e: [info@mariadb.com](mailto:info@mariadb.com)

**High Availability Solutions for the MariaDB and MySQL® Database**

**1 Introduction**

This paper introduces recommendations and some of the solutions used to create an availability or

high availability environment for the MariaDB and MySQL® server. It also raises some questions to help you select the most feasible availability solution for your service, database environment and business critical infrastructure. We are using “MySQL server” as an alias here for the MySQL server and the MariaDB® server as we at MariaDB do support both – all statements in this whitepaper are valid for

the MariaDB server and the MySQL server. Let us begin with a brief general overview of some high

availability terminology.

**2 High Availability (HA)**

Wikipedia, a strong user of the MariaDB and MySQL database, is a good place to start when looking for definitions of all kind of terms. High availability is a system design approach and associated service implementation that ensures a prearranged level of operational performance will be met during a contractual measurement period.

Source:<http://en.wikipedia.org/wiki/High_availability>

The best way to define a level of high availability for a service, system, software or for the complete stack is to define the required uptime per year. This availability can be expressed as a percentage per year. We also often find the terms “one nine”, “two nines” up to “six nines” to define the availability. The following table illustrates this:

**Availability % Downtime per year Downtime per month (30**

**days)**

**Downtime per week**

**90% ("one nine")** 36.5 days 72 hours 16.8 hours

**95%** 18.25 days 36 hours 8.4 hours

**97%** 10.96 days 21.6 hours 5.04 hours

**98%** 7.30 days 14.4 hours 3.36 hours

**99% ("two nines")** 3.65 days 7.20 hours 1.68 hours

**99.50%** 1.83 days 3.60 hours 50.4 minutes

**99.80%** 17.52 hours 86.23 minutes 20.16 minutes

**99.9% ("three nines")** 8.76 hours 43.2 minutes 10.1 minutes

**99.95%** 4.38 hours 21.56 minutes 5.04 minutes

**99.99% ("four nines")** 52.56 minutes 4.32 minutes 1.01 minutes

**99.999% ("five nines")** 5.26 minutes 25.9 seconds 6.05 seconds

**99.9999% ("six nines")** 31.5 seconds 2.59 seconds 0.605 seconds Source: <http://en.wikipedia.org/wiki/High_availability>

To achieve high availability for a service it is not sufficient to find the right HA solution for every module running the service. For instance, Gartner published the document “NSM: Often the Weakest Link in Business Availability” in 2001, which includes a statement still worth quoting today:



File Name: High Availability Solutions for the MySQL® Database -­‐ MariaDB White Paper.docx

Page 1

*Gartner research shows that an average of 80 percent of mission-­‐critical application service downtime is directly caused by people or process failures. The other 20 percent is caused by technology failure, environmental failure or a disaster.*

What can we learn from this quote as we look for ways to make our MySQL server highly available? To start, we need to find the best solution for the requirements that we have. Second, we need to

select a solution that fits into our processes or current implementation, and every person involved in the maintenance of this service needs to know how it works as well, as what needs to be done.

**3 High Availability for Databases**

For every part of a complete service, application server, caching and database, a solution has to be found that achieves the required availability. For some elements this can be quite easy, especially if all the data is static or stateless. If the data is static, you just need multiple copies of the data. If the data

is stateless, however, you do not need to think about recovering it. Think of it as a web server that only serves static web pages or of session information, which is useless when a web server goes down.

As soon as a service requires data to be dynamic and/or stateful, the application/web server will use a database to store that data.

A database server is therefore a sensitive element for the whole service. Requirements concerning the server’s high availability are always high; oftentimes higher than really needed. This results in high costs, which in turn sets even higher expectations on the solution. In addition, this can sometimes mean a decrease in performance and features, as a high availability olution was selected which defines a high level of availability, but adds some limitations to the use of the database.

This whitepaper discusses a selection of HA solutions, which may initially look at odds.

However, you will eventually be able to recognize the recommended solution that best meets your requirements. Of course, this requires some specific user knowledge, but that is to be expected for these types of solutions.

Before we start to introduce some of the most commonly used HA solutions for MySQL, let us discuss a few general terms, which will be used to distinguish these different solutions throughout this whitepaper.

