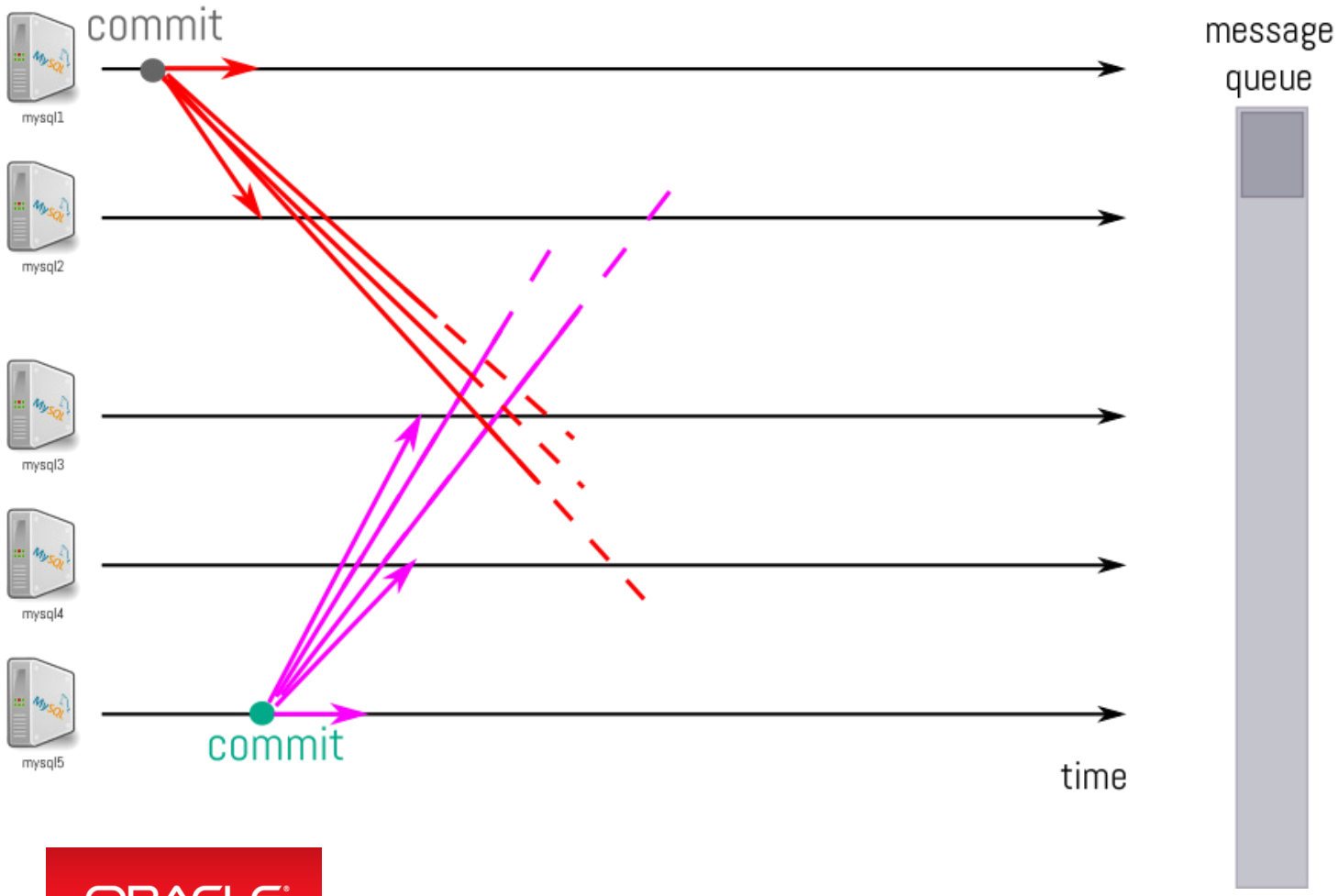
Group Replication : Total Order Delivery - GTID



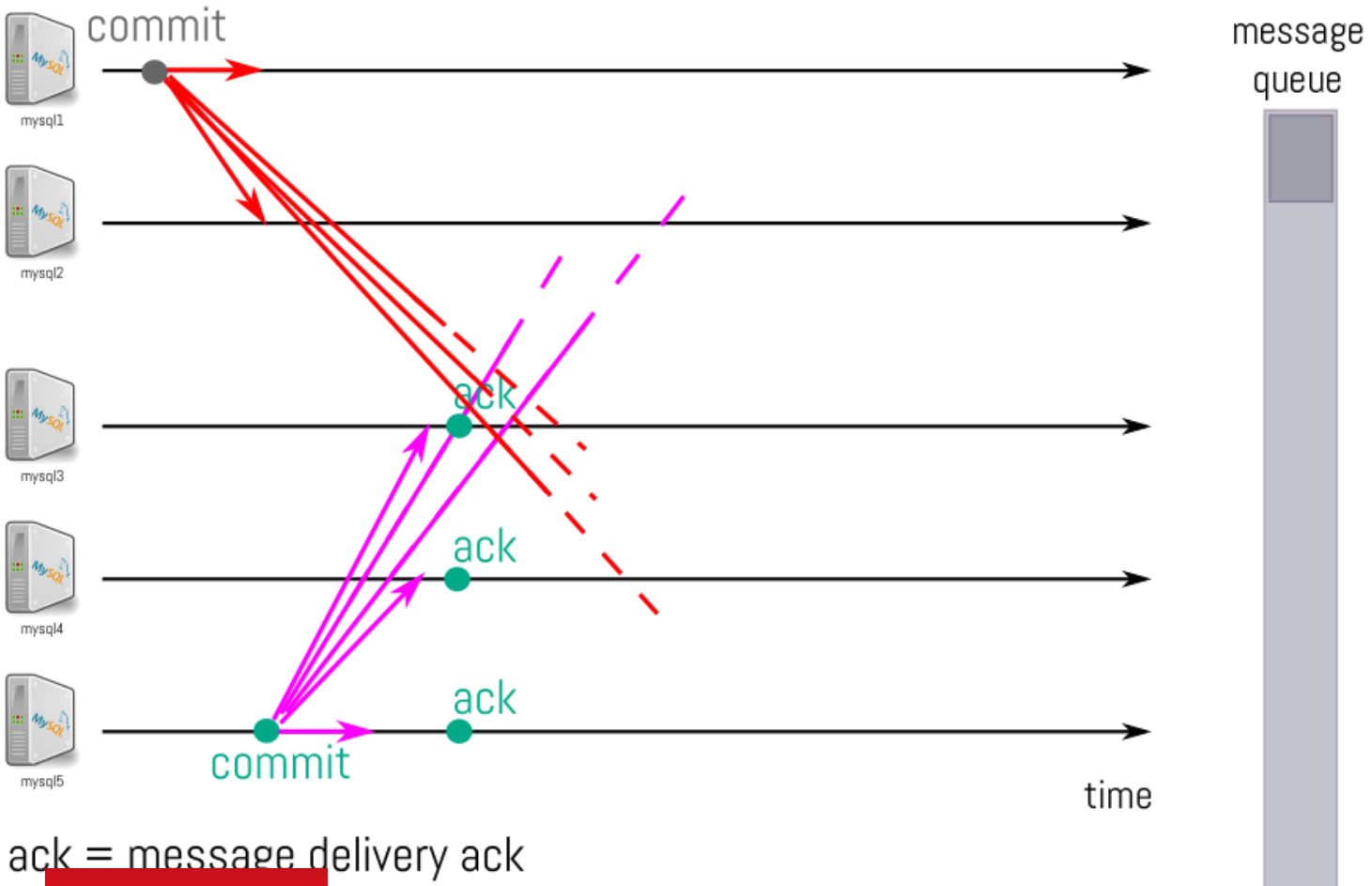
Group Replication : Total Order Delivery - GTID



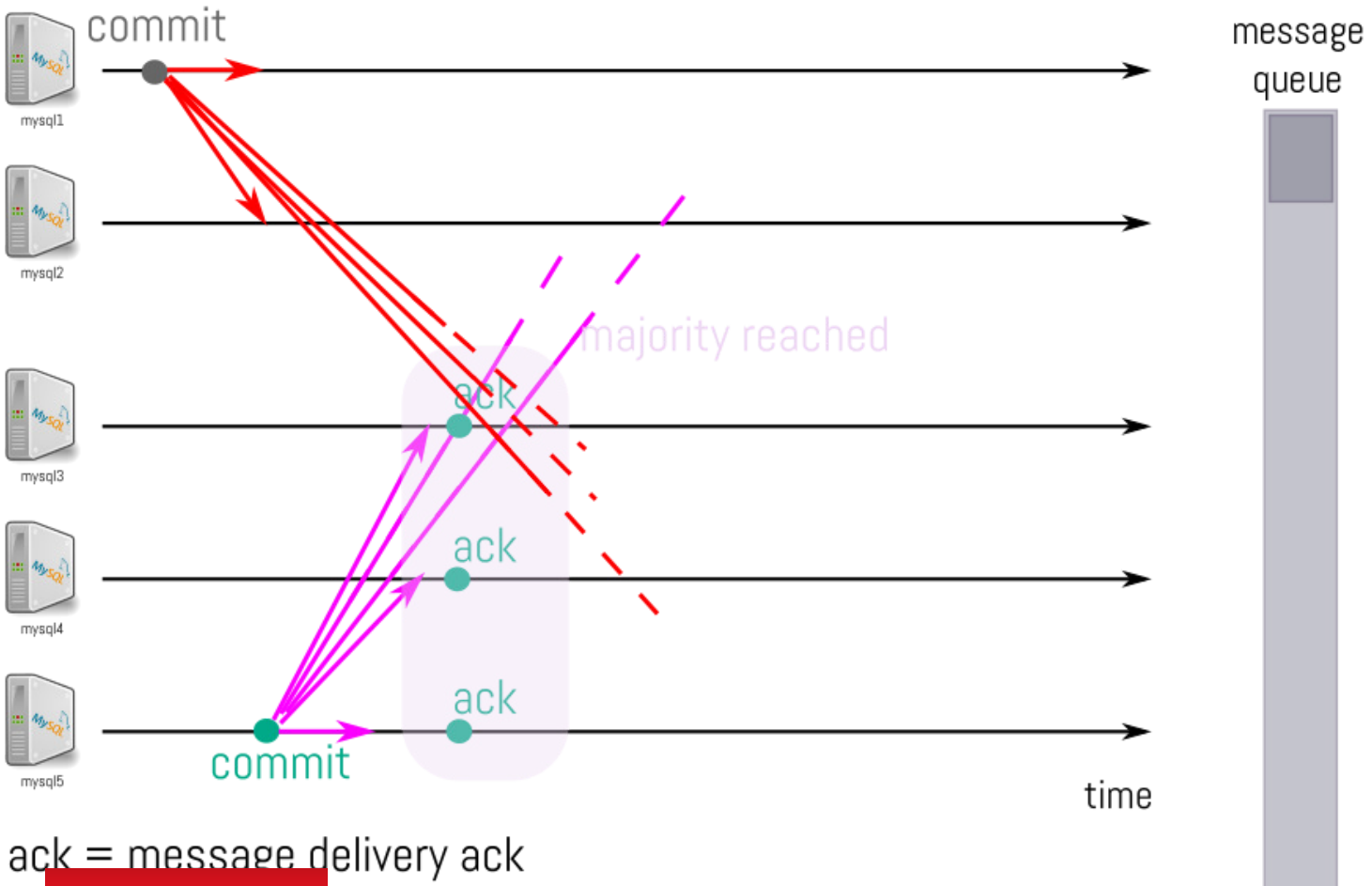
Group Replication : Total Order Delivery - GTID



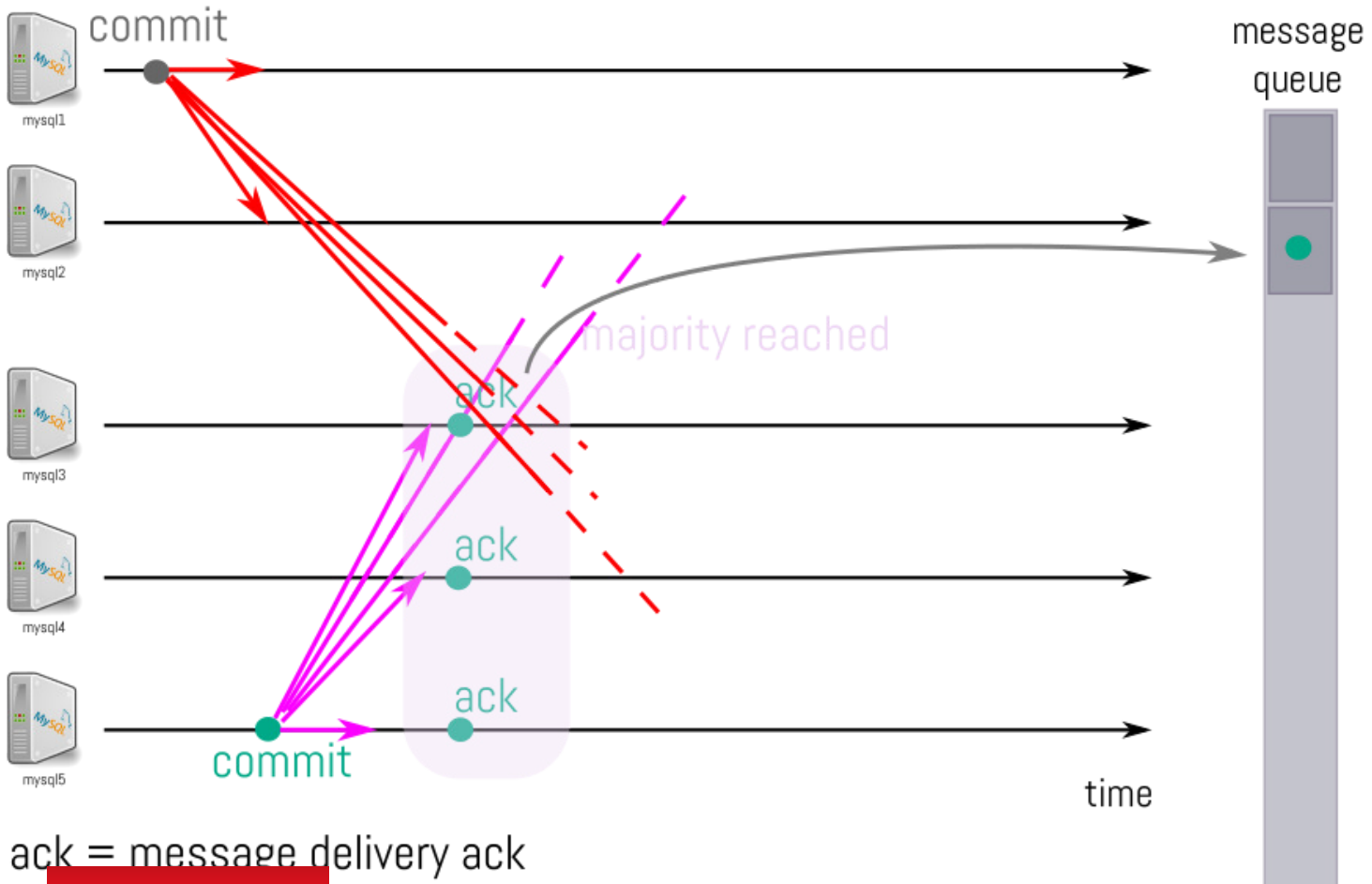
Group Replication : Total Order Delivery - GTID



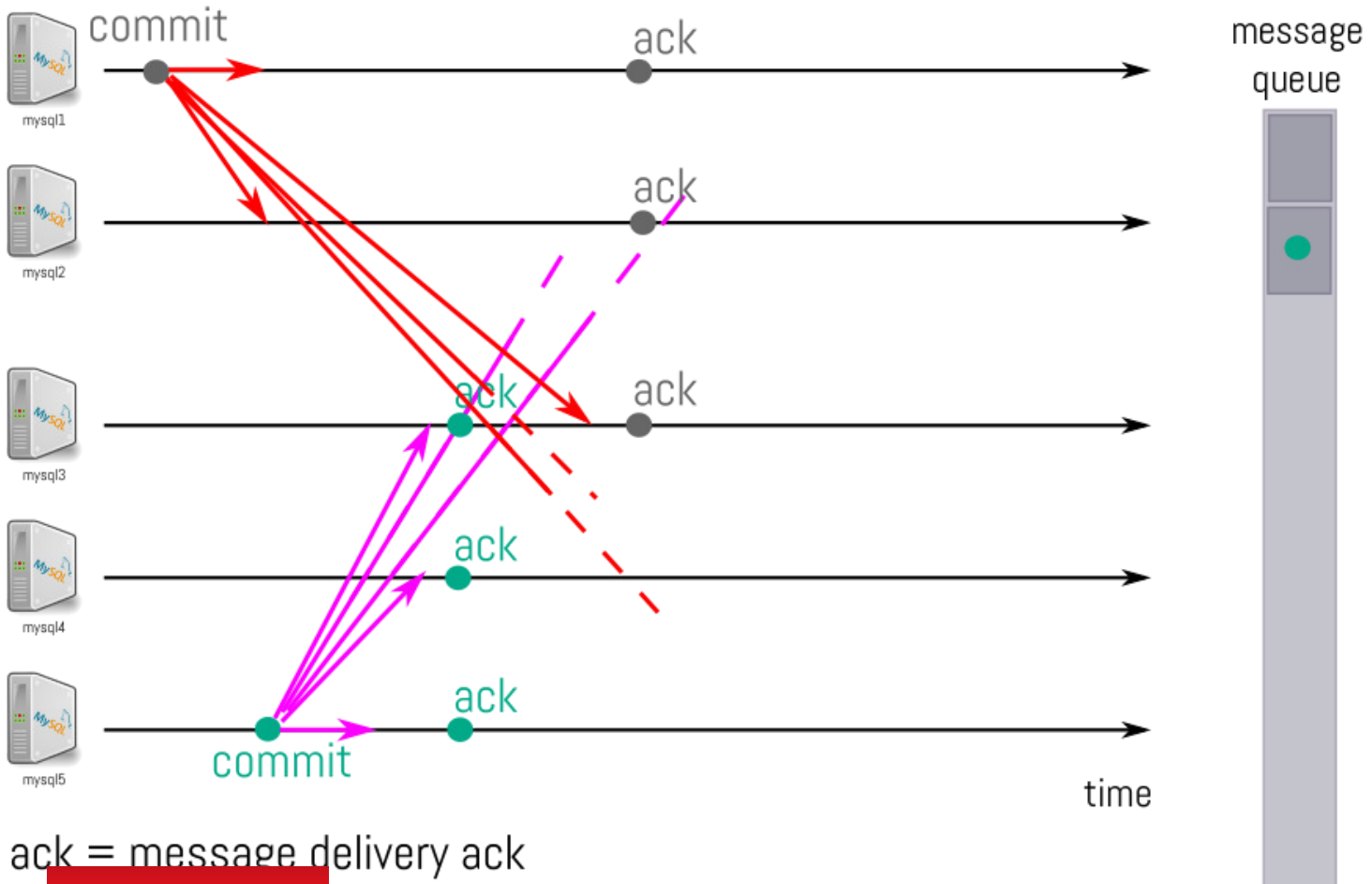
Group Replication : Total Order Delivery - GTID