**3.1 Single Point of Failure (SPOF)**

We define an element as a single point of failure, when a failure of this element results in a full stop of the overall service, as no other element can take over. Not every HA solution will provide an environment without any SPOF andbut this needs to bedecided based on:

• The overall requirements

• The costs for eliminating this SPOF

• The likelihood that it fails

• The time it takes to bring it into service again…if we need to eliminate this single point of failure

A few examples of a SPOF can be:

• A storage

• A WAN connection

• A replication channel

**3.2 Downtime**

Downtime can be defined as the period of time a service is down. We need to distinguish between the overall service and the individual elements. Downtime can be unplanned and planned. Examples for unplanned downtime are:

• Software bug

• Hardware crash

• SPOF with failure

Planned downtime is often a result of:

• Hardware or software upgrades

• Architecture-­‐changes

• Reorganization

We need to be aware that the planned downtime is also part of the overall availability.

**3.3 Asynchronous vs. Synchronous**

To build a redundant database system, data needs to be moved to a second server. Two basic concepts are used when moving data to another server/storage: asynchronous and synchronous.

The data transfer is asynchronous when the server transferring the data does not wait for an

'acknowledge message' that the data was received before defining an action/transaction as successful. The advantage here is that a process does not depend on another process, which also results in better overall performance. This comes with the risk that:

• Data might not be in synch on both servers

• If a failure occurs on the second server, a rollback is not possible anymore. Servers get out of sync, this has to be resolved manually

These risks can be eliminated by using a synchronous data replication, where the second server needs to send an 'acknowledge message' to the primary node before this node can commit a transaction. In this case the overall performance depends not only on the performance of both nodes, but also the network throughput between both servers can be the bottleneck here. It is therefore not recommended to use a synchronous setup over long distances like in a WAN.

**3.4 Shared vs. Local Storage**

Shared storage systems like SANs can provide built-­‐in high availability, though this comes with equally high costs. We will introduce HA solutions using shared storage for an active/passive setup, as well as ways to build HA solutions based on local storage.

A shared storage approach typically comes with the advantage of lower data maintenance. Data does not need to be replicated between two locations. This may change for disaster scenarios though,

where data needs to be stored in two places over long distances. It can become very expensive to build a solution for this scenario only based on a shared storage system.

Using local storage comes with low cost, as standard hardware and local discs can be used. As data is distributed over different systems, data maintenance can be a bit more complicated. For building redundant systems we need to implement ways for replication/mirroring, but we can be more flexible as we can pick the right solution for the requirements that we have.

In this whitepaper we still use the term “local storage” if the network-­‐based storage used is dedicated to only one server.

**3.5 Switchover, Failover and Failback**

When a manual process is used to switch from one system to a redundant or standby system in case of a failure, we are talking about switchover. Automatic switchover, without human intervention, is called failover.

In some cases, a failover is not possible:

• Where there is no way to automate a process

• Where manual checks of the failure are required to define the right way to solve an issue

In a complex service environment, there are a lot of elements which are not a single point of failure and where a failure does nor necessarily influence the overall availability. An example of this could be a redundant application server. As long as other application servers can manage the load, there is no need to build a failover system which may allow a safe restart but may also create even more damage when restarted without any prior investigation.

An often-­‐underestimated task for a high availability solution is how to handle the recovery of a failed system and how to fail-­‐back to this system after recovery. This again depends heavily on whether the system solely relies on static or stateless data, or on whether data needs to be re-­‐synchronized. Often the main reason for requiring a fail-­‐back is that the primary system is more performant than the redundant one. We need to decide whether it is more expensive to build a stable fail-­‐back procedure

or whether to spend some money on the redundant system that can stay active as the primary system.

This allows us to securely set up the failed system as the new redundant system.

**4 HA-­‐Solutions for the MySQL Server**

There are many different solutions for building an availability, or even high availability, solution for the

MySQL server. These can be segmented into solutions based on:

• Mainly driven by the MySQL server features

• Storage based

• HA specialized to Storage Engines

• HA Middleware

**5 HA Solutions based on MySQL Server Features**

In this chapter, we will introduce HA solutions, which are based on a feature built into the MySQL server, MySQL Server Replication. MySQL Replication is a well-­‐known feature to implement read scalability or scalable solutions based on sharding.

In this paper, we want to discuss its capabilities for creating redundant systems for high availability. We will cover:

• MySQL server replication

• Semi-­‐synchronous replication

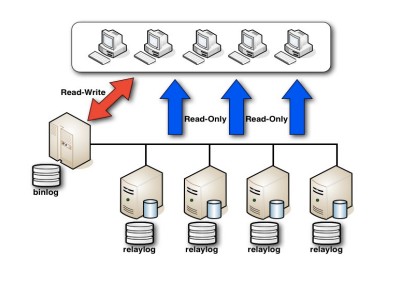
• Automatic failover using Master High Availability (MHA)

**5.1 MySQL Server Replication**

MySQL Server Replication is a strong feature, especially in the Online Applications sector, where read scalability implemented by using Replication is a widely used approach.

That said, MySQL Replication is not a high availability feature, per se. But as it is an easy way to setup a copy of a database server, or lets say, create redundancy, MySQL users also find it practical to use it as their first HA implementation for MySQL. And as so often, “never touch a running system”, it stays like this for a long time or even forever.

Let us have a brief look at how MySQL Replication works.



When binary logging is enabled for a MySQL server, we will give it the alias master. Data written to this master is also written to the binary log. Replicated servers, the slaves, collect that data into their own relay log via a so-­‐called I/O thread. The SQL thread on the slave reads the relay log and applies the writes on the slave.

NOTE: Standard MySQL Replication as of now is based on only two threads. Many connections (and therefore, threads) can be used by the master to change data. This can be a limitation for applications with a high number of writes.. However, one thread, the SQL thread, needs to be able to execute all the statements again on the slave (for SBR replications at least).

The method used for logging the written data can be statement-­‐based (SBR), row-­‐based (RBR) or mixed (MBR).

The whole process is asynchronous, which means that the master does not get a response from the slave to say that data is written. This is not that big of an issue for a scalability setup. But it can be for a HA as a fault on the master could result in a missing transaction on the slave, if the fault happens after the data is committed and written to the binary log, but not yet read by the slave. We can still read

that transaction from the binary log if the log file itself was not corrupted by the fault, but this is a

manual process which, for a HA solution, can take too long for the overall switch-­‐over.

**Standard Replication as provided with the MySQL database**

It is possible to define filters for the replication, for databases on the master, for tables on the slave. This is useful for scalability setups like sharding, but not for a HA solution where a slave should be the failover server for a master.

Let us have a look at how a switch-­‐over would be performed using MySQL Server Replication:

1. The master server is taken down or we encounter a fault by our monitoring

2. The slave server is updated to the last position in the relay log

3. The clients point at the designated slave server

4. The designated slave server becomes the master server

In Step 1, the MySQL server itself has not implemented any automatic procedure, all steps are manual ones. This implies that a well defined HA solution with MySQL Server Replication starts with a proper monitoring, at least for the master. MONyog Ultimate Monitor would be a good solution here.

For a planned switchover, such as when the master server needs some maintenance, you should ensure that the slave that will take over has received all events from the binary log before you shut down the master.

Step 2 is necessary to make sure that all transactions have been applied from the relay log before the slave is promoted to be the master and receive write requests from clients. This step can take some time for a write intensive application where the SQL thread cannot always apply changes in time.

NOTE: You can verify whether the slave received all transactions provided that you still have access to the master's binary log. Some manual work is needed if you found a transaction missing in the relay log that you don't want to loose.

Step 3 could also be implemented by using a virtual IP address, which will be routed to the current master.

In Step 4, the slaves become the master server. It is a wise idea to run a failover slave in read-­‐only mode to ensure that no data can be changed by a wrongly configured application. The setting must be changed in this step.

If other slaves have been connected to the failed master, binary log names and positions are not the same for the failed and the promoted master. You need to change the master info, bin log name and position for all the slaves.

• Asynchronous

• Local storage

• Hot standby

• Short switchover time

• Data loss possible

• All default storage engines

• Single I/O and SQL thread, delays possible

• Complex switch back

• Open Source

**Semi-­‐Synchronous Replication**

MySQL server version 5.5 and up also provides semi-­‐synchronous replication. This feature is implemented as a loadable module for master and server to enhance the MySQL Server Replication.

In the previous chapter we discussed the risk of loosing transactions due to the asynchronous implementation of MySQL Replication, which especially affects us for a HA solution. Semi-­‐synchronous replication addresses this by providing information on whether a slave is “semi-­‐synchron”.