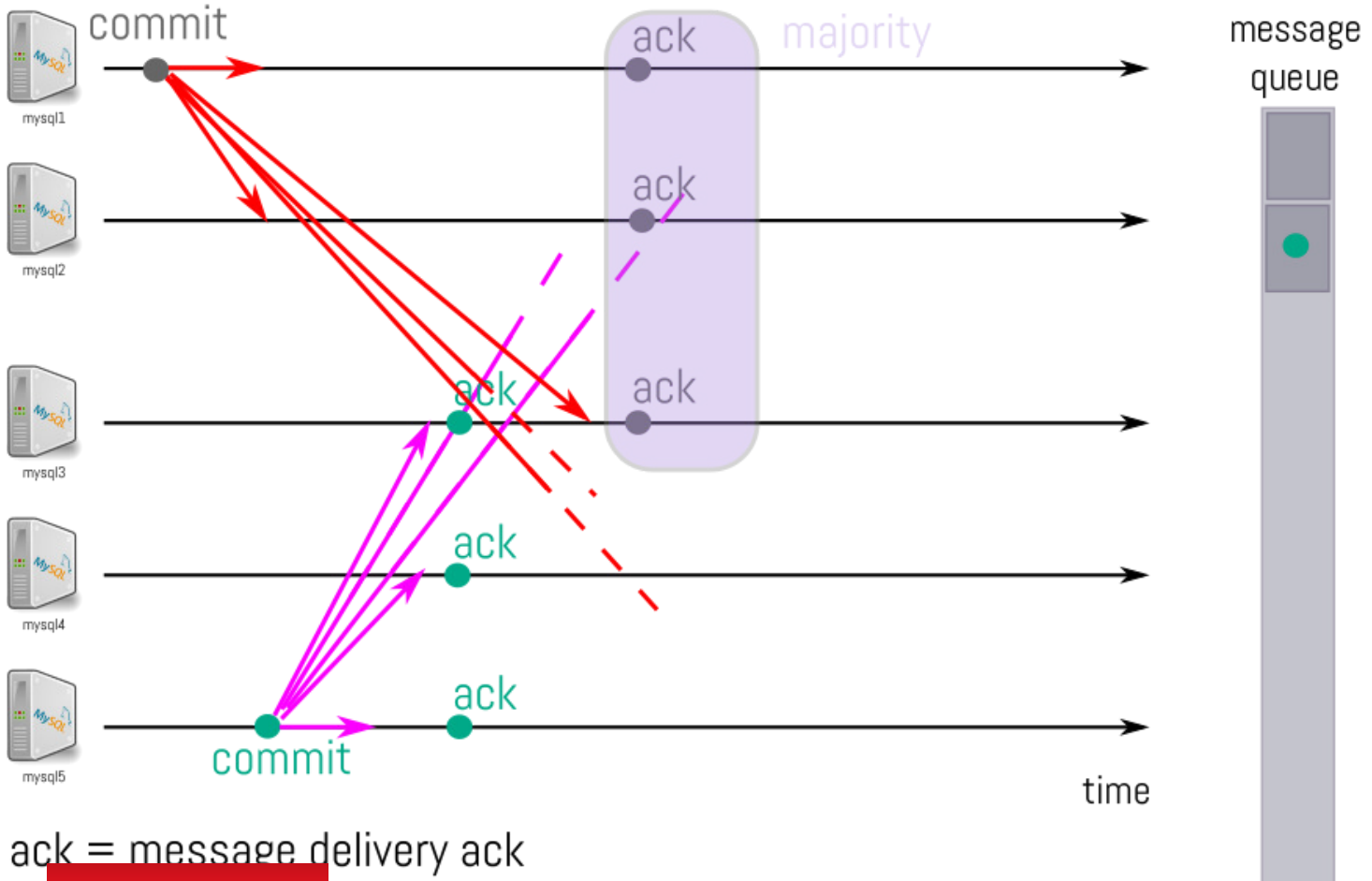
Group Replication : Total Order Delivery - GTID



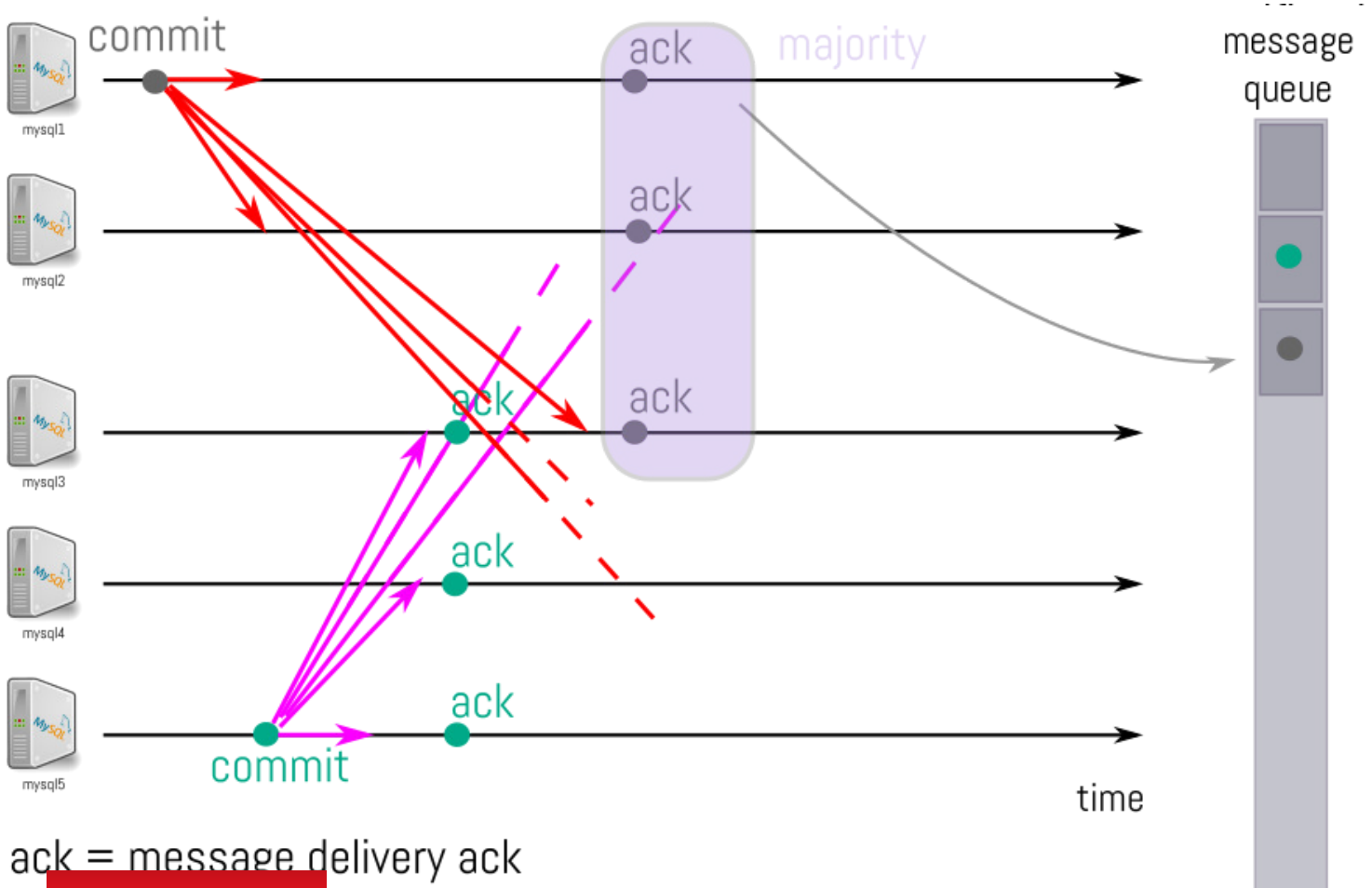
Group Replication : Total Order Delivery - GTID



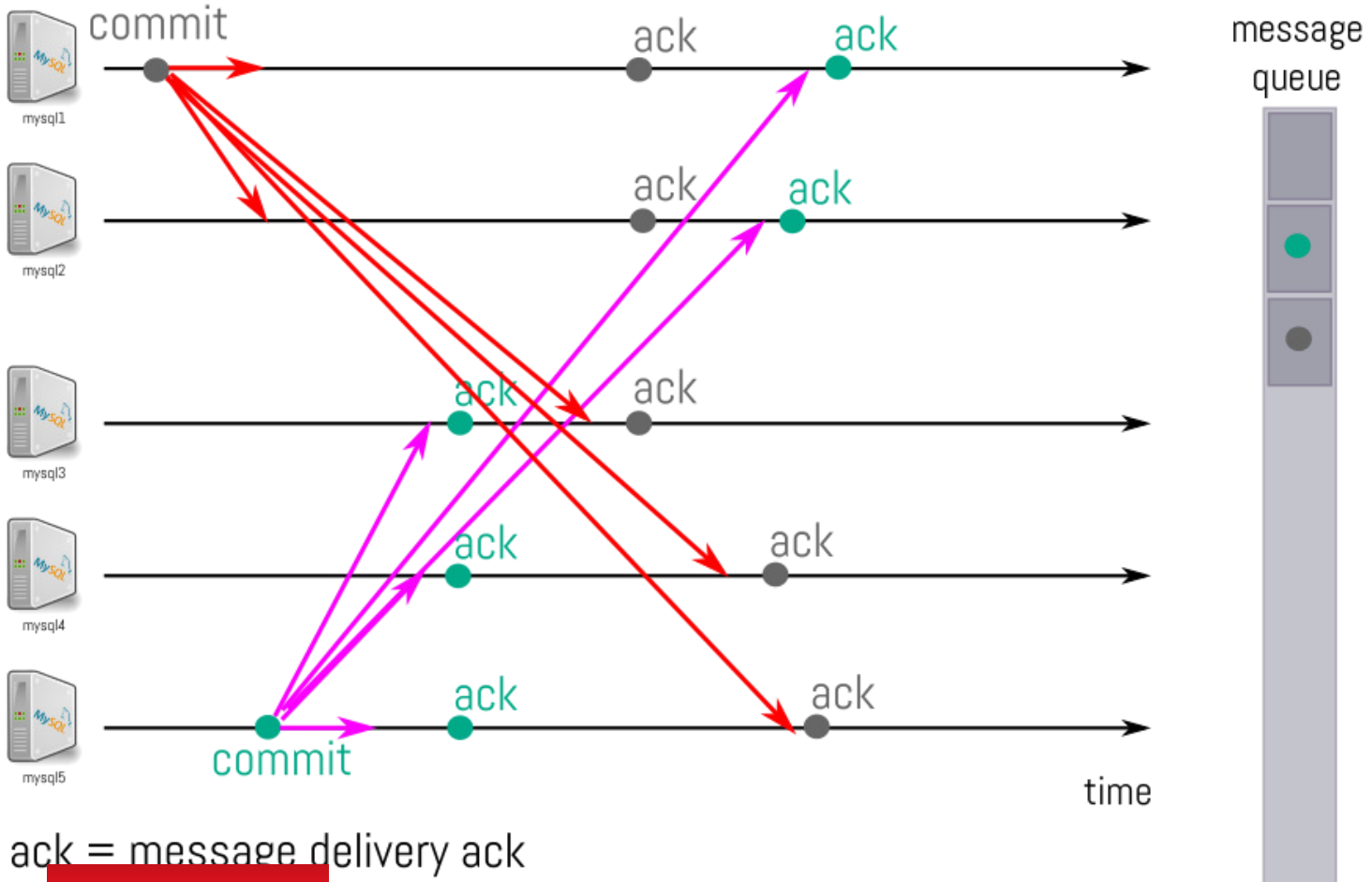
Group Replication : Total Order Delivery - GTID



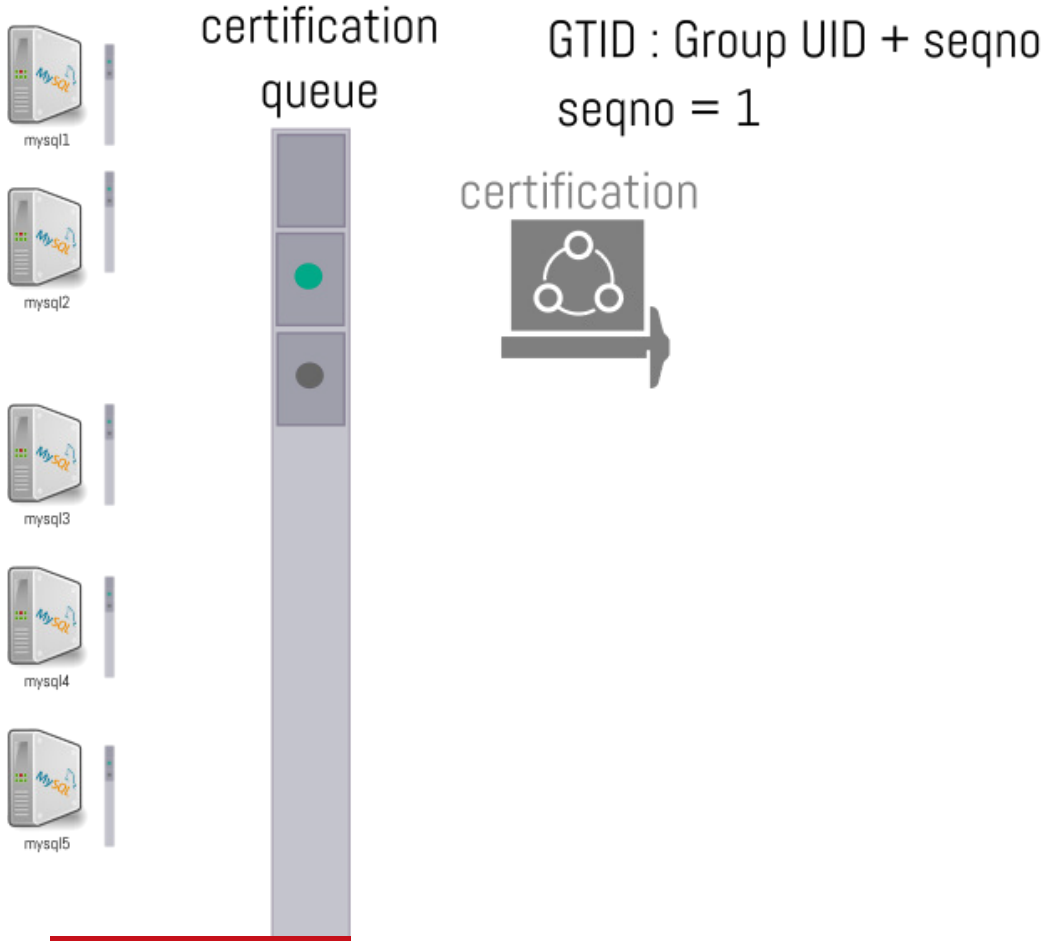
Group Replication : Total Order Delivery - GTID



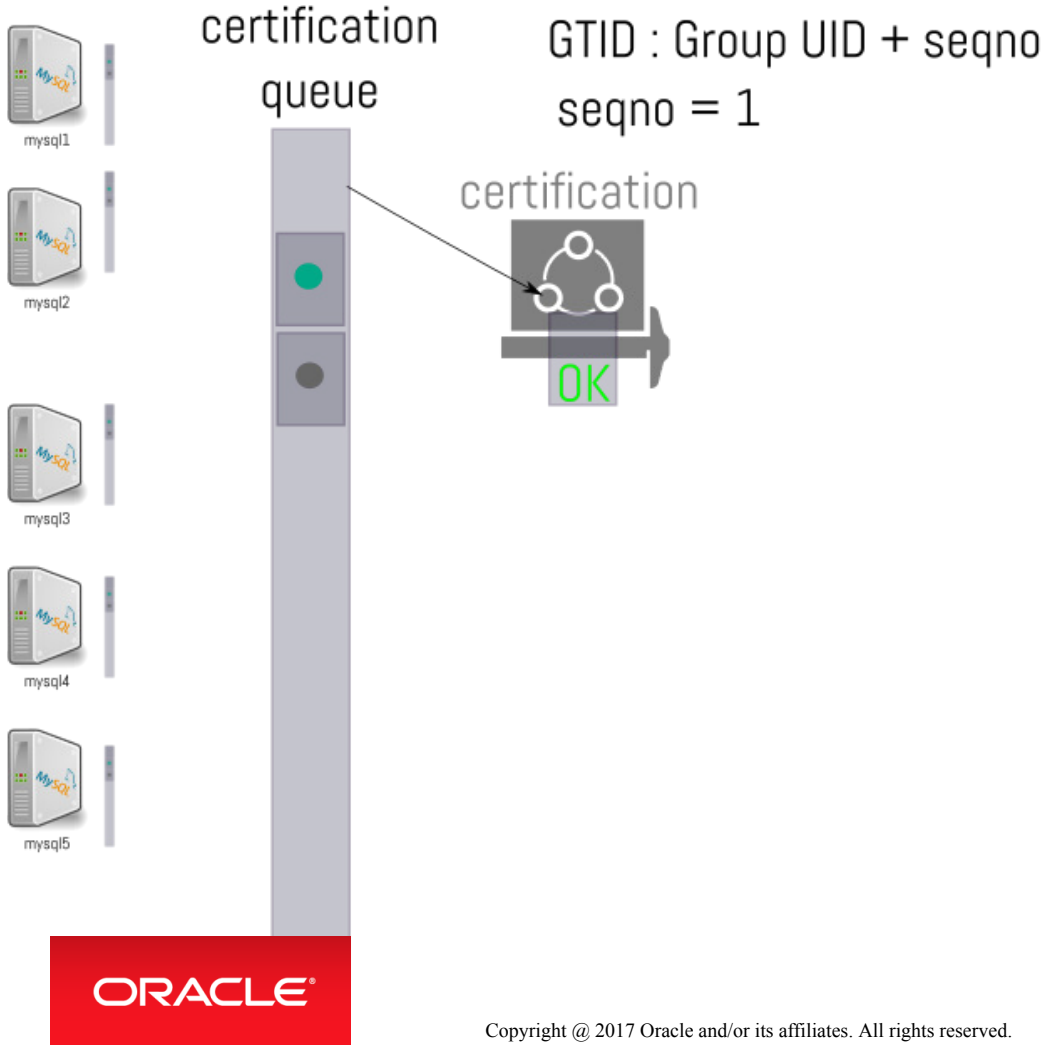
Group Replication : Total Order Delivery - GTID



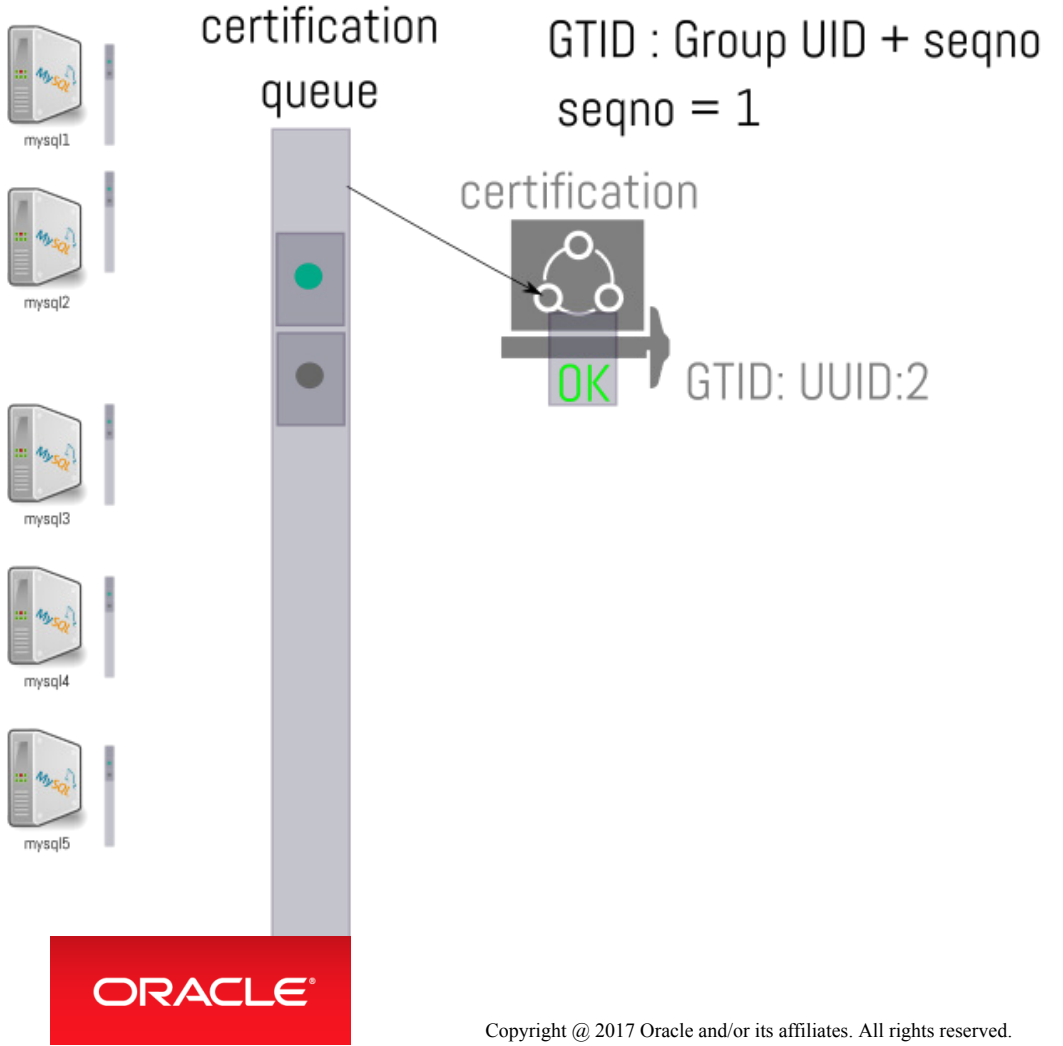
Group Replication : Total Order Delivery - GTID