One or more slaves can be defined as working semi-­‐synchronously. For these slaves, the master waits until the I/O thread on one or more of the semi-­‐synch slaves has flushed the transaction to disk. This ensures that all committed transactions are at least stored in the relay log of the slave. Transactions may not be applied to the slaves, but they are redundant in the binary log of the master and the relay log of the slave.

To avoid this, a master waits indefinitely to get the acknowledge from the slave and then changes to the standard replication mode after a defined timeout. The given parameter on the master should be monitored.

The steps for a failover are the same as when using the standard replication, with the slight difference that in Step 2, a slave should be chosen which shows itself to be semi-­‐synched with the master.

• Asynchronous

• Local storage

• All default storage engines

• Single I/O and SQL thread, delays possible

• Hot standby

• Short switchover time

• No data loss

• Complex switch back

• Open Source

**Automatic Failover with MHA**

We have learned that MySQL Replication allows us to run a redundant MySQL server and that we can switch to that server in case of a failure. However, we need to do this manually. We also know that some investigation time is needed to check for transactions, which have not been received by the slave.

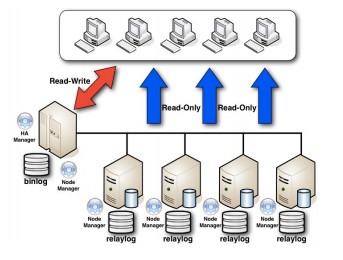
MariaDB offers support for MHA, a perl-­‐based solution, which addresses these points. It provides:

• Monitoring of master availability

• Automatic failover from a master to a slave

• utomation for the synchronization of the master's binary log to relay log, if it can still access the binary log.

• Synchronization of other slaves



MHA includes two main modules, MHA Manager and MHA Node. It has implemented some extension points, which can be used to integrate customized tasks. MHA Manager is monitoring the master or even a lot of different masters and controls the failover. It also allows to only control the failover and use pacemaker (for example) to detect a master failure and for virtual IP takeover. In case of a failover, MHA Mannager accesses MHA nodes via SSH to execute commands. MHA Manager should be

deployed on a separate server, which could also be the server running MONyog Ultimate Monitor, if it is used for multiple masters. For a master/slave pair, MHA Manager can also be installed on the slave. MHA Node is invoked by MHA Manager. MHA Node includes scripts to parse binary and relay logs, to apply events to other slaves etc. MHA runs on every node where MySQL Server is installed.

A typical failover using MHA would work like this:

1. Detect the failure of the master

2. Identify the slave with the most actual relay log

3. Apply missing events from the failed master's binary log to the identified slave, if possible

4. Recover other slaves to fix consistency issues

One big advantage of MHA is (as it is based on MySQL Replication) that it can be implemented into an existing MySQL server environment using replication without the need of configuration changes for

the MySQL server itself. Other HA solutions often require the use of a special storage engine or at least recommend one. MHA also takes advantage of the semi-­‐synchronous replication, if available.

• Asynchronous

• Read scalability

• Short failover time

• Local storage

• Hot standby

• Automatic failover

• Data loss possible without Semi-­‐synch replication

• All default storage engines

• Open Source

**6 Storage-­‐based Solutions**

Storage-­‐based environments offer the implementation of a HA solution based on the file system, which is used to store the data for MySQL server. In this chapter we will introduce:

• Synchronous Replication using DRBD®

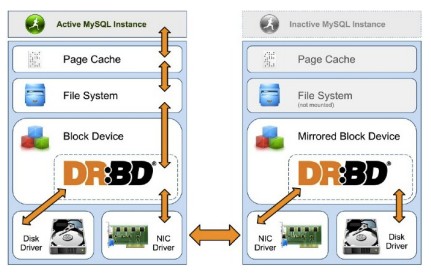
• Shared Storage Cluster

**6.1 Synchronous Replication using DRBD**

Another widely used HA solution in the MySQL Community is Distributed Replicated Block Device, better known as DRBD. This solution provides a synchronous way to form HA Clusters by mirroring a block device via an assigned network. DRBD also allows the use of asynchronous mirroring over long distances (WAN).

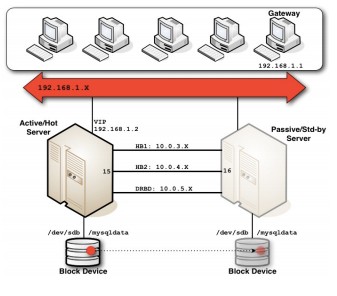
Linux Heartbeat is used for exchanging messages between cluster nodes and a cluster resource manager, typically Pacemaker, for starting and stopping a service. This allows for an automatic failover and also fail-­‐back.