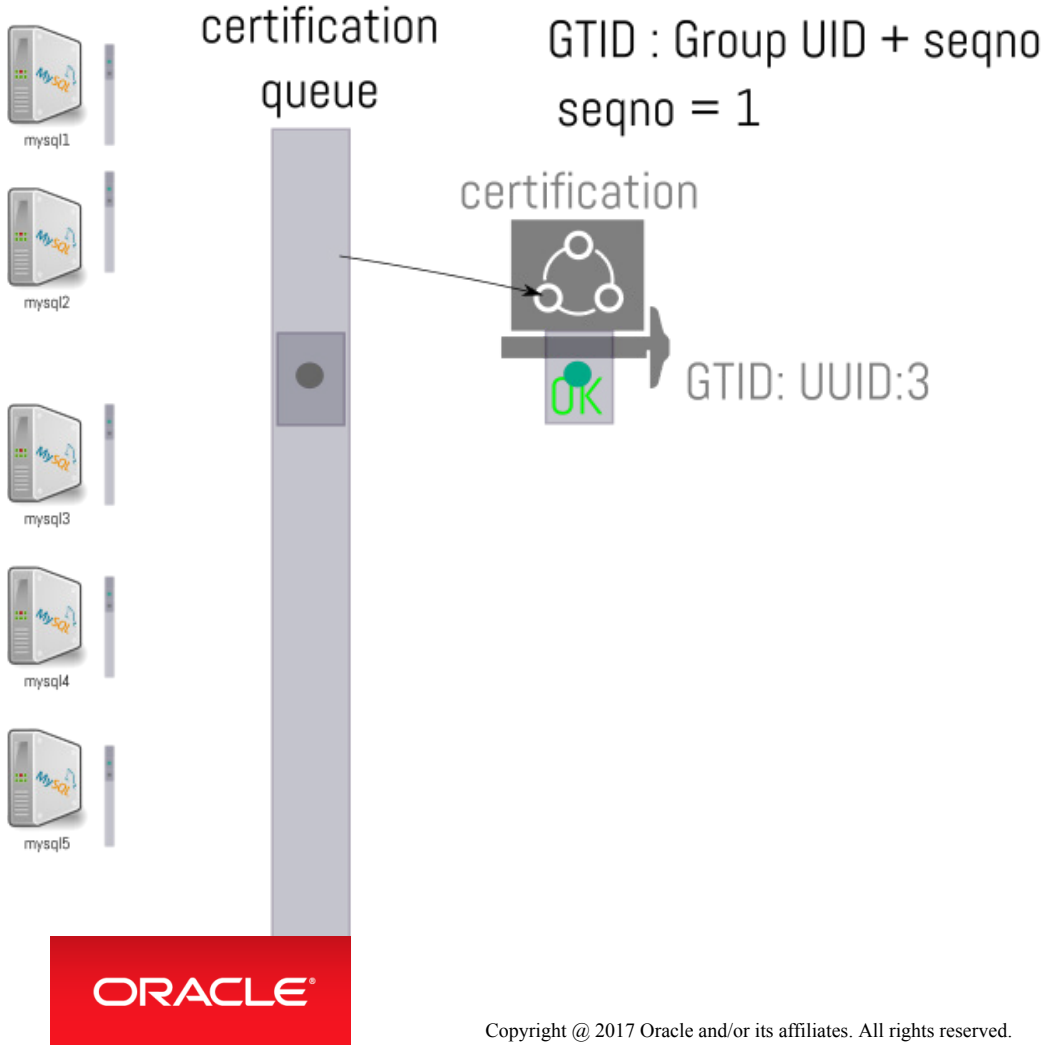
Group Replication : Total Order Delivery - GTID



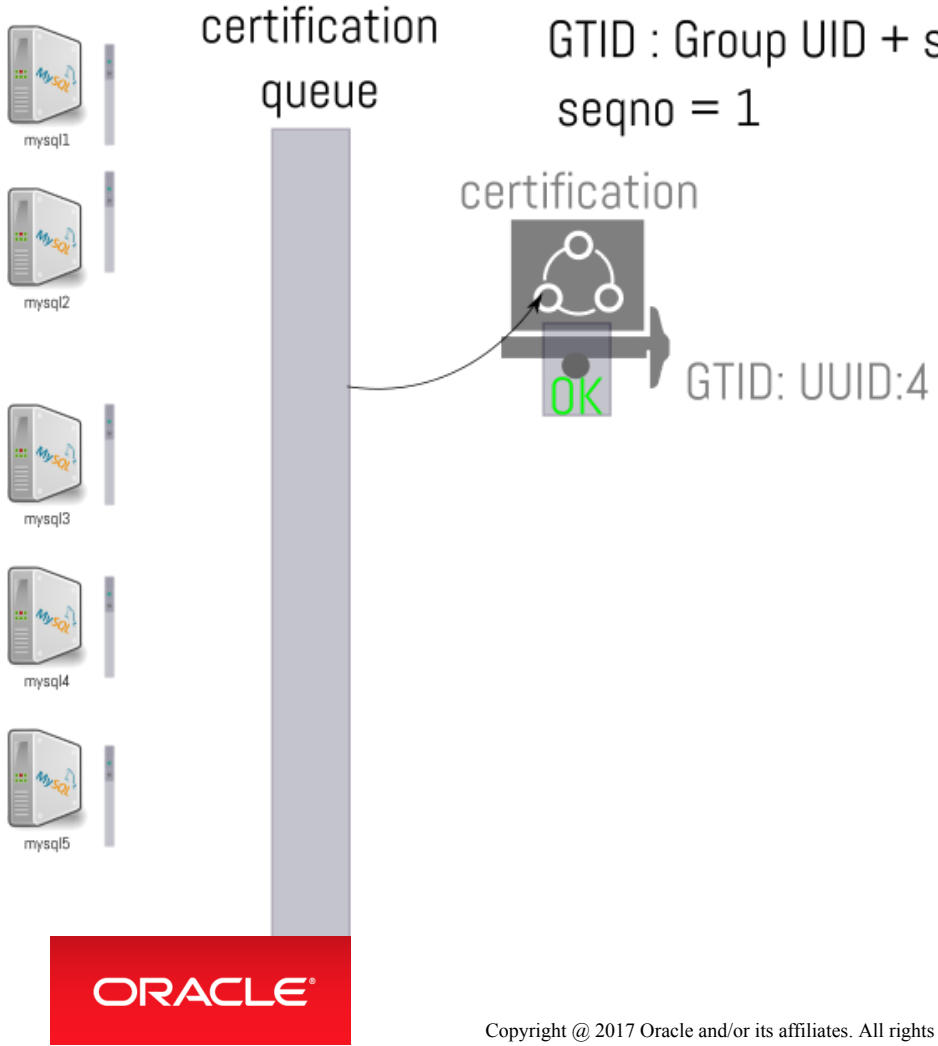
Group Replication : Total Order Delivery - GTID



Group Replication : Total Order Delivery - GTID

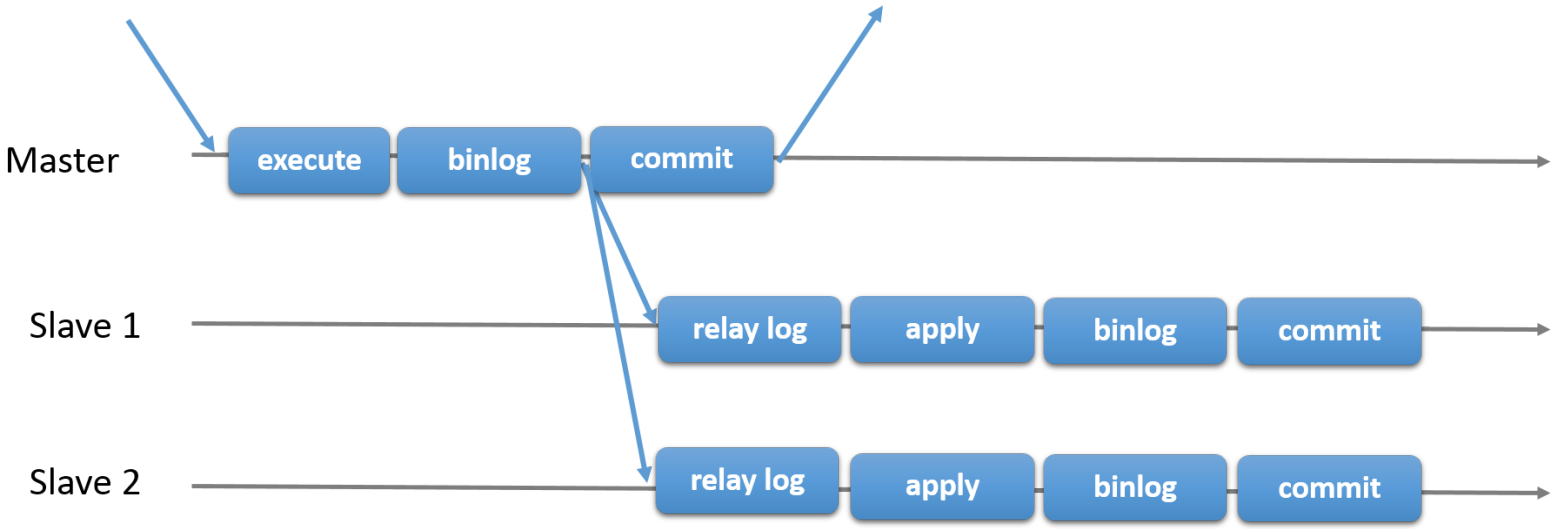


Group Replication : Total Order Delivery - GTID



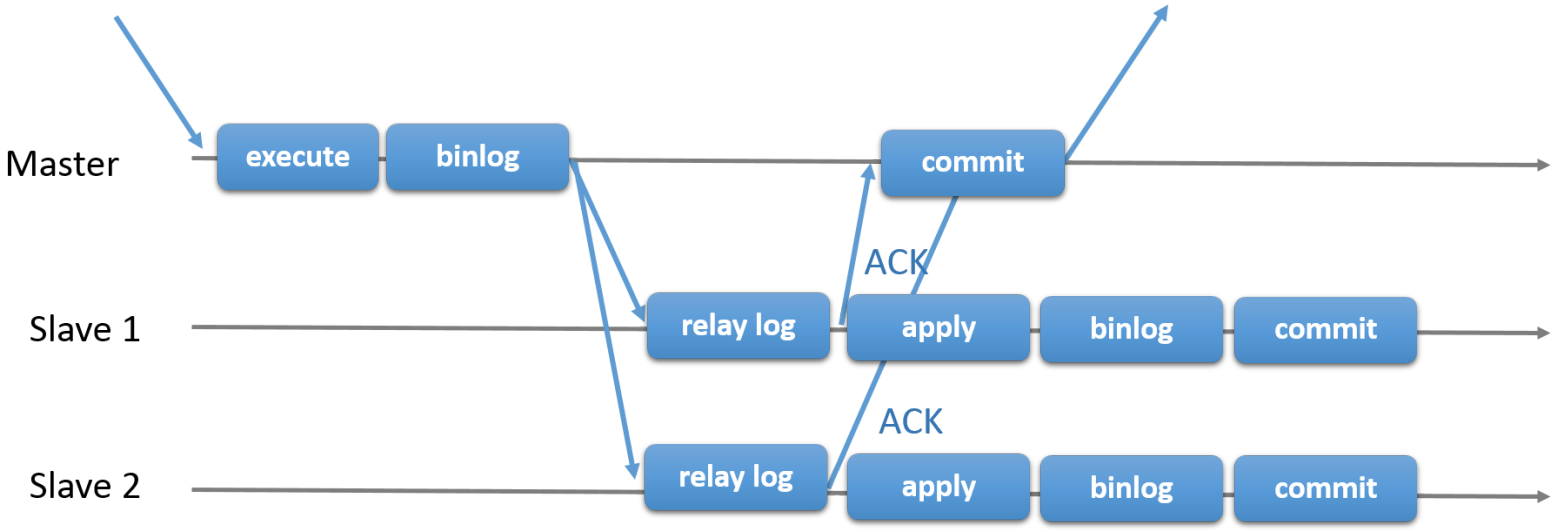
Group Replication: return commit

Asynchronous Replication:



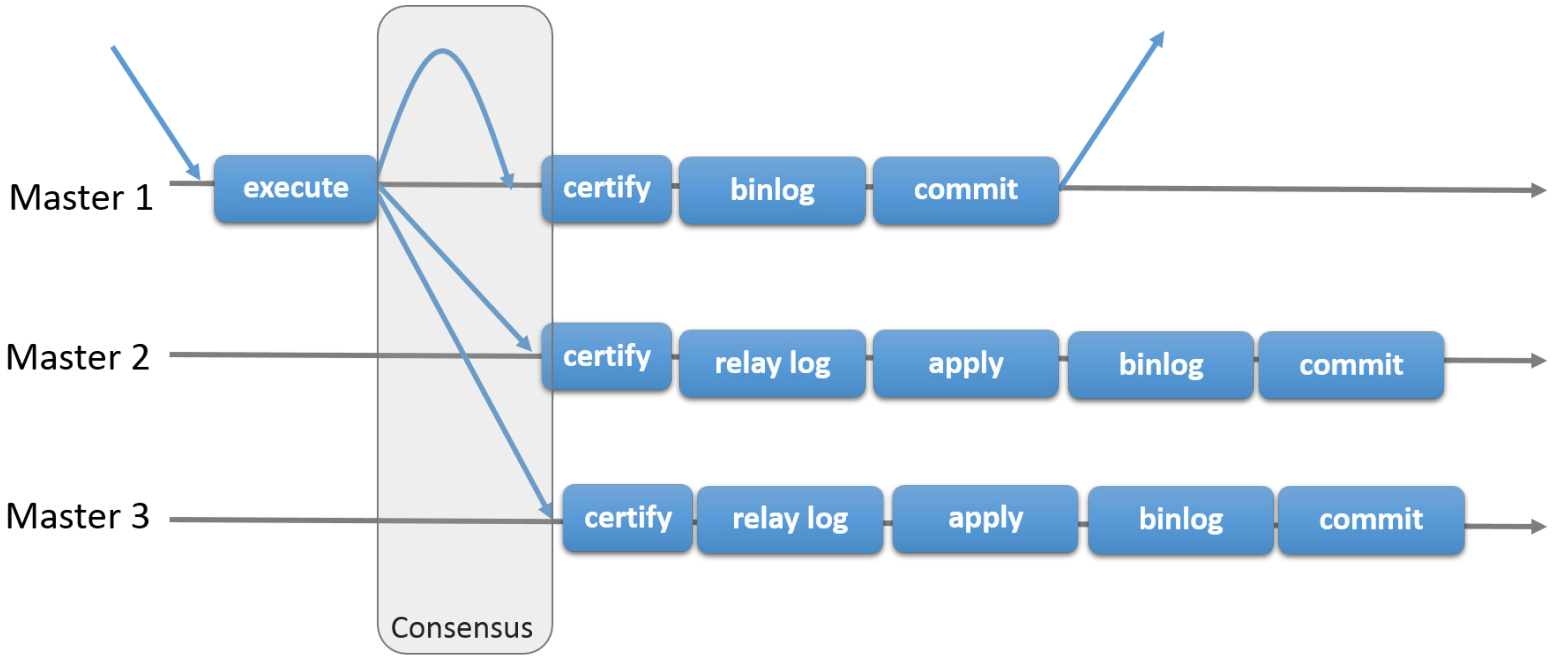
Group Replication: return from commit (2)

Semi-Sync Replication:



Group Replication: return from commit (3)

Group Replication:



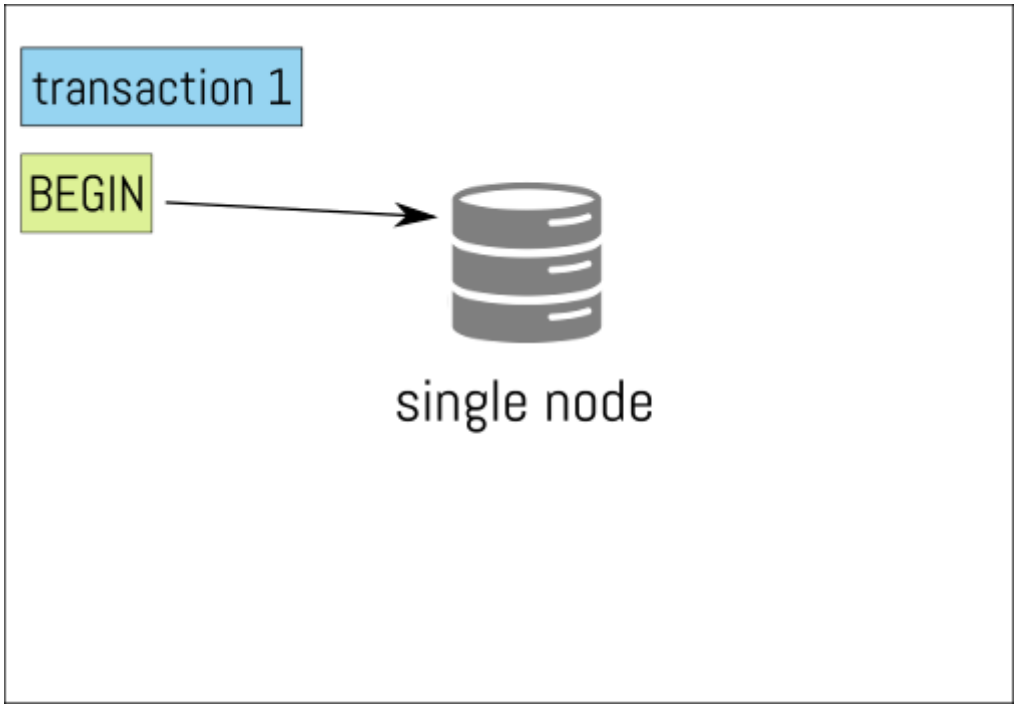
Group Replication : Optimistic Locking

Group Replication uses optimistic locking

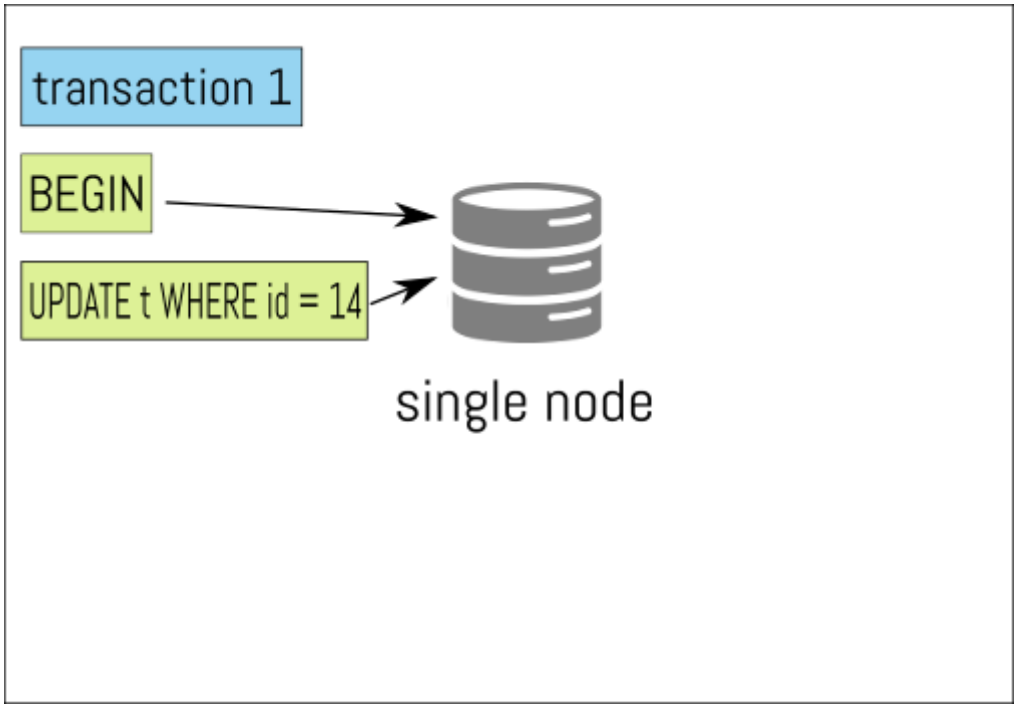
- during a transaction, **local (InnoDB) locking** happens
- **optimistically assumes** there will be no conflicts across nodes
  (no communication between nodes necessary)
- cluster-wide conflict resolution happens only at `COMMIT`, during **certification**

Let´s first have a look at the traditional locking to compare.