The following picture shows two servers using the standard components of a Linux kernel. DRBD is located between file system and disk to capture changed blocks and mirror these via TCP/IP to a second server.



DRBD knows two stages, primary and secondary. The node of stage primary is the active node, which covers writes to file from an application. It transfers changed data to the block device and through TCP/IP to the secondary nodes. Secondary nodes work as passive nodes, which means that files or blocks will only be updated by DRBD. The disk can be a local one or a shared storage device.

A typical setup for MySQL server running with two nodes could look like this:



MySQL server is installed on two nodes, the active (primary) and the passive (secondary) node, but mysqld only runs on the active node.

Applications access the nodes using a virtual IP, DRBD has its own network channel to mirror the blocks and Heartbeat uses two channels for the communication between the nodes.

Write requests from applications will be handled by the MySQL server instance on the active node, in the same way as a standalone installation would handle it. DRBD takes care of the changes on MySQL server files and mirrors these changes to the secondary node.

In case of a failure of the primary node:

1. Heartbeat encounters a failure of the primary node

2. Secondary node promoted to primary node

3. MySQL server starts on new primary node

4. Storage engine auto-­‐recovery where possible (InnoDB®)

5. Check runs for other storage engines (MyISAM®)

6. Virtual IP-­‐Address takeover

Step 5 can be a critical one as it can take quite some time to run the check on big tables and, if the check encounters issues, there is manual work involved to fix tables. It is therefore recommended to only use storage engines providing auto-­‐recovery to decrease failover time and to eliminate manual tasks.

When the failed node is up again, DRBD runs an auto-­‐recovery and is available as a secondary node. Be careful not to start the MySQL Instance in a startup script for this node. Pacemaker should start the instance.

NOTE: DRBD is only supported on Linux®, and has been part of the Linux kernel since version 2.6.33.

• Synchronous

• Active/Passive

• Local storage

• Failover time depends on transactions to recover

• No data loss possible

• InnoDB, XtraDB (auto-­‐recovery storage engines) recommended

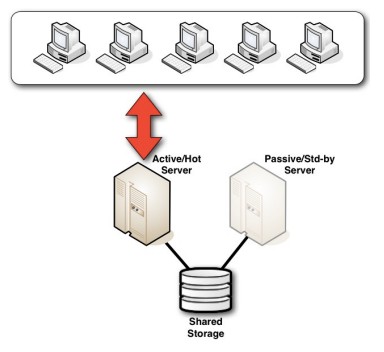
• Not for memory based tables

• Open Source

• No special cluster hardware needed

**6.2 Shared Storage Cluster**

Using a shared storage to store the data for a MySQL server is a similar HA solution to DRBD, but without the need to mirror data over the network. A shared storage is usually Shared Storage Area Networks (SAN) or Network Attached Storage (NAS). Data redundancy must be guaranteed by the shared storage.



Shared storage is typically used as an active/passive solution, in which applications connect to the active server and MySQL Server instance via a virtual IP address. Linux Heartbeat and Pacemaker can be used to create a failover solution. On other operating systems, OS clusters like Windows® cluster, can be used to monitor the behavior of the active system and to control and manage a failover to the passive node.

A clustered file system is not required. As for DRBD, it is recommended to use storage engines such as InnoDB or XtraDB®, since they provide a crash recovery, which starts as soon as the MySQL server instance on the passive node will be started in case a failure occurs.

Be aware that system tables are based on MyISAM; you should therefore run a check for these tables. These tables are small though and should not influence the failover time that much. The failover time for this setup also depends on the time the crash recovery may take and the time to unmount/mount the shared storage.

A big advantage of this HA solution is that a fail-­‐back is easily done by mounting the storage to the failed active node, when it is available again.