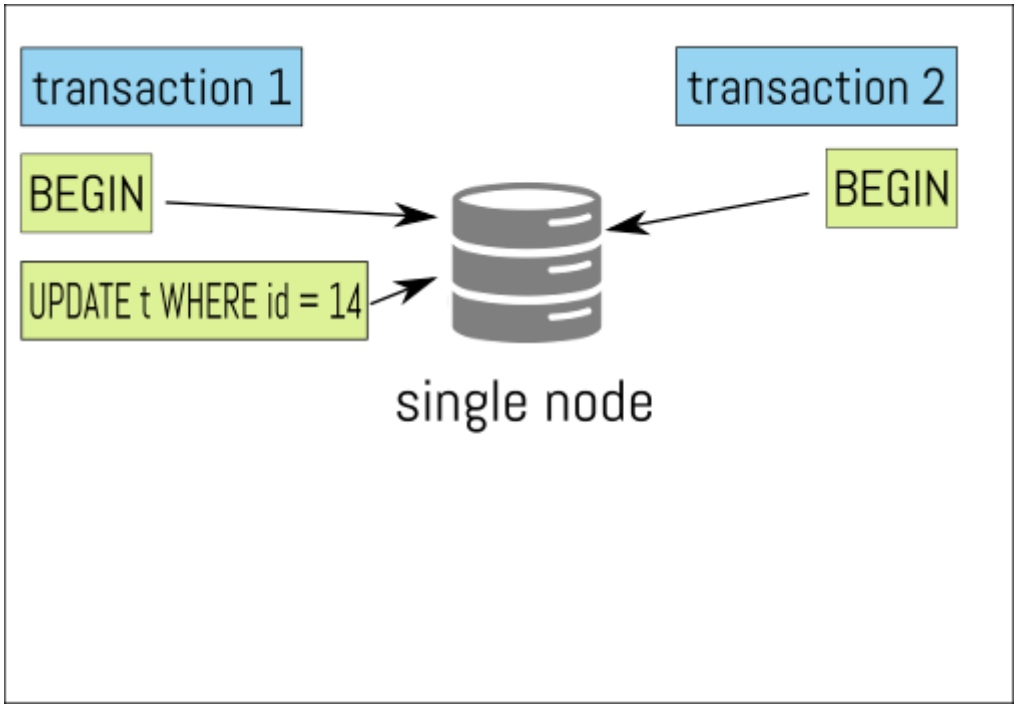
Traditional locking



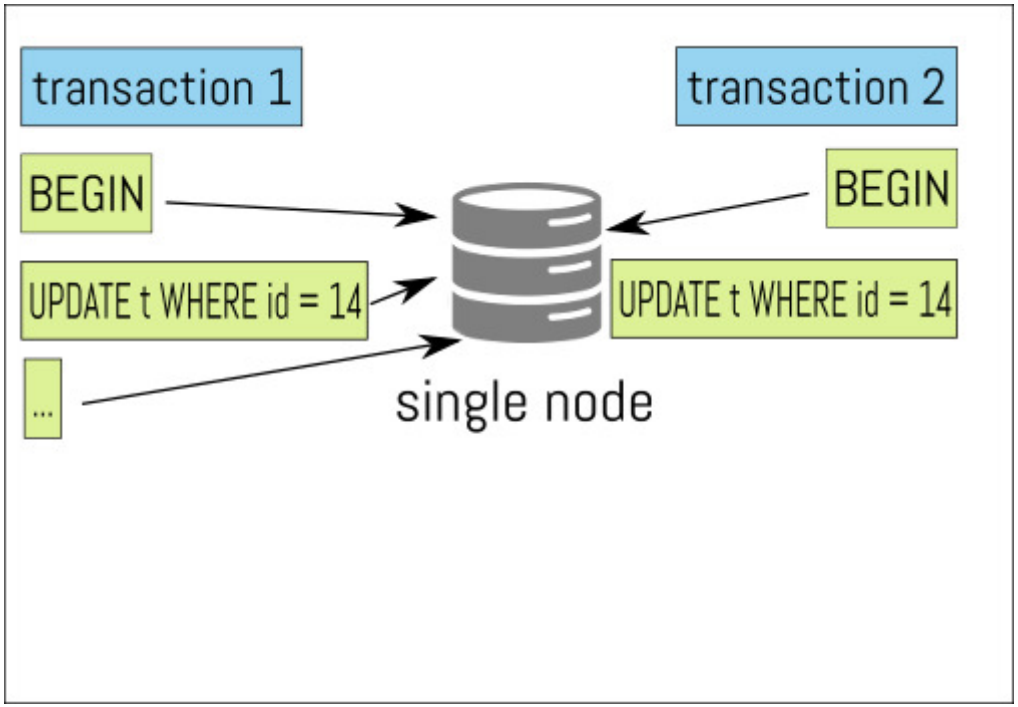
Traditional locking



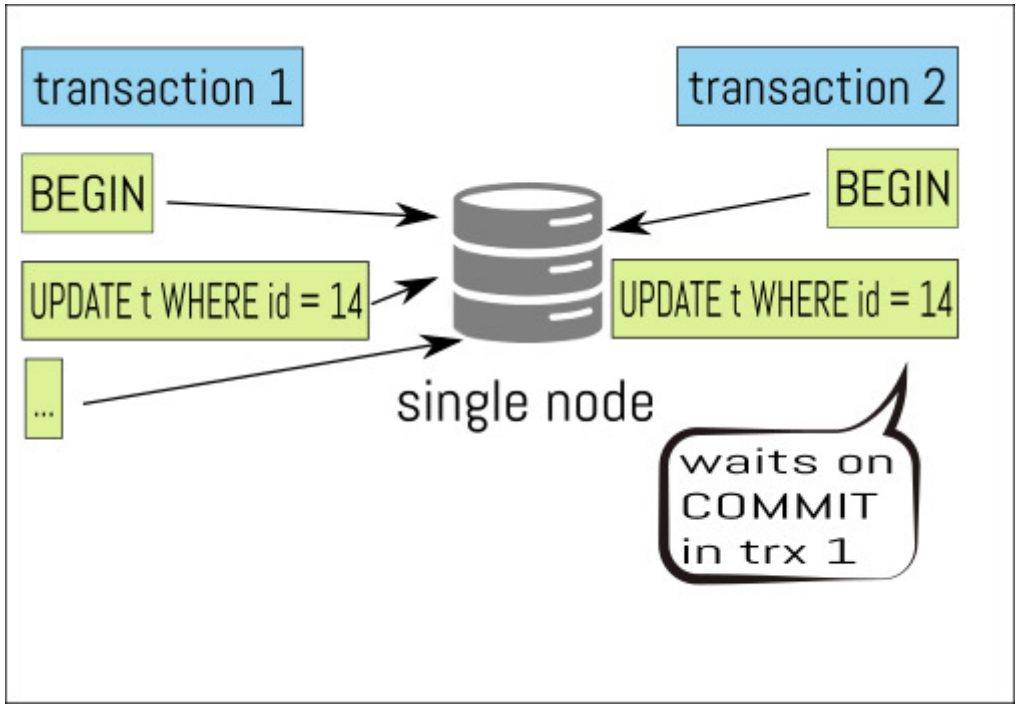
Traditional locking



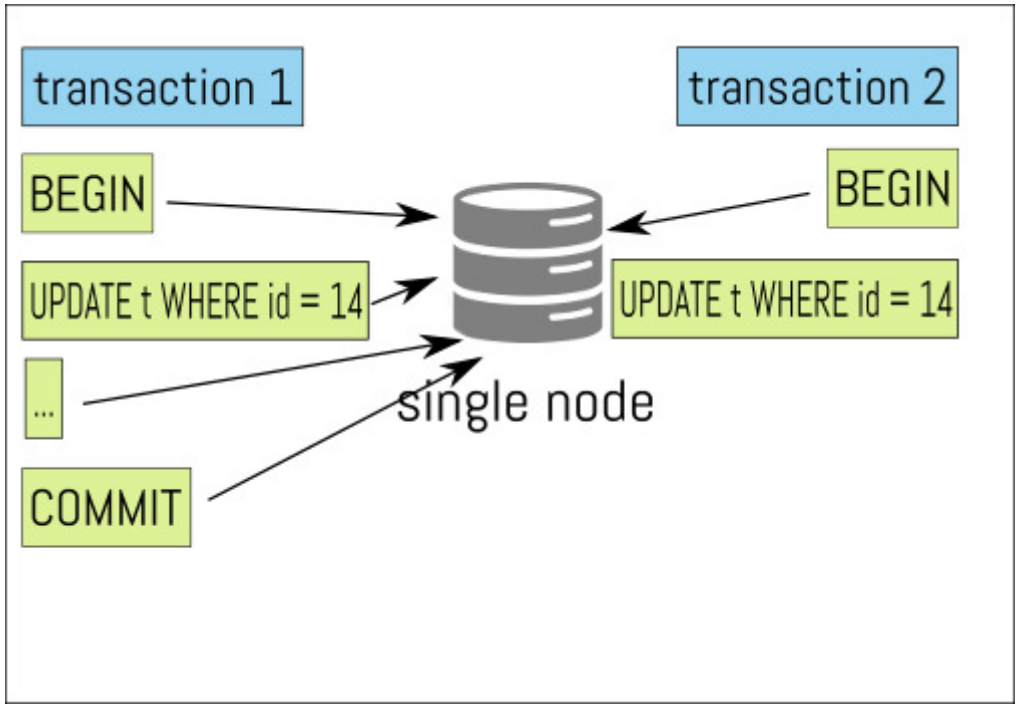
Traditional locking



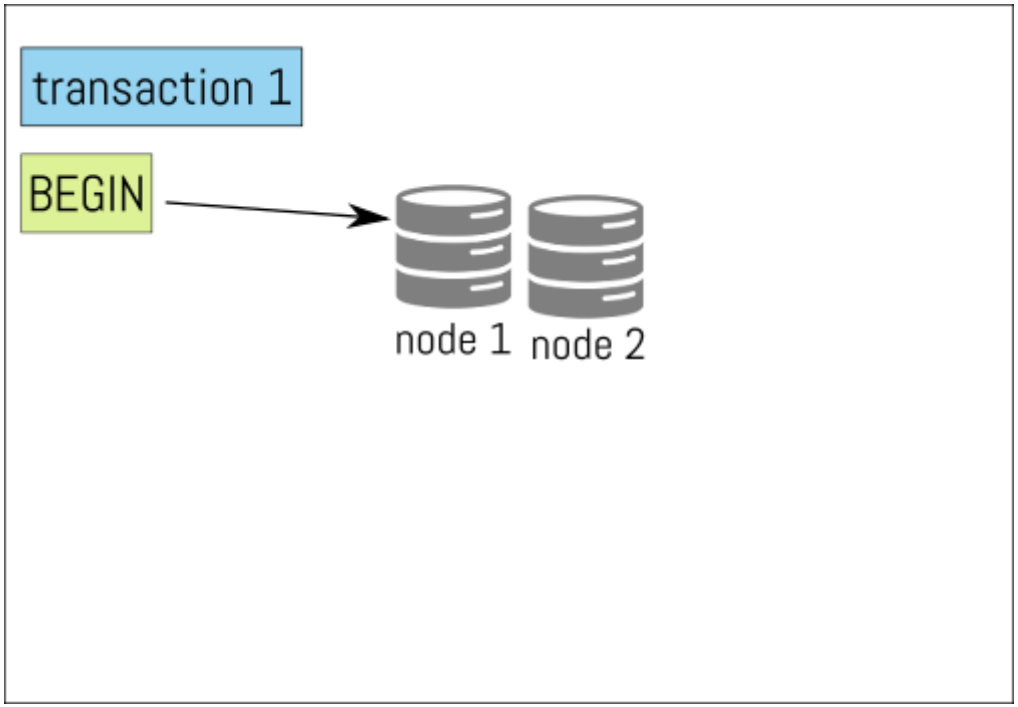
Traditional locking



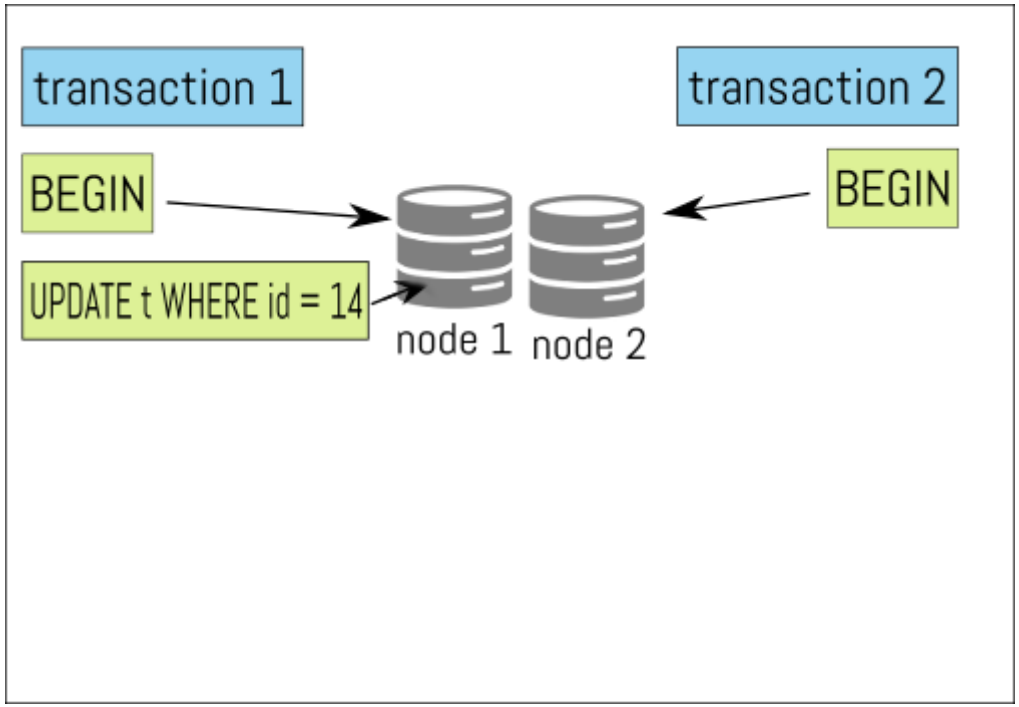
Traditional locking



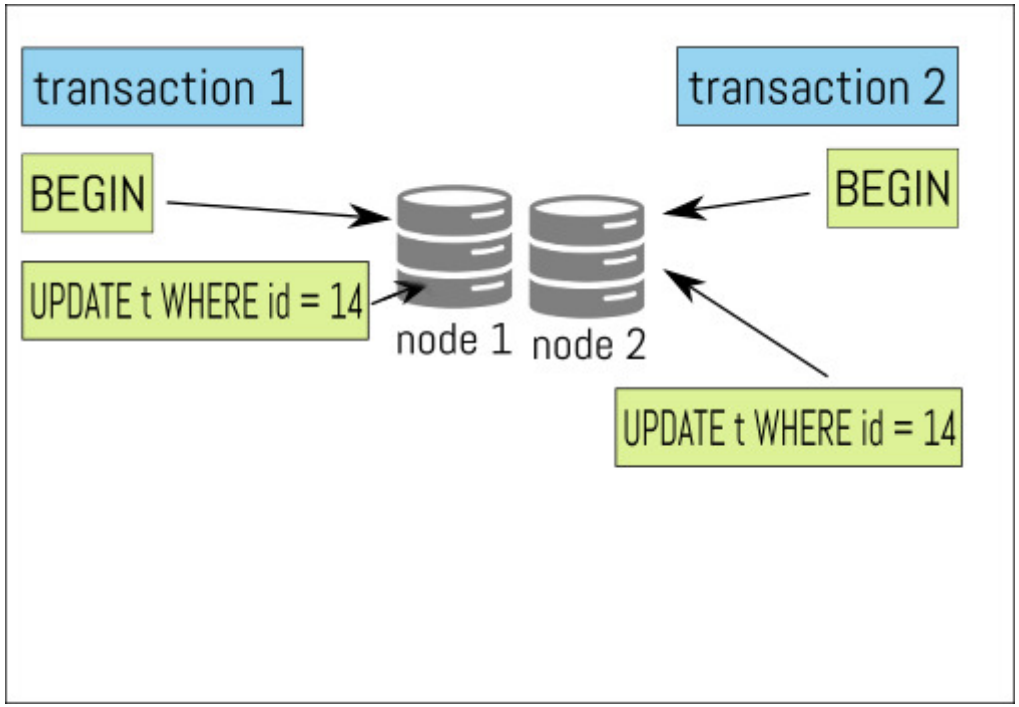
Optimistic Locking



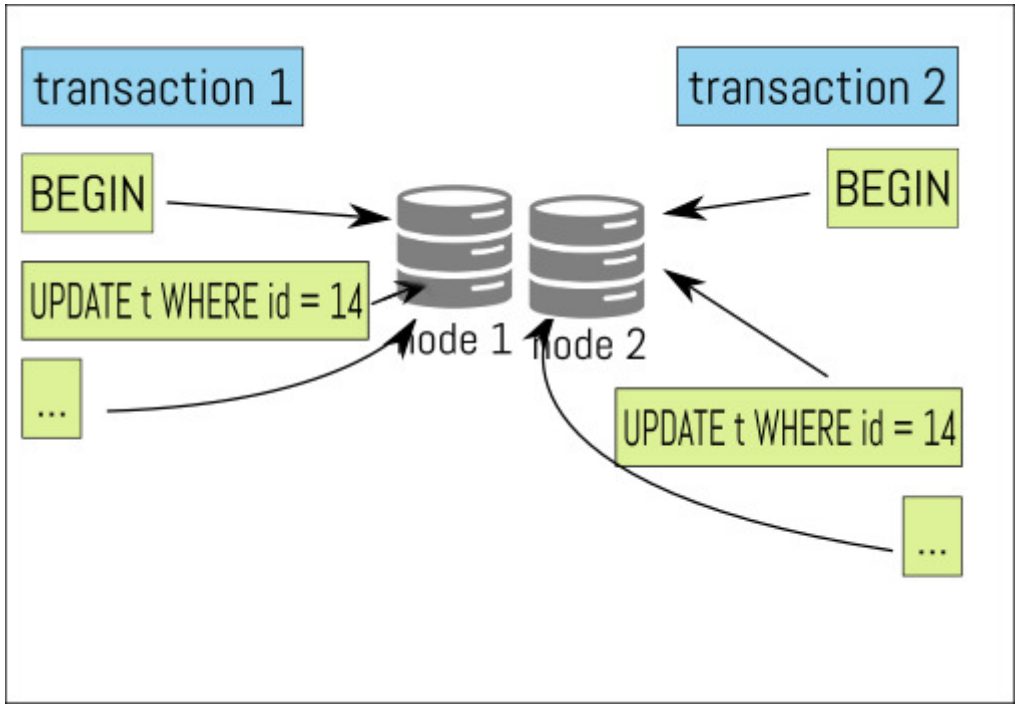
Optimistic Locking



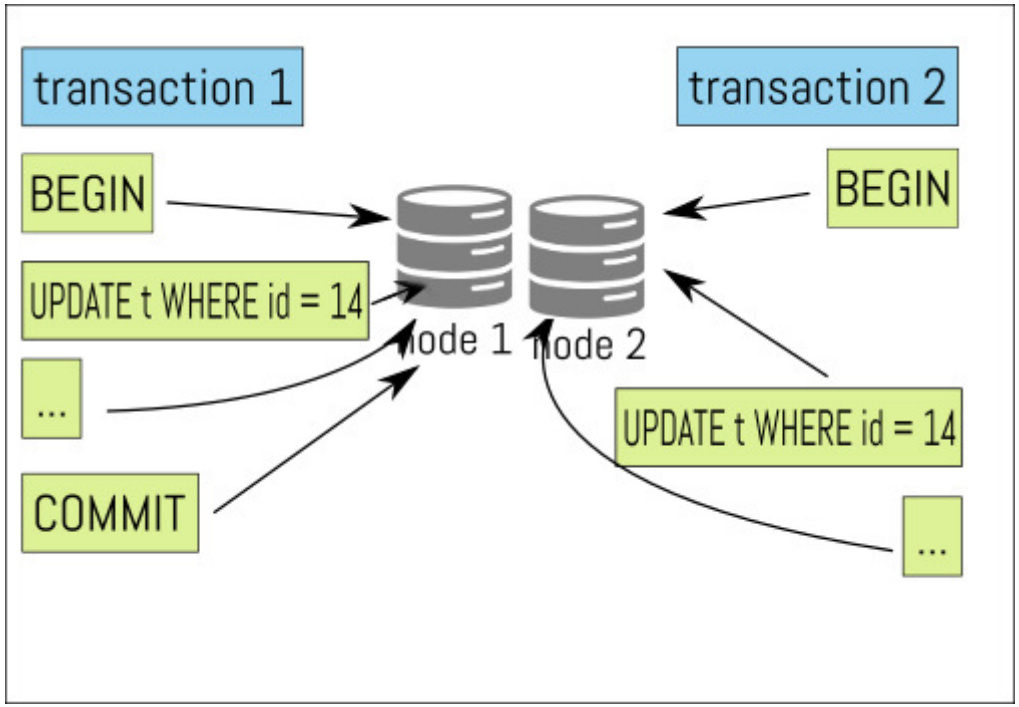
Optimistic Locking



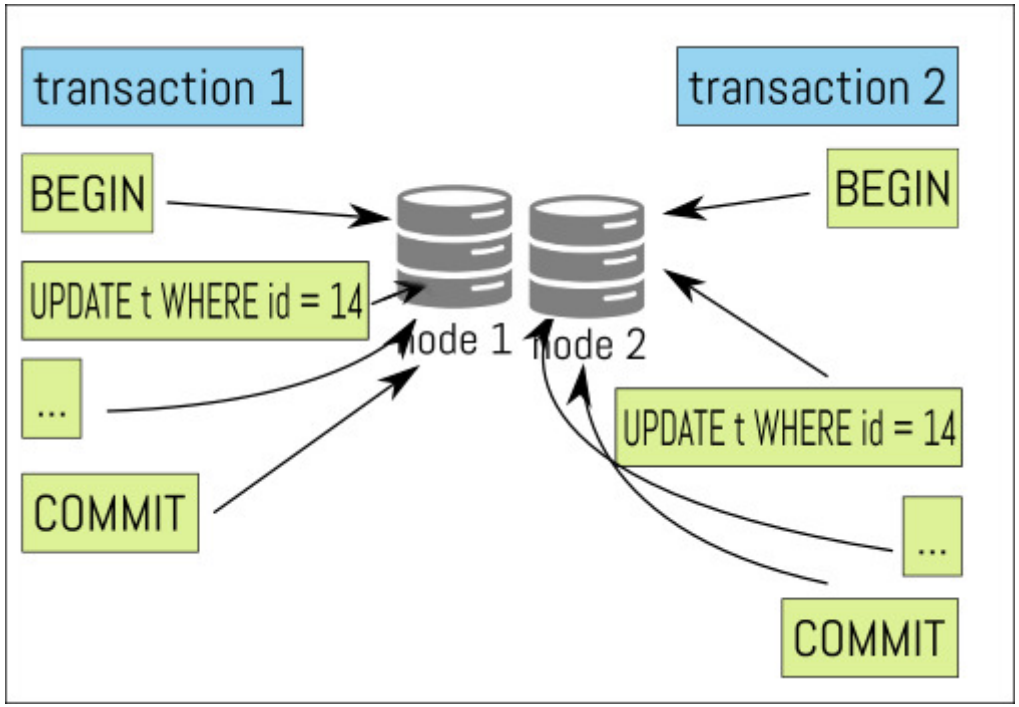
Optimistic Locking



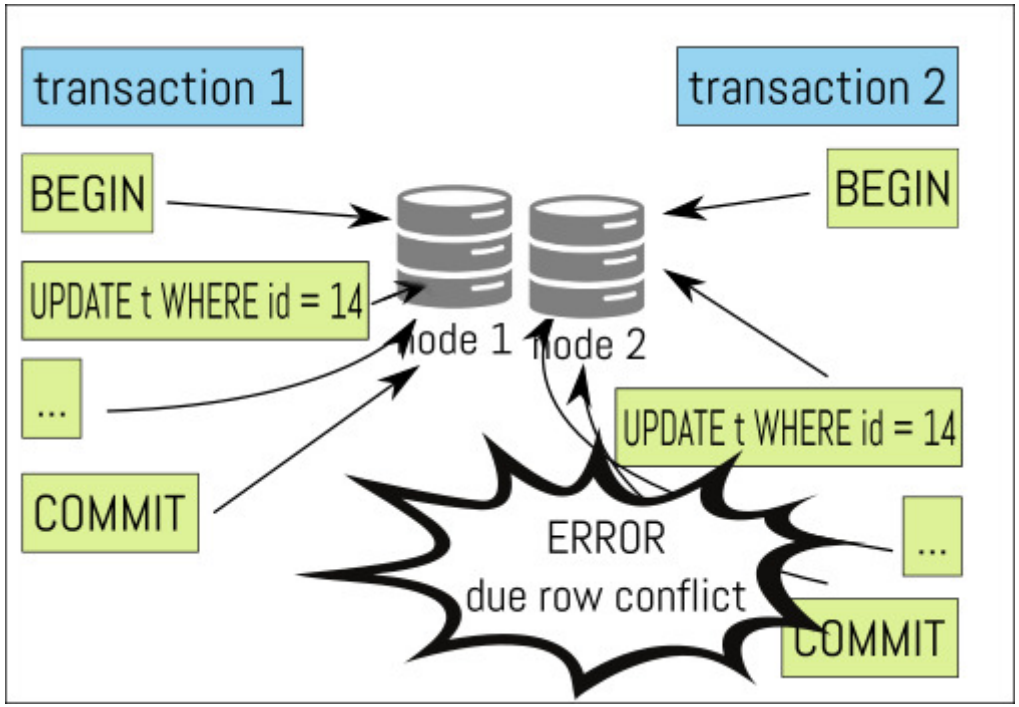
Optimistic Locking



Optimistic Locking



Optimistic Locking



The system returns error 149 as certification failed:

```
ERROR 1180 (HY000): Got error 149 during COMMIT
```

Certification

Certification is the process that only needs to answer the following unique question:

- ***can the write (transaction) be applied ?***
  - based on unapplied earlier transactions
  - such conflicts must come for other members/nodes

Certification (2)

- certification is a deterministic operation
- certification individually happens on every member/node
- communication with other members is not needed for certification
  - pass: enter in the apply queue
  - fail: drop the transaction
- serialized by the total order in GCS/Xcom + GTID
- first committer wins rule

Drawbacks of optimistic locking

having a first-committer-wins system means conflicts will more likely happen with:

- large transactions
- long running transactions

GTID

- GTIDs are the same as those used by asynchronous replication.

```
mysql>  SELECT * FROM performance_schema.replication_connection_status\G
*************************** 1. row ***************************
             CHANNEL_NAME: group_replication_applier
               GROUP_NAME: afb80f36-2bff-11e6-84e0-0800277dd3bf
              SOURCE_UUID: afb80f36-2bff-11e6-84e0-0800277dd3bf
                THREAD_ID: NULL
            SERVICE_STATE: ON
COUNT_RECEIVED_HEARTBEATS: 0
 LAST_HEARTBEAT_TIMESTAMP: 0000-00-00 00:00:00
  RECEIVED_TRANSACTION_SET: afb80f36-2bff-11e6-84e0-0800277dd3bf:1-57,
f037578b-46b1-11e6-8005-08002774c31b:1-48937
        LAST_ERROR_NUMBER: 0
       LAST_ERROR_MESSAGE:
     LAST_ERROR_TIMESTAMP: 0000-00-00 00:00:00
```

GTID

- but transactions use the Group´s GTID

```
mysql> show master status\G
*************************** 1. row ***************************
             File: mysql4-bin.000001
         Position: 1501
     Binlog_Do_DB:
 Binlog_Ignore_DB:
 Executed_Gtid_Set: afb80f36-2bff-11e6-84e0-0800277dd3bf:1-57,
f037578b-46b1-11e6-8005-08002774c31b:1-48937
```

Requirements

- exclusively works with InnoDB tables
- every table must have a PK defined
- only IPV4 is supported
- a good network with low latency is important
- maximum of 9 members per group
- `log-bin` must be enabled and only `binlog_format=ROW` is supported

Requirements (2)

- enable GTIDs
- replication meta-data must be stored in system tables

```
--master-info-repository=TABLE
--relay-log-info-repository=TABLE
```

- writesets extraction must be enabled **TODO: I need to check on 5.7.19 and 8.0.1**

```
--transaction-write-set-extraction=XXHASH64
```

- `log-slave-updates` must also **be enabled**

Limitations

- binlog checksum is not supported

```
--binlog-checksum=NONE
```

- savepoints **were** not supported before 5.7.19 & 8.0.1
- *SERIALIZABLE* is not supported as transaction isolation level
- http://lefred.be/content/mysql-group-replication-limitations-savepoints/
- http://lefred.be/content/mysql-group-replication-and-table-design/

Is my workload ready for Group Replication ?

As the writesets (transactions) are replicated to all available nodes on commit, and as they are certified on every node, a very large writeset could increase the amount of certification errors.

Additionally, changing the same record on all the nodes (hotspot) concurrently will also cause problems.

And finally, the certification uses the primary key of the tables, a table without PK is also a problem.

Is my workload ready for Group Replication ?

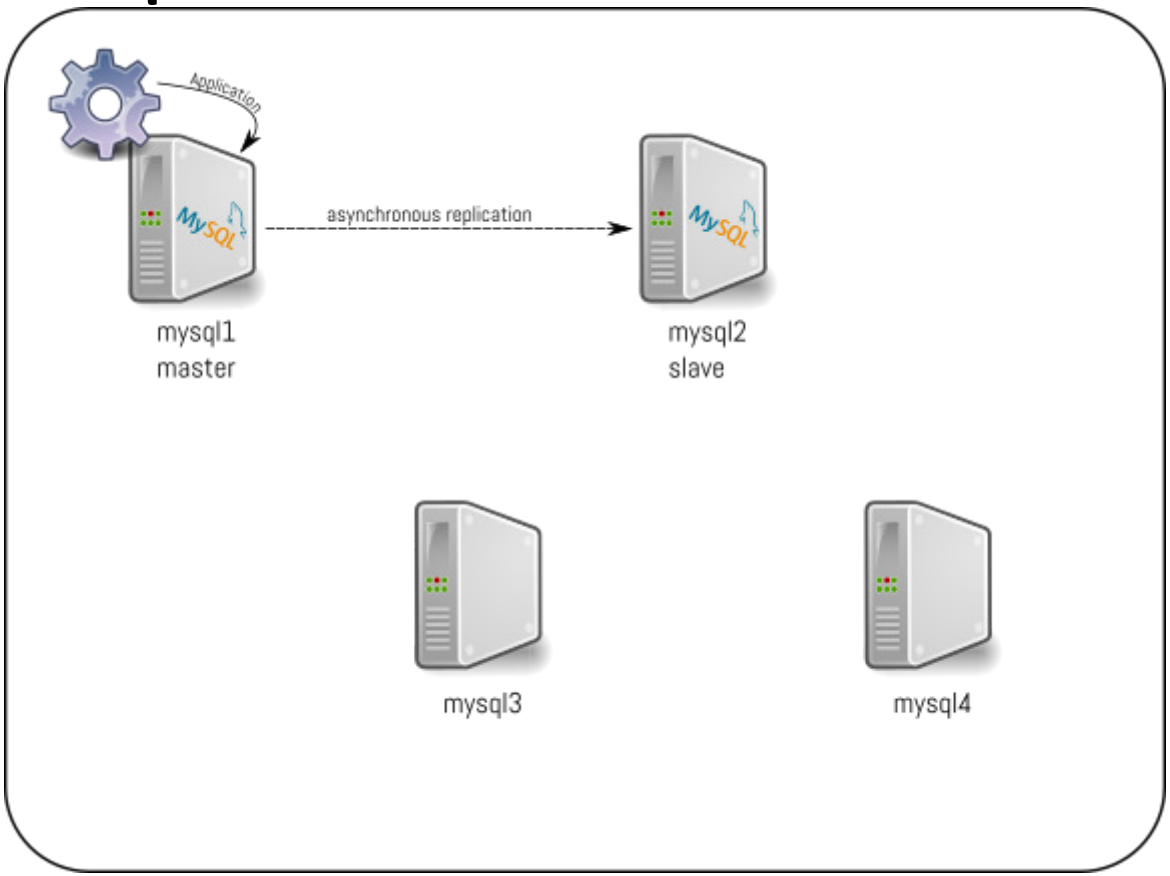
Therefore, when using Group Replication, we should pay attention to these points:

- PK is mandatory (and a good one is better)
- avoid large transactions
- avoid hotspot

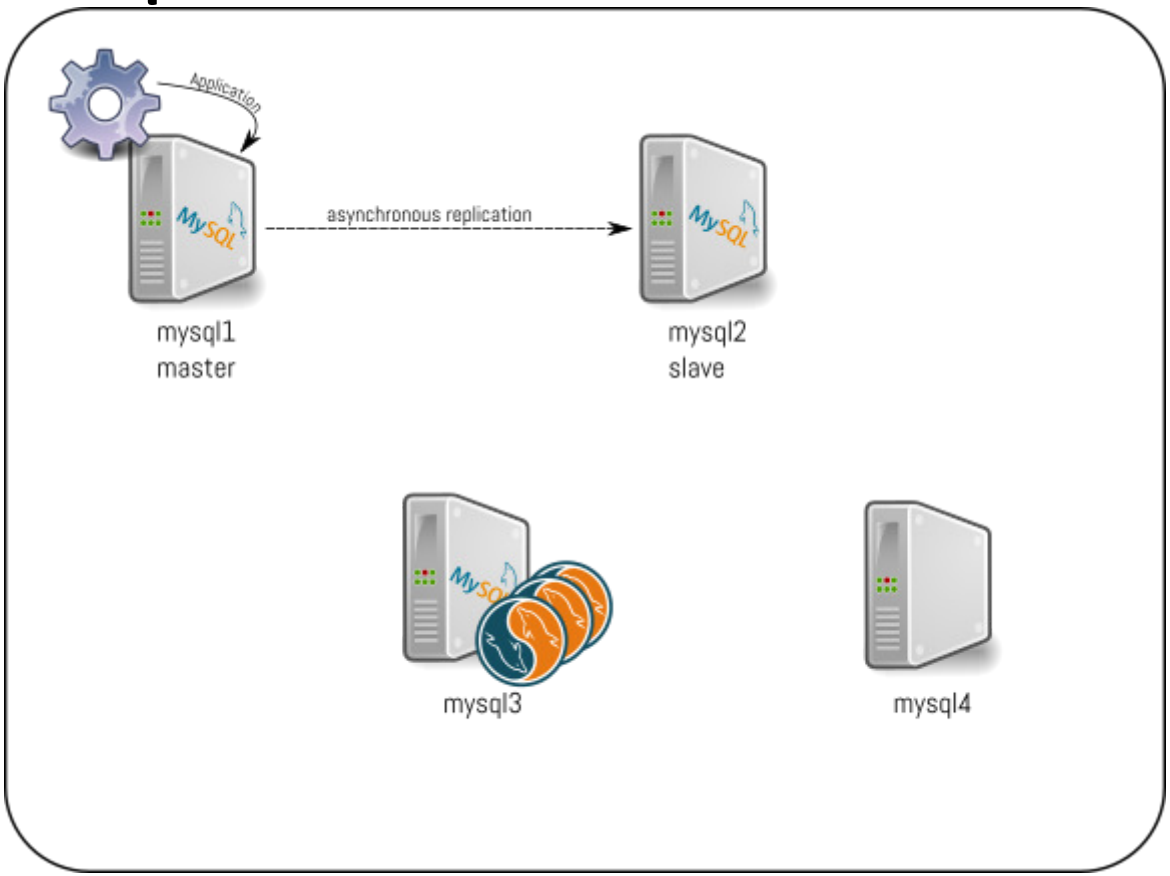
ready ?