The failover should be followed by these steps:

1. Heartbeat encounters a failure of the active node

2. Stop MySQL server on the active node, if possible and needed

3. Unmount disk

4. Promote passive node to active node

5. Start MySQL server on new active node

6. Storage engine auto-­‐recovery where possible (InnoDB)

7. Check runs for other storage engines (MyISAM)

8. Virtual IP-­‐Address takeover

It is also possible to create an active/active cluster, but this comes with some limitations as the MySQL

instances would not share any information other than the files. Some of the limitations are:

• Only the storage engine MyISAM can be used, as MyISAM can work with external locking when enabled

• POSIX locking must be supported by the OS and file system

• Query cache and delayed key writes need to be turned off

• DDL needs to FLUSH TABLES

• Synchronous (no replication required)

• Active/Passive

• Active/Active with limitations

• Shared Storage

• Failover time depends on transactions to recover

• No data loss possible

• InnoDB, XtraDB (auto-­‐recovery storage engines) recommended

• Not for memory based tables

• Expensive storage system

• OS Cluster for failover

**7 HA Specialized for Storage Engines**

In this chapter, we introduce HA solutions which may require migrating to another storage engine. This may occur when the solution is only implemented based on one of the default storage engines or where the storage engine itself and related processes build the HA solution.

We have talked already about recommendations to only use storage engines, which provide crash recovery for storage based HA solutions. In that case, the HA solution does not really depend on a storage engine as a technical limitation -­‐ it is just a recommendation since using other storage engines might influence the recovery time too much.

**7.1 MySQL® Cluster**

MySQL® Cluster can be described as a HA solution providing highest availability, while using standard hardware. It fulfills the concept of removing any single point of failure by providing a shared-­‐nothing architecture. MySQL Cluster is fully transactional and was originally developed for high volumes of small transactions.

MySQL Cluster consists of three types of nodes:

• SQL Node

• Data Node

• Management Node

The SQL Node is MySQL server including the NDB/Cluster storage engine. The SQL node handles all the SQL requests, parsing, optimizing and query caching and is the standard way to connect from applications using the connectors available for MySQL server. Native APIs can used to directly connect applications and Data Nodes to remove the overhead of an SQL layer.

The data node (or at least two data nodes) build a data node group which stores the data. But data nodes are not only handling the storage of the data -­‐ they are also responsible for:

• Coordinating the transactions

• The record level locking

• Failover

• Fail-­‐back

• Implementing the two-­‐phase commit protocol to ensure that every change in the database is redundantly stored.

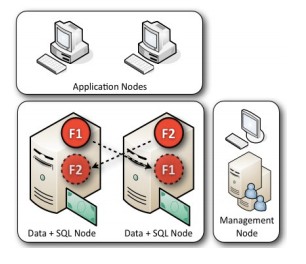
A data node therefore always needs to communicate with other data nodes and SQL nodes over the network, which requires low network latency. Spreading MySQL Cluster in two data centers connected in a WAN is therefore not recommended. A MySQL database can run with several data nodes, which allows distribution of the data for scalability and performance, while data is replicated for redundancy on up to four data nodes.

The management node is the node that provides the MySQL Cluster configuration and is used to manage the cluster; it works as an arbitrator for split-­‐brain situations.

MySQL Cluster is also the only solution which provides read and write scalability using multiple SQL Nodes for reads and writes.

The share-­‐nothing architecture allows full online operations for maintenance and administration. Rolling upgrades allow upgrading of a MySQL Cluster without downtime.

A minimal setup for MySQL Cluster, which provides full HA and a share-­‐nothing architecture, is shown in the following picture.



There are two Application Nodes, both with read and write access two SQL Nodes. If one Application Node has a failure (in this case for example a Web Server), the other Application Server is still available to execute reads and writes.

One Data and SQL Node is installed on the same physical server; another Data and SQL Node on a second one. F1 and F2 are cluster partitions handling the data. The server on the left actively operates on data stored in partition F1 and holds copies of F2; the server on the right is responsible for data in partition F2 and stores copies of F1. Partitions F1 and F2 put together build the full set of data stored for MySQL Cluster tables. If one of the servers has a failure, the other one will instantly take over the control for both partitions. As there is no crash recovery, mounting or start of a process needed, the failover is therefore transparent for the Application Nodes.