Migration from Master-Slave to GR

The plan

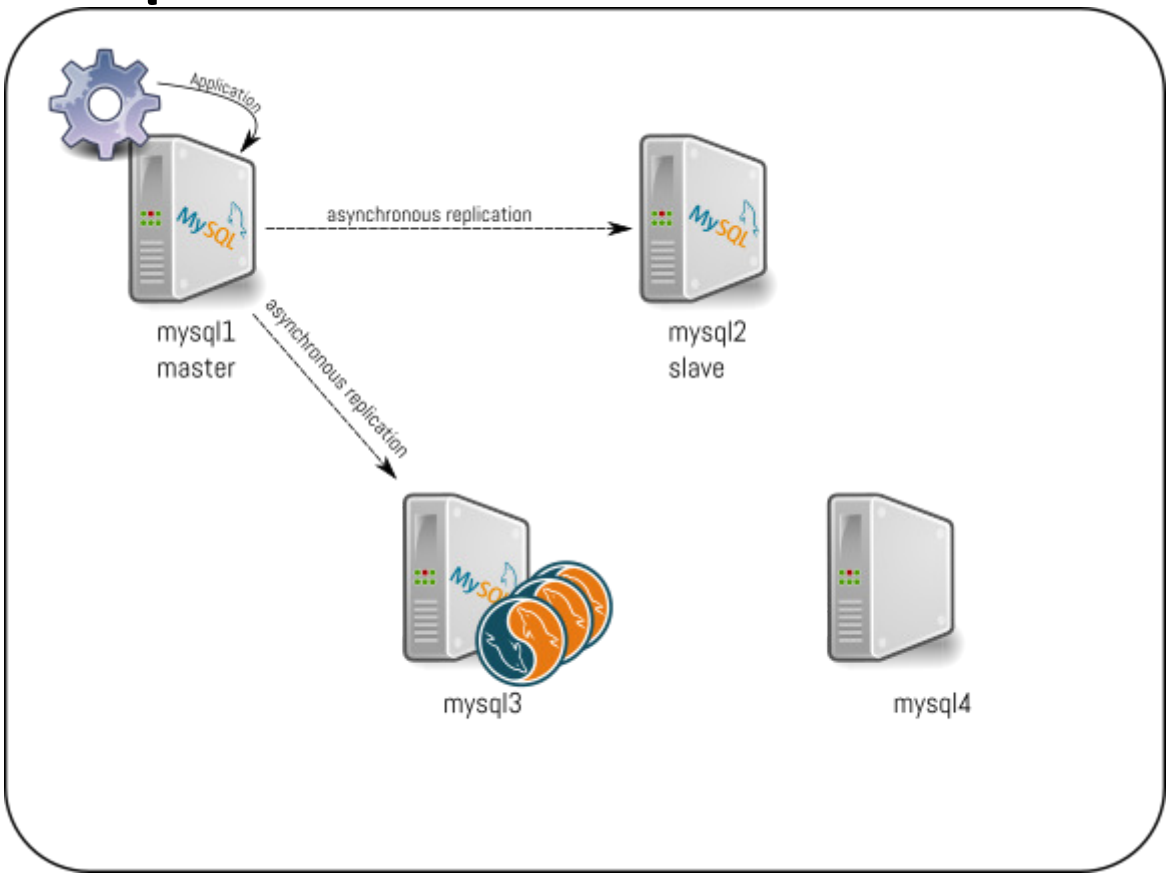


The plan



1) We install and setup MySQL InnoDB Cluster on one of the new servers

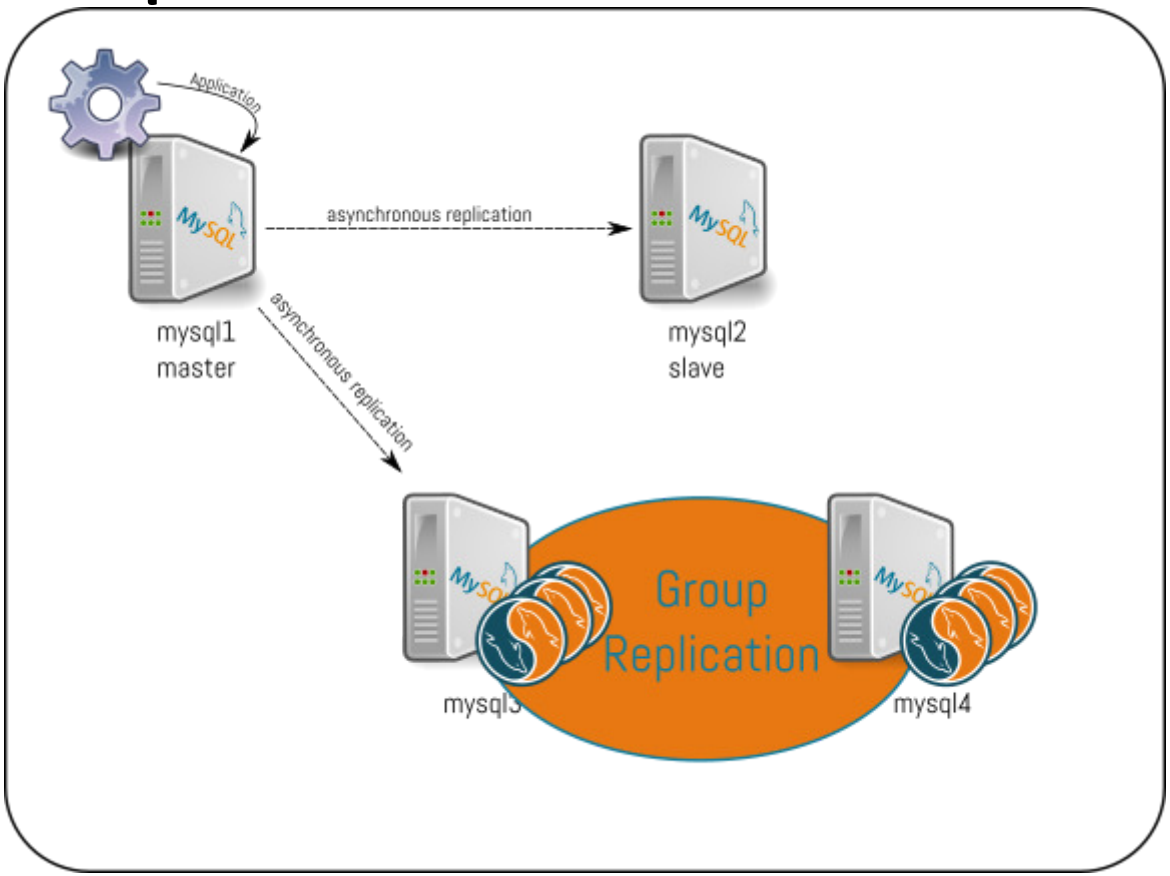
The plan



2) We restore a backup

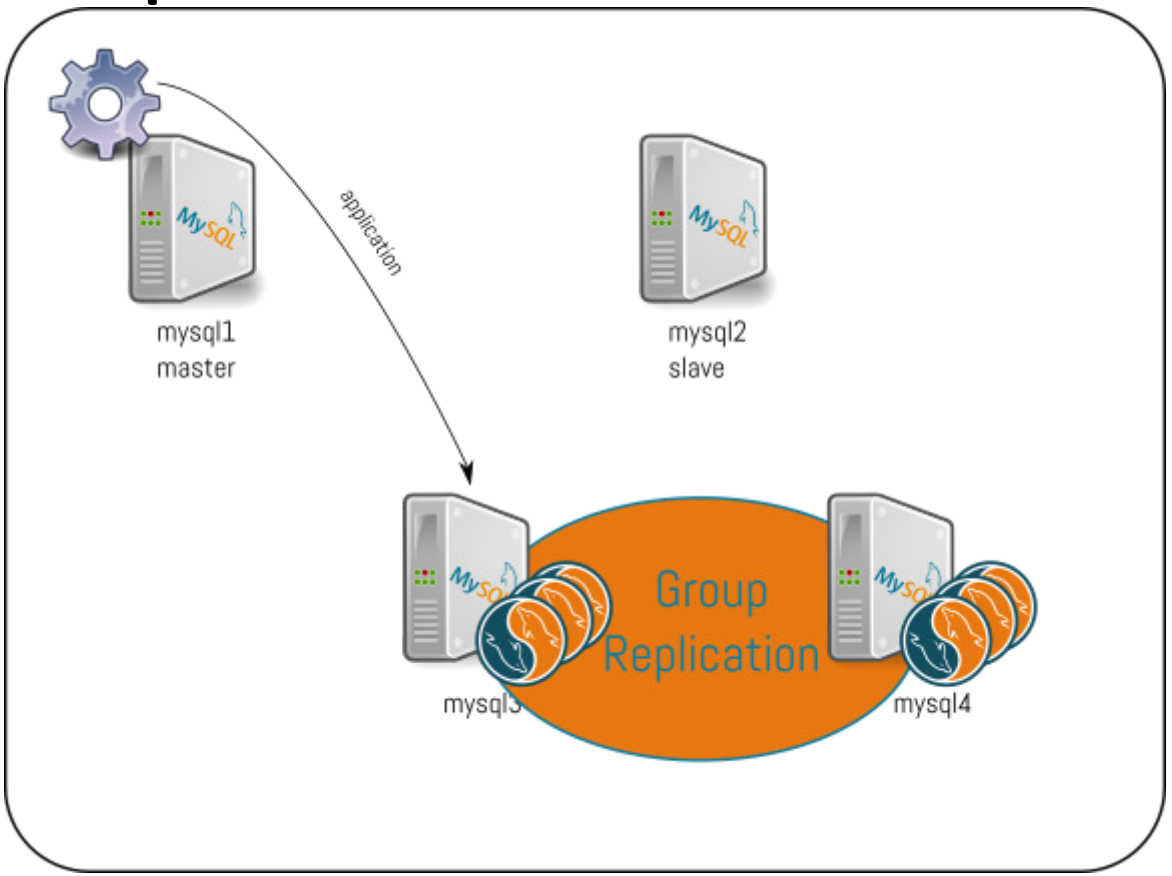
3) setup asynchronous replication on the new server.

The plan



4) We add a new instance to our group

The plan

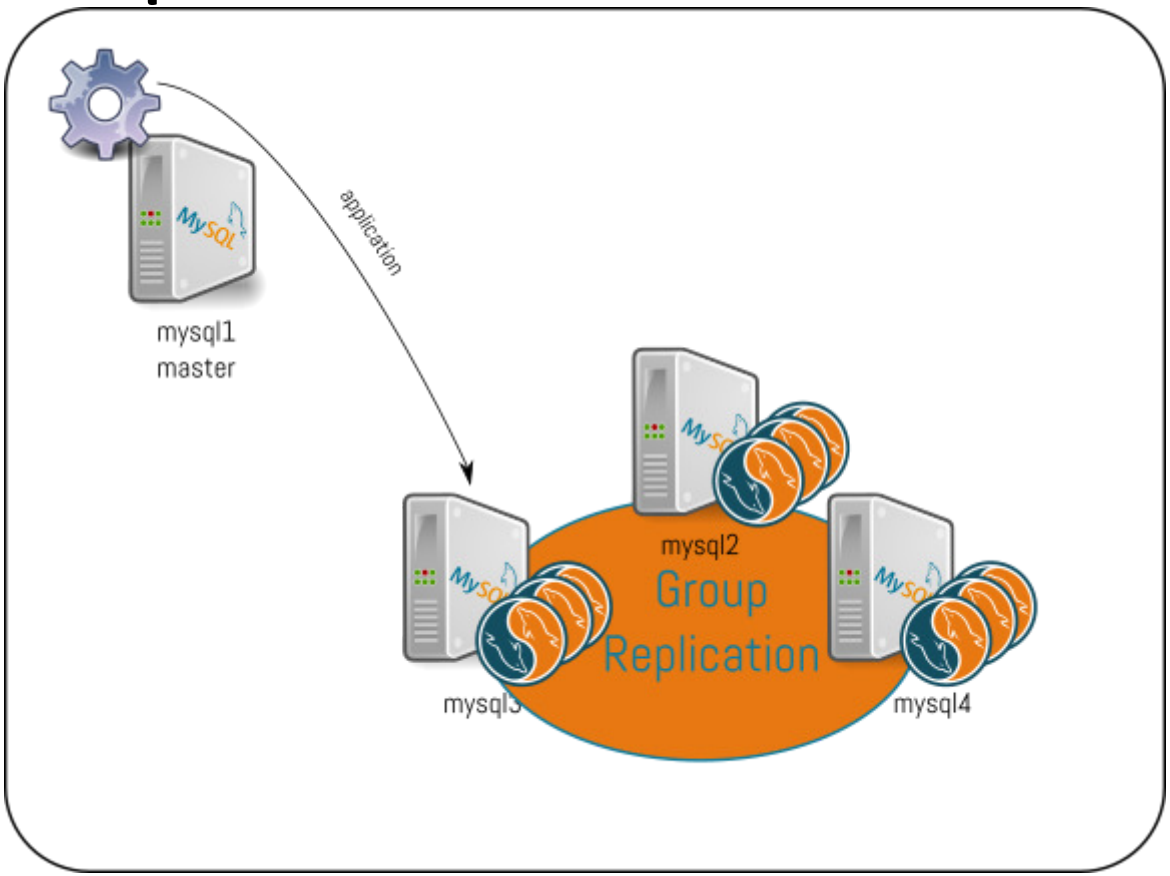


5) We point the application to one of our new nodes.

6) We wait and check that asynchronous replication is caught up

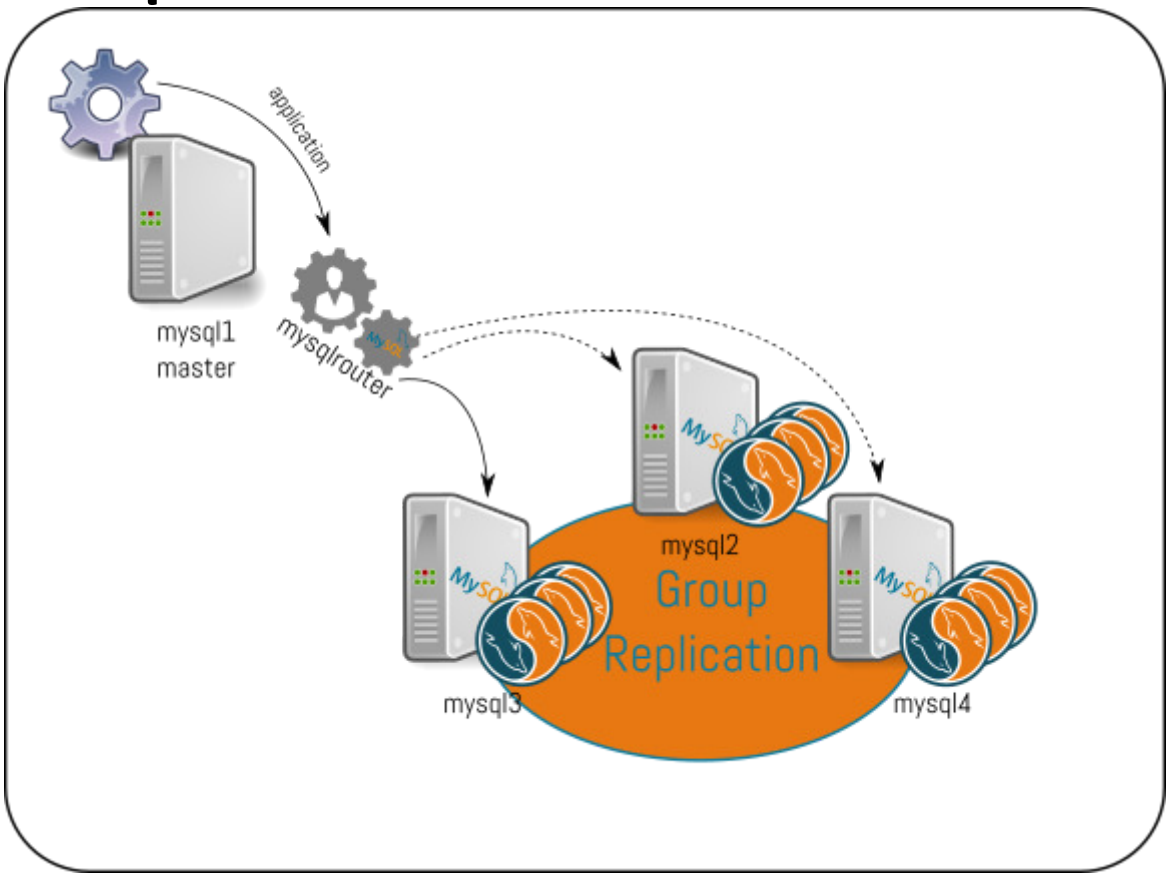
7) we stop those asynchronous slaves

The plan

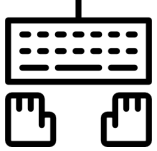


8) We attach the mysql2 slave to the group

The plan



9) Use MySQL Router for directing traffic



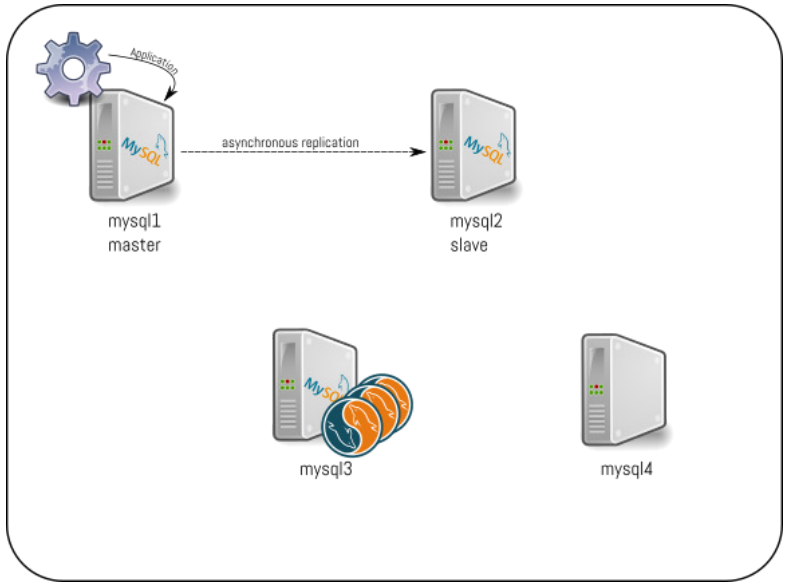
LAB2: Prepare mysql3

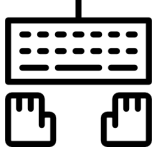
Asynchronous slave

Latest MySQL 5.7 is already installed on `mysql3`.

Let´s take a backup on `mysql1` using **meb**:

```
[mysql1 ~]# mysqlbackup \
  --host=127.0.0.1 \
  --backup-dir=/tmp/backup \
  --user=root --password=X \
  backup-and-apply-log
```





LAB2: Prepare mysql3 (2)

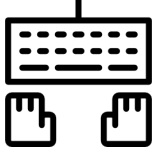
Asynchronous slave

Copy the backup from `mysql1` to `mysql3`:

```
[mysql1 ~]# scp -r /tmp/backup mysql3:/tmp
```

And restore it:

```
[mysql3 ~]# mysqlbackup --backup-dir=/tmp/backup --force copy-back
[mysql3 ~]# rm /var/lib/mysql/mysql*-bin*     # just some cleanup
[mysql3 ~]# chown -R mysql. /var/lib/mysql
```

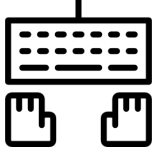


LAB3: mysql3 as asynchronous slave (2)

Asynchronous slave

Configure `/etc/my.cnf` with the minimal requirements:

```
[mysqld]
...
server_id=3
enforce_gtid_consistency = on
gtid_mode  = on
log_bin
log_slave_updates
```

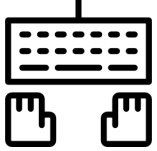


LAB2: Prepare mysql3 (3)

Asynchronous slave

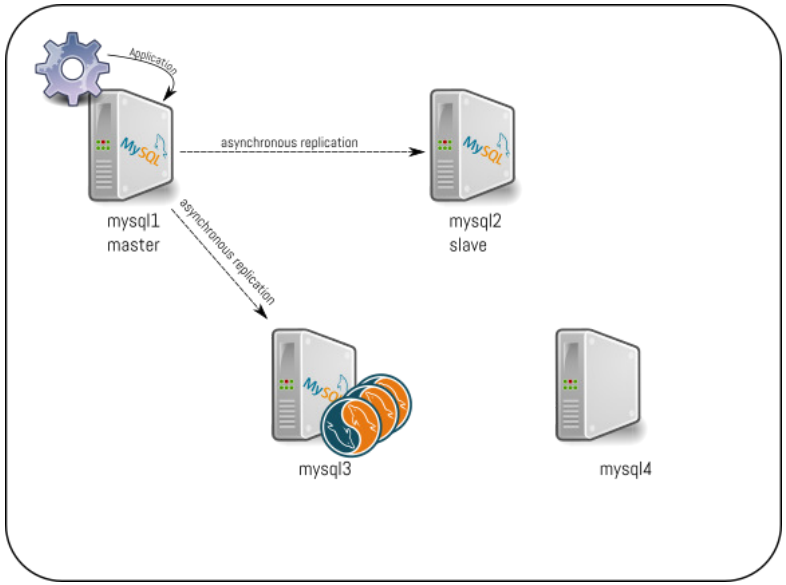
Let´s start MySQL on `mysql3`:

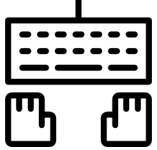
```
[mysql3 ~]# systemctl start mysqld
```



LAB3: mysql3 as asynchronous slave (1)

- find the GTIDs purged
- change MASTER
- set the purged GTIDs
- start replication





LAB3: mysql3 as asynchronous slave (2)

Find the latest purged GTIDs:

```
[mysql3 ~]# cat /tmp/backup/meta/backup_gtid_executed.sql
  SET @@GLOBAL.GTID_PURGED='33351000-3fe8-11e7-80b3-08002718d305:1-1002';
```

Connect to mysql3 and setup replication:

```
mysql> CHANGE MASTER TO MASTER_HOST="mysql1",
       MASTER_USER="repl_async", MASTER_PASSWORD='Xslave',
       MASTER_AUTO_POSITION=1;

mysql> RESET MASTER;
mysql> SET global gtid_purged="VALUE FOUND PREVIOUSLY";
mysql> START SLAVE;
```

Check that you receive the application´s traffic

Administration made easy and more...

MySQL-Shell

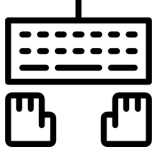
MySQL Shell

The MySQL Shell is an interactive Javascript, Python, or SQL interface supporting development and administration for the MySQL Server and is a component of the MySQL Server. You can use the MySQL Shell to perform data queries and updates as well as various administration operations.

MySQL Shell (2)

The MySQL Shell provides:

- Both Interactive and Batch operations
- Document and Relational Models
- CRUD Document and Relational APIs via scripting
- Traditional Table, JSON, Tab Separated output results formats
- MySQL Standard and X Protocols
- and more...



LAB4: MySQL InnoDB Cluster

Create a single instance cluster

Time to use the new MySQL Shell !

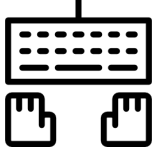
```
[mysql3 ~]# mysqlsh
```

Let´s verify if our server is ready to become a member of a new cluster:

```
mysql-js> dba.checkInstanceConfiguration('root@mysql3:3306')
```

Change the configuration !

```
mysql-js> dba.configureLocalInstance()
```



LAB4: MySQL InnoDB Cluster (2)

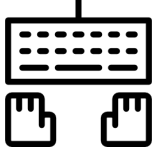
Restart mysqld to use the new configuration:

```
[mysql3 ~]# systemctl restart mysqld
```

Create a single instance cluster

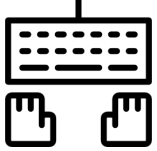
```
[mysql3 ~]# mysqlsh
```

```
mysql-js> dba.checkInstanceConfiguration('root@mysql3:3306')

mysql-js> \c root@mysql3:3306

mysql-js> cluster = dba.createCluster('MyInnoDBCluster')
```



LAB4: Cluster Status

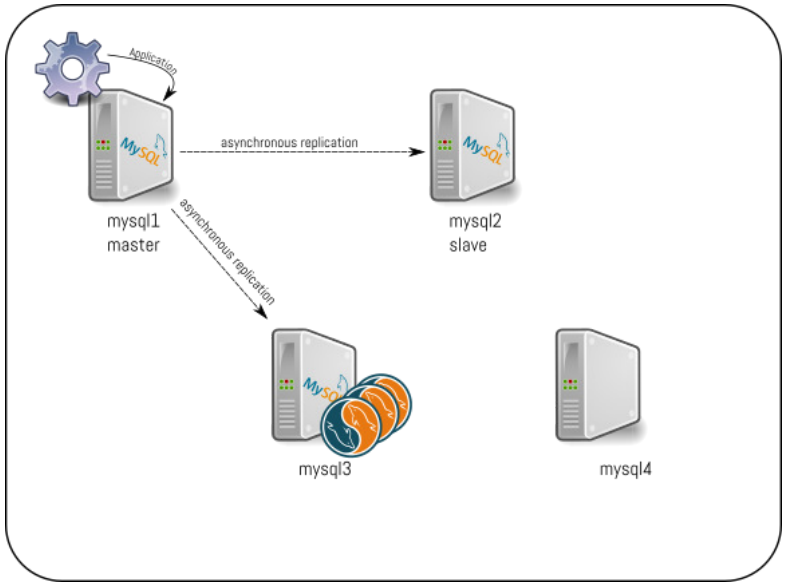
```
mysql-js> cluster.status()
{
    "clusterName": "MyInnoDBCluster",
    "defaultReplicaSet": {
        "name": "default",
        "primary": "mysql3:3306",
        "status": "OK_NO_TOLERANCE",
        "statusText": "Cluster is NOT tolerant to any failures.",
        "topology": {
            "mysql3:3306": {
                "address": "mysql3:3306",
                "mode": "R/W",
                "readReplicas": {},
                "role": "HA",
                "status": "ONLINE"
            }
        }
    }
}
```

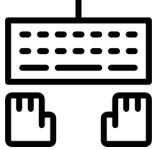


LAB5: add mysql4 to the cluster (1)

Add `mysql4` to the Group:

- restore the backup
- set the purged GTIDs
- use MySQL shell





LAB5: add mysql4 to the cluster (2)

Copy the backup from `mysql1` to `mysql4`:

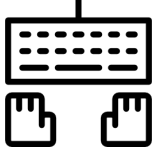
```
[mysql1 ~]# scp -r /tmp/backup mysql4:/tmp
```

And restore it:

```
[mysql4 ~]# mysqlbackup --backup-dir=/tmp/backup --force copy-back
[mysql4 ~]# rm /var/lib/mysql/mysql*-bin*     # just some cleanup
[mysql4 ~]# chown -R mysql. /var/lib/mysql
```

Start MySQL on `mysql4`:

```
[mysql4 ~]# systemctl start mysqld
```



LAB5: MySQL shell to add an instance (3)

```
[mysql4 ~]# mysqlsh
```

Let´s verify the config:

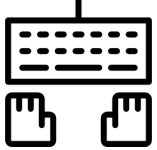
```
mysql-js> dba.checkInstanceConfiguration('root@mysql4:3306')
```

And change the configuration:

```
mysql-js> dba.configureLocalInstance()
```

Restart the service to enable the changes:

```
[mysql4 ~]# systemctl restart mysqld
```



LAB5: MySQL InnoDB Cluster (4)

Group of 2 instances

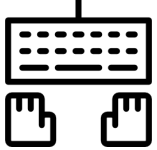
Find the latest purged GTIDs:

```
[mysql4 ~]# cat /tmp/backup/meta/backup_gtid_executed.sql
  SET @@GLOBAL.GTID_PURGED='33351000-3fe8-11e7-80b3-08002718d305:1-1002';
...
```

Connect to `mysql4` and set `GTID_PURGED`

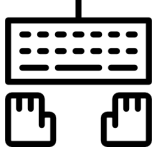
```
[mysql4 ~]# mysqlsh
```

```
mysql-js> \c root@mysql4:3306
mysql-js> \sql
mysql-sql> RESET MASTER;
mysql-sql> SET global gtid_purged="VALUE FOUND PREVIOUSLY";
```



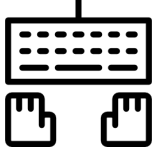
LAB5: MySQL InnoDB Cluster (5)

```
mysql-sql> \js

mysql-js> dba.checkInstanceConfiguration('root@mysql4:3306')

mysql-js> \c root@mysql3:3306

mysql-js> cluster = dba.getCluster()

mysql-js> cluster.checkInstanceState('root@mysql4:3306')

mysql-js> cluster.addInstance("root@mysql4:3306")

mysql-js> cluster.status()
```



Cluster Status

```
mysql-js> cluster.status()
{
    "clusterName": "MyInnoDBCluster",
    "defaultReplicaSet": {
        "status": "Cluster is NOT tolerant to any failures.",
        "topology": {
            "mysql3:3306": {
                "address": "mysql3:3306",
                "status": "ONLINE",
                "role": "HA",
                "mode": "R/W",
                "leaves": {
                    "mysql4:3306": {
                        "address": "mysql4:3306",
                        "status": "RECOVERING",
                        "role": "HA",
                        "mode": "R/O",
                        "leaves": {}
                    }
                }
            }
        }
    }
}
```

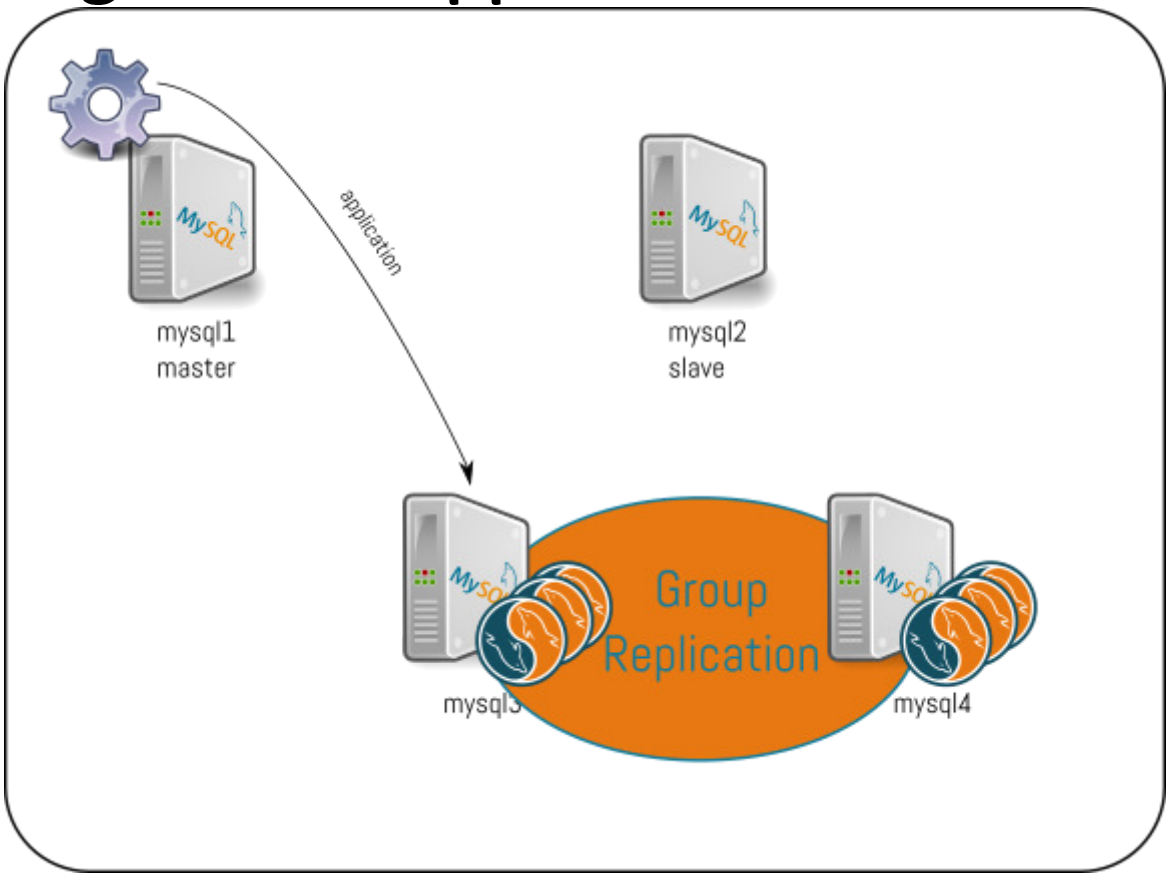


Recovering progress

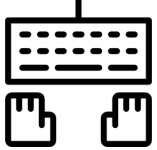
On standard MySQL, monitor the `group_replication_recovery` channel to see the progress:

```
mysql> show slave status for channel 'group_replication_recovery'\G
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: mysql3
                  Master_User: mysql_innodb_cluster_rpl_user
                             ...
              Slave_IO_Running: Yes
             Slave_SQL_Running: Yes
                             ...
           Retrieved_Gtid_Set: 6e7d7848-860f-11e6-92e4-08002718d305:1-6,
7c1f0c2d-860d-11e6-9df7-08002718d305:1-15,
b346474c-8601-11e6-9b39-08002718d305:1964-77177,
e8c524df-860d-11e6-9df7-08002718d305:1-2
            Executed_Gtid_Set: 7c1f0c2d-860d-11e6-9df7-08002718d305:1-7,
b346474c-8601-11e6-9b39-08002718d305:1-45408,
e8c524df-860d-11e6-9df7-08002718d305:1-2
                             ...
```

Migrate the application



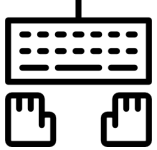
point the application
to the cluster



LAB6: Migrate the application

Now we need to point the application to `mysql3`, this is the only downtime !

```
...
[  21257s] threads: 4, tps: 12.00, reads: 167.94, writes: 47.98, response time: 1
[  21258s] threads: 4, tps: 6.00,  reads: 83.96, writes:  23.99, response time: 1
[  21259s] threads: 4, tps: 7.00,  reads: 98.05, writes:  28.01, response time: 1
[  31250s] threads: 4, tps: 8.00,  reads: 111.95, writes: 31.99, response time: 3
[  31251s] threads: 4, tps: 11.00, reads: 154.01, writes: 44.00, response time: 1
[  31252s] threads: 4, tps: 11.00, reads: 153.94, writes: 43.98, response time: 1
[  31253s] threads: 4, tps: 10.01, reads: 140.07, writes: 40.02, response time: 1
^C
[mysql1 ~]# run_app.sh mysql3
```



LAB6: Migrate the application

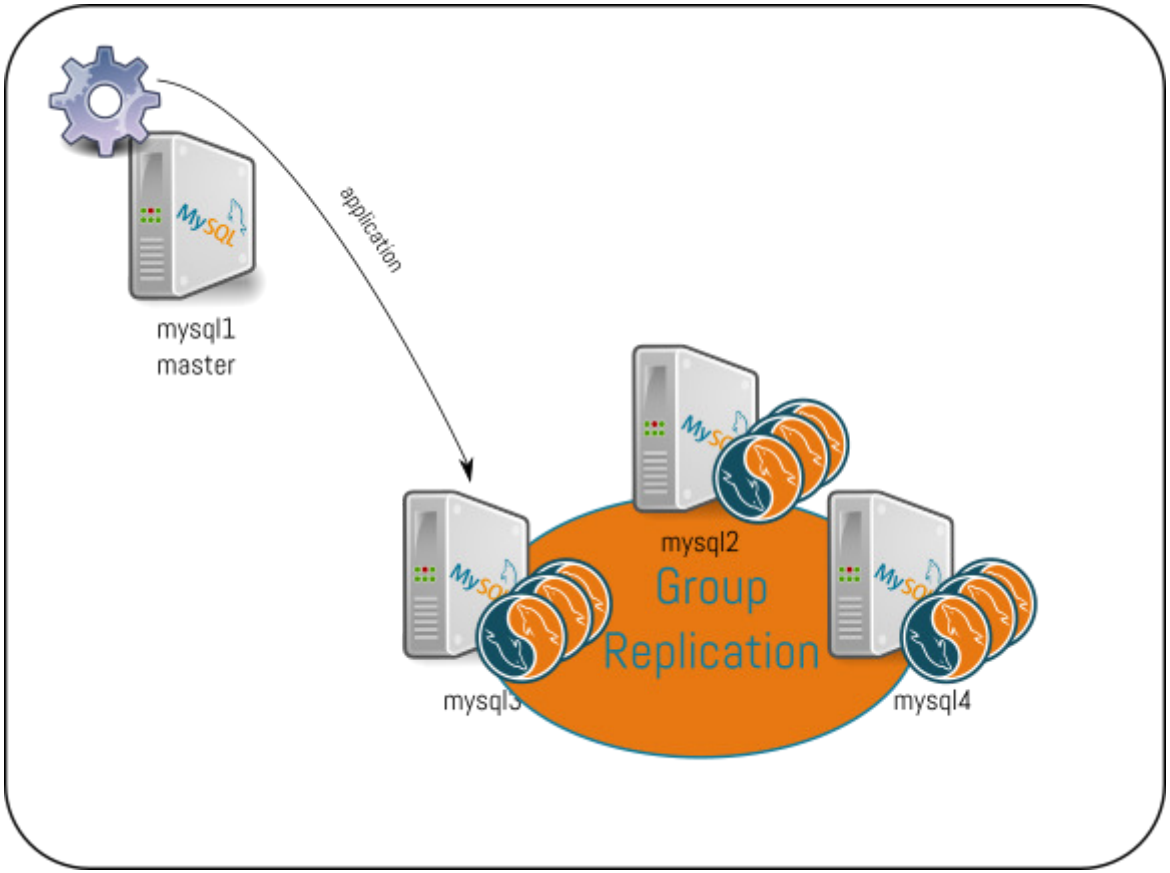
Stop asynchronous replication on `mysql2` and `mysql3`:

```
mysql2> stop slave;
mysql3> stop slave;
```

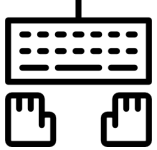
Make sure `gtid_executed` range on `mysql2` is lower or equal than on `mysql3`

```
mysql[2-3]> show global variables like 'gtid_executed'\G
mysql[2-3]> reset slave all;
```

Add a third instance



previous slave (`mysql2`) can now be part of the cluster



LAB7: Add mysql2 to the group

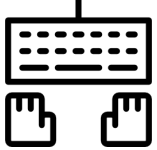
We first validate the instance using MySQL Shell and we configure it.

```
[mysql2 ~]# mysqlsh
```

```
mysql-js> dba.checkInstanceConfiguration('root@mysql2:3306')

mysql-js> dba.configureLocalInstance()
```

We also need to remove **super_read_only** from `my.cnf` to be able to use the shell to add the node to the cluster.

```
[mysql2 ~]# systemctl restart mysqld
```

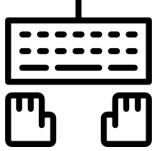


LAB7: Add mysql2 to the group (2)

Back in MySQL shell we add the new instance:

```
[mysql2 ~]# mysqlsh
```

```
mysql-js> dba.checkInstanceConfiguration('root@mysql2:3306')
mysql-js> \c root@mysql3:3306
mysql-js> cluster = dba.getCluster()
mysql-js> cluster.addInstance("root@mysql2:3306")
mysql-js> cluster.status()
```



LAB7: Add mysql2 to the group (3)


```
{
    "clusterName": "MyInnoDBCluster",
    "defaultReplicaSet": {
        "name": "default",
        "primary": "mysql3:3306",
        "status": "OK",
        "statusText": "Cluster is ONLINE and can tolerate up to ONE failure.",
        "topology": {
            "mysql2:3306": {
                "address": "mysql2:3306",
                "mode": "R/O",
                "readReplicas": {},
                "role": "HA",
                "status": "ONLINE"
            },
            "mysql3:3306": {
                "address": "mysql3:3306",
                "mode": "R/W",
                "readReplicas": {},
                "role": "HA",
                "status": "ONLINE"
            },
            "mysql4:3306": {
                "address": "mysql4:3306",
                "mode": "R/O",
                "readReplicas": {},
```

writing to a single server

Single Primary Mode

Default = Single Primary Mode

By default, MySQL InnoDB Cluster enables `Single Primary Mode`.

```
mysql> show global variables like 'group_replication_single_primary_mode';
+----------------------------------------+-------+
| Variable_name                          | Value |
+----------------------------------------+-------+
| group_replication_single_primary_mode | ON    |
+----------------------------------------+-------+
```

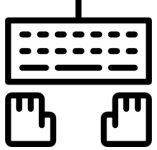
Default = Single Primary Mode

By default, MySQL InnoDB Cluster enables `Single Primary Mode`.

```
mysql> show global variables like 'group_replication_single_primary_mode';
+----------------------------------------+-------+
| Variable_name                          | Value |
+----------------------------------------+-------+
| group_replication_single_primary_mode | ON    |
+----------------------------------------+-------+
```

In Single Primary Mode, a single member acts as the writable master (`PRIMARY`) and the rest of the members act as hot-standbys (`SECONDARY`).

The group itself coordinates and configures itself automatically to determine which member will act as the `PRIMARY`, through a leader election mechanism.



Who´s the Primary Master ?