• Synchronous (no replication required)

• Active/Active (SQL Nodes)

• Local storage

• Failover time some seconds

• No data loss possible

• Only for Cluster(NDB) Storage Engine

• Commodity hardware

• Application Cluster

**8 HA Middleware**

**8.1 Tungsten® Replicator and Tungsten Enterprise**

Tungsten® Replicator is developed by Continuent. Tungsten Replicator is open source software, licensed under GPL V2. There is also Tungsten® Enterprise, which includes enhancements such as:

• Cluster Management

o Easy setup of master/slave clusters

o Automated failover and recovery

o Floating IP address management

o Backup integration

• Performance Scaling

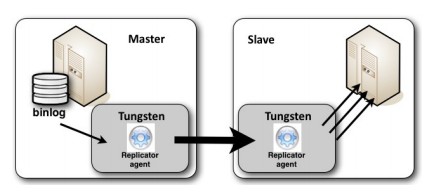
o SQL read/write splitting

o Transparent read load balancing

o Slave latency tracking

Continuent describes Tungsten Enterprise as being a complete replication and data management solution for the MySQL database. Tungsten Enterprise has implemented its own asynchronous replication by using external replicators instead of the internal MySQL replicator.

The master binary log is the source for the replicator processes used to transfer the changes to slaves. Such a replicator process exists on the master and slave host. The following graph illustrates the data flow:



One difference to the standard replication provided by the MySQL database is that changes can be applied to slaves with multiple threads sharded by the database. The parallel implementation can help significantly to reduce or even to eliminate slave lag problems, but only when multiple databases are used and are working independently of each other.

Tungsten Enterprise supports different kinds of replication setups like typical master/slave or master/master, but also multi-­‐source setups.

Tungsten Enterprise works with a global transaction ID when replicating between master and slaves. This global transaction ID is generated by the master. As mentioned earlier in this document, asynchronous replication can lead to binary logs, which are not in synch between master and slaves. Having a global transaction ID makes life easier in case of a failover to synch the missing transaction

between the master binary log (if still available) and the slaves, especially when a master is used with multiple slaves.

While Tungsten Replicator provides all the basic features for a storage engineindependent replication, Tungsten Enterprise is the product which should be used for building a high availability solution as it includes the features to:

• Get automated failover and recovery

• Use floating IP support

• Centrally manage the cluster

• Implement backup and restore procedures

• Synchronize missing transactions to slaves

• Asynchronous (Tungsten Replicator)

• Read scalability

• Short fail-­‐over time

• Local storage

• Hot standby

• Automatic Failover

• Data loss possible, but reduced

• Global Transaction IDs

• Per Schema multi-­‐threaded replication

• All default storage engines

• Truly multi-­‐master

• Partially Open Source

**9 Summary**

A good start to summarize this white paper and the different high availability solutions that we discussed is a comparison chart. The following chart predominantly compares:

• Data safeness

• The influence on the MySQL storage engine architecture

• Switch-­‐ or failover (manual or automatic)

• Performance

• Scalability

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **MySQL Replication** | **MHA** | **Tungsten** | **DRBD** | **Shared**  **Cluster** | **MySQL Cluster** |
| **100% Data** | ✘ | ✘ | ✘ | ✔ | ✔ | ✔ |
| **Safe** |  |  |  |  |  |  |
| **All Storage** | ✔ | ✔ | ✔ | ✘ | ✘ | ✘ |
| **Engines** |  |  |  |  |  |  |
| **Automatic** | ✘ | ✔ | ✔ | ✔ | ✘ | ✔ |
| **Failover** |  |  |  |  |  |  |
| **Performance** | \* | \* | \* | \*\*\* | -­‐ | \* |
| **Overhead (\* -­‐** |  |  |  |  |  |  |
| **Best)** |  |  |  |  |  |  |
| **Performance** | \*\* | \*\* | \* | \*\*\* | \* | \* |
| **Operations** |  |  |  |  |  |  |
| **(\*\*\* -­‐ Best)** |  |  |  |  |  |  |
| **Scalability** | \*\* | \*\* | \*\*\* | \* | \* | \*\*\* |
| **Solution** |  |  |  |  |  |  |
| **(\*\*\*-­‐ Best)** |  |  |  |  |  |  |

This whitepaper provides recommendations and guidelines on a number of high availability solutions for MySQL as well as on different approaches for implementing an availability solution for MySQL installations. We also recommend that you explore additional solutions such as SchoonerSQL®, Galera®, MMM or VM based HA.

• When evaluating a high availability solution it is important to define:

• Which level of High Availability is needed

• Whether an asynchronous approach can be used

o The impact that the loss of one transaction can have

• Whether automatic failover is needed or if a switchover could be enough

• Whether a reasonable service can be provided if a component is down

• Whether the SLAs for providing data for reads are the same as for changing data