As the Primary Master is elected, all nodes part of the group knows which one was elected. This value is exposed in status variables:

```
mysql> show status like 'group_replication_primary_member';
+----------------------------------+--------------------------------------+
| Variable_name                    | Value                                |
+----------------------------------+--------------------------------------+
| group_replication_primary_member | 28a4e51f-860e-11e6-bdc4-08002718d305 |
+----------------------------------+--------------------------------------+

mysql> select member_host as "primary master"
       from performance_schema.global_status
       join  performance_schema.replication_group_members
       where variable_name = 'group_replication_primary_member'
       and member_id=variable_value;
+---------------+
| primary master|
+---------------+
| mysql3        |
+---------------+
```

Create a Multi-Primary Cluster:

It´s also possible to create a Multi-Primary Cluster using the Shell:

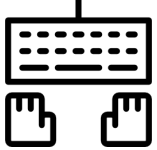
```
mysql-js> cluster=dba.createCluster('perconalive',{multiMaster: true})

A new InnoDB cluster will be created on instance 'root@mysql3:3306'.

The MySQL InnoDB cluster is going to be setup in advanced Multi-Master Mode.
Before continuing you have to confirm that you understand the requirements and
limitations of Multi-Master Mode. Please read the manual before proceeding.


I have read the MySQL InnoDB cluster manual and I understand the requirements
and limitations of advanced Multi-Master Mode.
Confirm [y|N]:
```

Or you can force it to avoid interaction (for automation) :

```
js> cluster=dba.createCluster('perconalive',{multiMaster: true, force: true})
```

get more info

Monitoring



Performance Schema

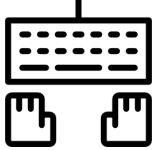
Group Replication uses `Performance_Schema` to expose status

```
mysql3> SELECT * FROM performance_schema.replication_group_members\G
*************************** 1. row ***************************
CHANNEL_NAME: group_replication_applier
   MEMBER_ID: 00db47c7-3e23-11e6-afd4-08002774c31b
 MEMBER_HOST: mysql3.localdomain
 MEMBER_PORT: 3306
MEMBER_STATE: ONLINE

mysql3> SELECT * FROM performance_schema.replication_connection_status\G
*************************** 1. row ***************************
             CHANNEL_NAME: group_replication_applier
               GROUP_NAME: afb80f36-2bff-11e6-84e0-0800277dd3bf
              SOURCE_UUID: afb80f36-2bff-11e6-84e0-0800277dd3bf
                THREAD_ID: NULL
            SERVICE_STATE: ON
COUNT_RECEIVED_HEARTBEATS: 0
 LAST_HEARTBEAT_TIMESTAMP: 0000-00-00 00:00:00
 RECEIVED_TRANSACTION_SET: afb80f36-2bff-11e6-84e0-0800277dd3bf:1-2
        LAST_ERROR_NUMBER: 0
       LAST_ERROR_MESSAGE:
     LAST_ERROR_TIMESTAMP: 0000-00-00 00:00:00
```

Member State

These are the different possible state for a node member:

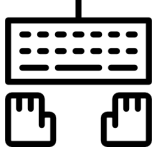
- `ONLINE`
- `OFFLINE`
- `RECOVERING`
- `ERROR`: when a node is leaving but the plugin was not instructed to stop
- `UNREACHABLE`



Status information & metrics

Members

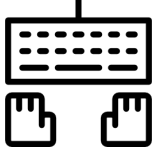
```
mysql>  SELECT * FROM performance_schema.replication_group_members\G

*************************** 1. row ***************************
CHANNEL_NAME: group_replication_applier
   MEMBER_ID: 00db47c7-3e23-11e6-afd4-08002774c31b
 MEMBER_HOST: mysql3.localdomain
 MEMBER_PORT: 3306
  MEMBER_STATE: ONLINE
*************************** 2. row ***************************
CHANNEL_NAME: group_replication_applier
   MEMBER_ID: e1544c9d-4451-11e6-9f5a-08002774c31b
 MEMBER_HOST: mysql4.localdomain.localdomain
 MEMBER_PORT: 3306
  MEMBER_STATE: ONLINE
```



Status information & metrics

Connections

```
mysql> SELECT * FROM performance_schema.replication_connection_status\G

*************************** 1. row ***************************
             CHANNEL_NAME: group_replication_applier
               GROUP_NAME: afb80f36-2bff-11e6-84e0-0800277dd3bf
              SOURCE_UUID: afb80f36-2bff-11e6-84e0-0800277dd3bf
                THREAD_ID: NULL
            SERVICE_STATE: ON
 COUNT_RECEIVED_HEARTBEATS: 0
  LAST_HEARTBEAT_TIMESTAMP: 0000-00-00 00:00:00
  RECEIVED_TRANSACTION_SET: 5de4400b-3dd7-11e6-8a71-08002774c31b:1-814089,
                            afb80f36-2bff-11e6-84e0-0800277dd3bf:1-2834
         LAST_ERROR_NUMBER: 0
        LAST_ERROR_MESSAGE:
      LAST_ERROR_TIMESTAMP: 0000-00-00 00:00:00
*************************** 2. row ***************************
             CHANNEL_NAME: group_replication_recovery
               GROUP_NAME:
              SOURCE_UUID:
                THREAD_ID: NULL
            SERVICE_STATE: OFF
 CO                 ARTBEATS: 0
```



Status information & metrics

Local node status

```
mysql> select * from performance_schema.replication_group_member_stats\G

*************************** 1. row ***************************
                      CHANNEL_NAME: group_replication_applier
                           VIEW_ID: 14679667214442885:4
                         MEMBER_ID: e1544c9d-4451-11e6-9f5a-08002774c31b
       COUNT_TRANSACTIONS_IN_QUEUE: 0
        COUNT_TRANSACTIONS_CHECKED: 5961
          COUNT_CONFLICTS_DETECTED: 0
COUNT_TRANSACTIONS_ROWS_VALIDATING: 0
TRANSACTIONS_COMMITTED_ALL_MEMBERS: 5de4400b-3dd7-11e6-8a71-08002774c31b:1-81408
                                    afb80f36-2bff-11e6-84e0-0800277dd3bf:1-5718
    LAST_CONFLICT_FREE_TRANSACTION: afb80f36-2bff-11e6-84e0-0800277dd3bf:5718
```

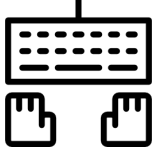
Performance_Schema

You can find GR information in the following **Performance_Schema** tables:

- `replication_applier_configuration`
- `replication_applier_status`
- `replication_applier_status_by_worker`
- `replication_connection_configuration`
- `replication_connection_status`
- `replication_group_member_stats`
- `replication_group_members`

Status during recovery

```
mysql> SHOW SLAVE STATUS FOR CHANNEL 'group_replication_recovery'\G

*************************** 1. row ***************************
               Slave_IO_State:
                  Master_Host: <NULL>
                  Master_User: gr_repl
                  Master_Port: 0
                        ...
               Relay_Log_File: mysql4-relay-bin-group_replication_recovery.00000
                        ...
            Slave_IO_Running: No
           Slave_SQL_Running: No
                        ...
           Executed_Gtid_Set: 5de4400b-3dd7-11e6-8a71-08002774c31b:1-814089,
afb80f36-2bff-11e6-84e0-0800277dd3bf:1-5718
             ...
                Channel_Name: group_replication_recovery
```



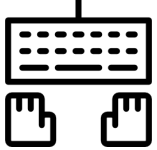
Sys Schema

The easiest way to detect if a node is a member of the primary component (when there are partitioning of your nodes due to network issues for example) and therefore a valid candidate for routing queries to it, is to use the sys table.

Additional information for `sys` can be downloaded at https://github.com/lefred/mysql_gr_routing_check/blob/master/addition_to_sys.sql

On the primary node:

```
[mysql? ~]# mysql < /tmp/addition_to_sys.sql
```



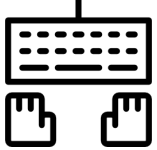
Sys Schema

Is this node part of PRIMARY Partition:

```
mysql3> SELECT sys.gr_member_in_primary_partition();
+------------------------------------+
| sys.gr_node_in_primary_partition() |
+------------------------------------+
| YES                                |
+------------------------------------+
```

To use as healthcheck:

```
mysql3> SELECT * FROM sys.gr_member_routing_candidate_status;
+------------------+-----------+---------------------+----------------------+
| viable_candidate | read_only | transactions_behind | transactions_to_cert |
+------------------+-----------+---------------------+----------------------+
| YES              | YES       | 0                   | 0                    |
+------------------+-----------+---------------------+----------------------+
```



LAB8: Sys Schema - Health Check

On one of the non Primary nodes, run the following command:

```
mysql-sql> flush tables with read lock;
```

Now you can verify what the healthcheck exposes to you:

```
mysql-sql> SELECT * FROM sys.gr_member_routing_candidate_status;
+------------------+-----------+---------------------+----------------------+
| viable_candidate | read_only | transactions_behind | transactions_to_cert |
+------------------+-----------+---------------------+----------------------+
| YES              | YES       |                 950 |                    0 |
+------------------+-----------+---------------------+----------------------+
```

```
mysql-sql> UNLOCK TABLES;
```

application interaction

MySQL Router

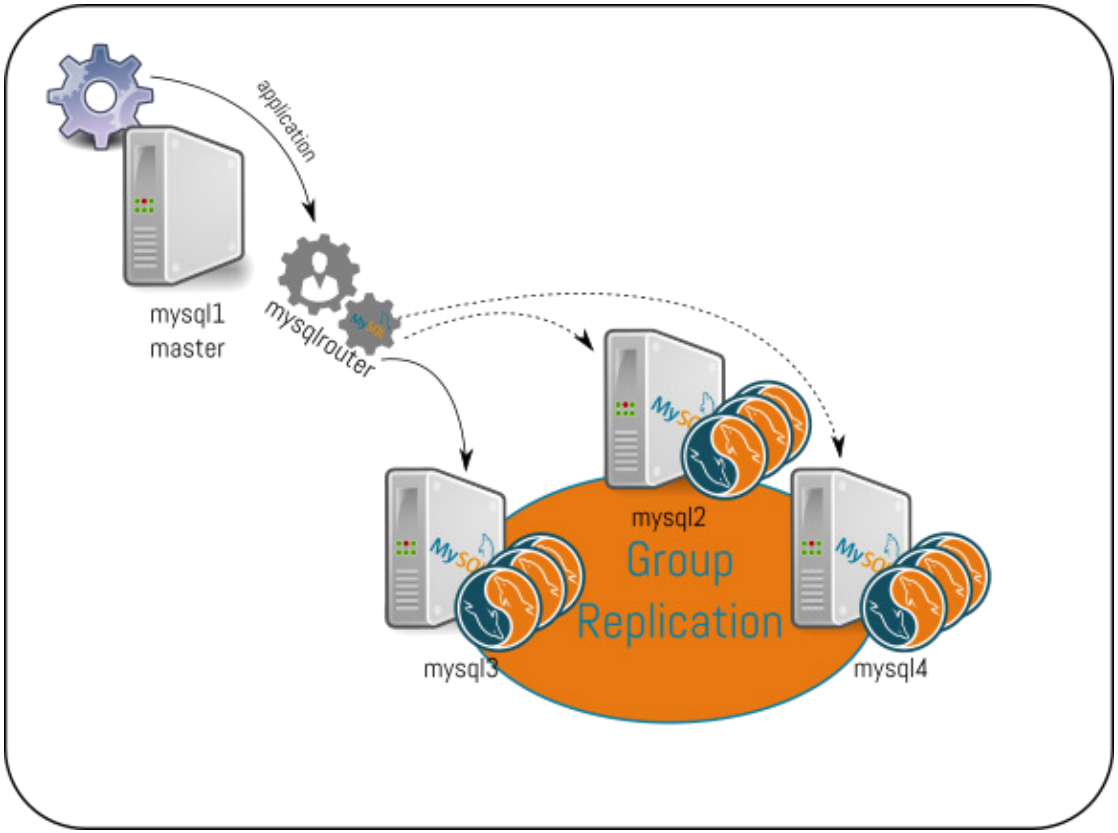
MySQL Router

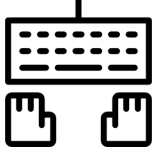
MySQL Router is lightweight middleware that provides transparent routing between your application and backend MySQL Servers. It can be used for a wide variety of use cases, such as providing high availability and scalability by effectively routing database traffic to appropriate backend MySQL Servers.

MySQL Router doesn´t require any specific configuration. It configures itself automatically (bootstrap) using MySQL InnoDB Cluster´s metadata.

LAB9: MySQL Router

We will now use `mysqlrouter` between our application and the cluster.

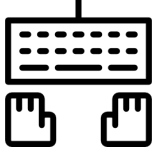




LAB9: MySQL Router (2)

Configure MySQL Router that will run on the app server (`mysql1`). We bootstrap it using the **Primary-Master**:

```
[root@mysql1 ~]# mysqlrouter --bootstrap mysql3:3306 --user mysqlrouter
Please enter MySQL password for root:
WARNING: The MySQL server does not have SSL configured and metadata used by
         the router may be transmitted unencrypted.

Bootstrapping system MySQL Router instance...
MySQL Router  has now been configured for the InnoDB cluster 'MyInnoDBCluster'.

The following connection information can be used to connect to the cluster.

Classic MySQL protocol connections to cluster 'MyInnoDBCluster':
- Read/Write Connections: localhost:6446
- Read/Only Connections: localhost:6447

X protocol connections to cluster 'MyInnoDBCluster':
- Read/Write Connections: localhost:64460
- Read/Only Connections: localhost:64470
```



LAB9: MySQL Router (3)

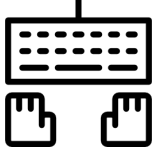
Now let´s modify the configuration file to listen on port 3306:

in `/etc/mysqlrouter/mysqlrouter.conf`:

```
[routing:MyInnoDBCluster_default_rw]
-bind_port=6446
+bind_port=3306
```

We can stop `mysqld` on `mysql1` and start `mysqlrouter` into a **screen** session:

```
[mysql1 ~]# systemctl stop mysqld
[mysql1 ~]# systemctl start mysqlrouter
```



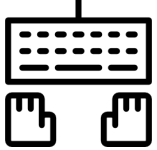
LAB9: MySQL Router (4)

Before killing a member we will change `systemd`'s default behavior that restarts `mysqld` immediately:

in `/usr/lib/systemd/system/mysqld.service` add the following under `[Service]`

```
RestartSec=30
```

```
[mysql3 ~]# systemctl daemon-reload
```



LAB9: MySQL Router (5)

Now we can point the application to the router (back to mysql1):

```
[mysql1 ~]# run_app.sh
```

Check app and kill `mysqld` on **mysql3** (the Primary Master R/W node) !

```
[mysql3 ~]# kill -9 $(pidof mysqld)
```

```
mysql> select member_host as "primary" from performance_schema.global_status
       join  performance_schema.replication_group_members
       where variable_name = 'group_replication_primary_member'
       and member_id=variable_value;
+---------+
| primary |
+---------+
| mysql4  |
+---------+
```

ProxySQL / HA Proxy / F5 / ...

3rd party router/proxy

3rd party router/proxy

MySQL InnoDB Cluster can also work with third party router / proxy.

If you need some specific features that are not yet available in **MySQL Router**, like transparent R/W splitting, then you can use your software of choice.

The important part of such implementation is to use a good health check to verify if the MySQL server you plan to route the traffic is in a valid state.

MySQL Router implements that natively, and it´s very easy to deploy.

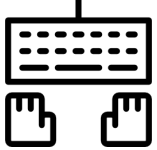
ProxySQL also has native support for Group Replication which makes it maybe the best choice for advanced users.

operational tasks

Recovering Node

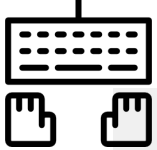
Recovering Nodes/Members

- The old master (`mysql3`) got killed.
- MySQL got restarted automatically by `systemd`
- Let´s add `mysql3` back to the clsuter



LAB10: Recovering Nodes/Members

```
[mysql3 ~]# mysqlsh

mysql-js> \c root@mysql4:3306                    # The current master

mysql-js> cluster = dba.getCluster()

mysql-js> cluster.status()

mysql-js> cluster.rejoinInstance("root@mysql3:3306")

Rejoining the instance to the InnoDB cluster. Depending on the original
problem that made the instance unavailable, the rejoin operation might not be
successful and further manual steps will be needed to fix the underlying
problem.

Please monitor the output of the rejoin operation and take necessary action if
the instance cannot rejoin.

Please provide the password for 'root@mysql3:3306':
Rejoining instance to the cluster ...

The instance 'root@mysql3:3306' was successfully rejoined on the cluster.

The instance 'mysql3:3306' was successfully added to the MySQL Cluster.
```

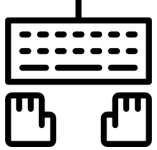


```
mysql-js> cluster.status()
{    "clusterName": "MyInnoDBCluster",
     "defaultReplicaSet": {
         "name": "default",
         "primary": "mysql4:3306",
         "status": "OK",
         "statusText": "Cluster is ONLINE and can tolerate up to ONE failure.",
         "topology": {
             "mysql2:3306": {
                 "address": "mysql2:3306",
                 "mode": "R/O",
                 "readReplicas": {},
                 "role": "HA",
                 "status": "ONLINE" },
             "mysql3:3306": {
                 "address": "mysql3:3306",
                 "mode": "R/O",
                 "readReplicas": {},
                 "role": "HA",
                 "status": "ONLINE" },
             "mysql4:3306": {
                 "address": "mysql4:3306",
                 "mode": "R/W",
                 "readReplicas": {},
                 "role": "HA",
                 "status": "ONLINE" }
         }
     }
}
```

Recovering Nodes/Members (automatically)

This time before killing a member of the group, we will **persist** the configuration on disk in *my.cnf*.

We will use again the same MySQL command as previously `dba.configureLocalInstance()` but this time when all nodes are already part of the Group.



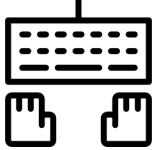
LAB10: Recovering Nodes/Members (2)

Verify that all nodes are `ONLINE`.

```
...
mysql-js> cluster.status()
```

Then on all nodes run:

```
mysql-js> dba.configureLocalInstance()
```



LAB10: Recovering Nodes/Members (3)

Kill one node again:

```
[mysql3 ~]# kill -9 $(pidof mysqld)
```

`systemd` will restart `mysqld` and verify if the node joined.

understanding

Flow Control

Flow Control

When using MySQL Group Replication, it´s possible that some members are lagging behind the group. Due to load, hardware limitation, etc... This lag can become problematic to keep good certification performance and keep the possible certification failures as low as possible.

More problems can occur in multi-primary/write clusters when the applying queue grows, the risk to have conflicts with those not yet applied transactions increases.

Flow Control (2)

Within MySQL Group Replication´s FC implementation :

- the Group is never totally stalled
- the node having issues doesn´t send flow control messages to the rest of the group asking to slow down

Flow Control (3)

Every member of the Group send some statistics about its queues (applier queue and certification queue) to the other members.

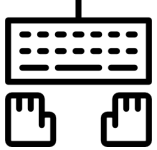
Then every node decide to slow down or not if they realize that one node reached the threshold for one of the queue:

- `group_replication_flow_control_applier_threshold` (default is 25000)
- `group_replication_flow_control_certifier_threshold` (default is 25000)

Flow Control (4)

So when `group_replication_flow_control_mode` is set to QUOTA on the node seeing that one of the other members of the cluster is lagging behind (threshold reached), it will throttle the write operations to the the minimum quota.

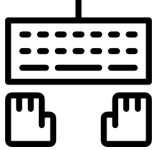
This quota is calculated based on the number of transactions applied in the last second, and then it is reduced below that by subtracting the *"over the quota"* messages from the last period.



LAB10: Flow Control

During this last lab, we will reduce the **flow control** threshold on the Primary Master:

```
mysql> show global variables like '%flow%';
+-----------------------------------------------------+-------+
| Variable_name                                       | Value |
+-----------------------------------------------------+-------+
| group_replication_flow_control_applier_threshold    | 25000 |
| group_replication_flow_control_certifier_threshold | 25000 |
| group_replication_flow_control_mode                 | QUOTA |
+-----------------------------------------------------+-------+
3 rows in set (0.08 sec)

mysql> set global group_replication_flow_control_applier_threshold=100;
```



LAB10: Flow Control (1)

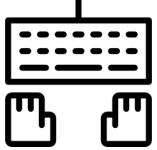
And now we block all writes on one of the Secondary-Masters:

```
mysql> flush tables with read lock;
```

And we check how the queue is growing:

```
mysql> SELECT * FROM sys.gr_member_routing_candidate_status;
+------------------+-----------+---------------------+----------------------+
| viable_candidate | read_only | transactions_behind | transactions_to_cert |
+------------------+-----------+---------------------+----------------------+
| YES              | NO        |                 487 |                    0 |
+------------------+-----------+---------------------+----------------------+
```

Did you notice something on the application when the threshold was reached ?



LAB10: Flow Control (2)

If nothing happened, please increase the trx rate:

```
[root@mysql1 ~]# run_app.sh mysql1 --tx-rate=500
```

When the application's writes are low, you can just remove the lock and see the queue and the effect on the application:

```
mysql> UNLOCK TABLES;
```

Create flow control again and when you see the application writing just a few transactions, on the Primary-Master, disable the flow control mode:

```
mysql> set global group_replication_flow_control_mode='DISABLED';
```

Thank you !

Any Questions ?