Database Replication

There are times when it is useful to create multiple copies of a database. Not just a snapshot of a moment in time, but live, constantly updated copies.

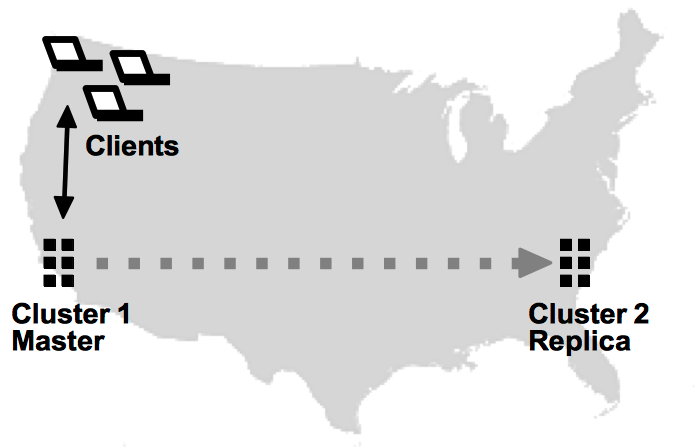
K-safety maintains redundant copies of partitions within a single VoltDB database, which helps protect the database cluster against individual node failure. Database replication also creates a copy. However, database replication creates and maintains copies in separate, often remote, databases.

VoltDB supports two forms of database replication:

* One-way (Passive)
* Two-way (Cross Datacenter)

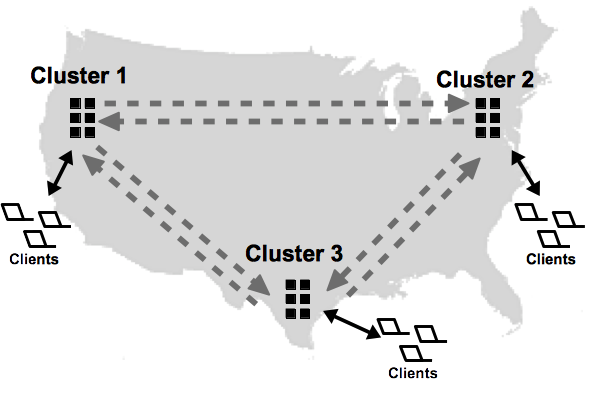
**Passive replication** copies the contents from one database, known as the master database, to the other, known as the replica. In passive replication, replication occurs in one direction: from the master to the replica. Clients can connect to the master database and perform all normal database operations, including INSERT, UPDATE, and DELETE statements. As shown in [Figure 11.1, “Passive Database Replication”](https://docs.voltdb.com/UsingVoltDB/ChapReplication.php#DbRepPassivefig) changes are copied from the master to the replica. To ensure consistency between the two databases, the replica is started as a read-only database, where only transactions replicated from the master can modify the database contents.

**Figure 11.1. Passive Database Replication**



**Cross Datacenter Replication (XDCR)**, or active replication, copies changes in both directions. XDCR can be set up on multiple clusters (not just two). Client applications can then perform read/write operations on any of the participating clusters and changes in one database are then copied and applied to all the other databases. [Figure 11.2, “Cross Datacenter Replication”](https://docs.voltdb.com/UsingVoltDB/ChapReplication.php#DbRepActivefig) shows how XDCR can support client applications attached to each database instance.

**Figure 11.2. Cross Datacenter Replication**



Database replication (DR) provides two key business advantages. The first is protecting your business data against catastrophic events, such as power outages or natural disasters, which could take down an entire cluster. This is often referred to as disaster recovery. Because the clusters can be in different geographic locations, both passive DR and XDCR allow other clusters to continue unaffected when one becomes inoperable. Because the replica is available for read-only transactions, passive DR also allows you to offload read-only workloads, such as reporting, from the main database instance.

The second business issue that DR addresses is the need to maintain separate, active copies of the database in separate locations. For example, XDCR allows you to maintain copies of a product inventory database at two or more separate warehouses, close to the applications that need the data. This feature makes it possible to support massive numbers of clients that could not be supported by a single database instance or might result in unacceptable latency when the database and the users are geographically separated. The databases can even reside on separate continents.

It is important to note, however, that database replication is not instantaneous. The transactions are committed locally, then copied to the other database or databases. So, when using XDCR to maintain multiple active clusters you must be careful to design your applications to avoid possible conflicts when transactions change the same record in two databases at approximately the same time. See [Section 11.3.7, “Understanding Conflict Resolution”](https://docs.voltdb.com/UsingVoltDB/DbRepHowToActive.php#DbRepActiveConflicts) for more information about conflict resolution.

The remainder of this chapter discusses the following topics:

* [Section 11.1, “How Database Replication Works”](https://docs.voltdb.com/UsingVoltDB/ChapReplication.php#DbRepOverview)
* [Section 11.2, “Using Passive Database Replication”](https://docs.voltdb.com/UsingVoltDB/DbRepHowToPassive.php)
* [Section 11.3, “Using Cross Datacenter Replication”](https://docs.voltdb.com/UsingVoltDB/DbRepHowToActive.php)
* [Section 11.4, “Monitoring Database Replication”](https://docs.voltdb.com/UsingVoltDB/DbRepManage.php)

## 11.1. How Database Replication Works

Database replication (DR) involves duplicating the contents of selected tables between two database clusters. In passive DR, the contents are copied in one direction: from master to replica. In active or cross datacenter DR, changes are copied in both directions.

You identify which tables to replicate in the schema, by specifying the table name in a DR TABLE statement. For example, to replicate all tables in the voter sample application, you would execute three DR TABLE statements when defining the database schema:

DR TABLE contestants;

DR TABLE votes;

DR TABLE area\_code\_state;

### **11.1.1. Starting Database Replication**

You enable DR by including the <dr> tag in the configuration files when initializing the database. The <dr> element identifies three pieces of information:

* A unique cluster ID for each database. The ID is required and can be any number between 0 and 127, as long as each cluster has a different ID.
* The role the cluster plays, whether master, replica, or xdcr. The default is master.
* For the replica and xdcr roles, a connection source listing the host name or IP address of one or more nodes from the other databases.

For example:

<dr id="2" role="replica">

<connection source="serverA1, serverA2" />

</dr>

Each cluster must have a unique ID. For passive DR, only the replica needs a <connection> element, since replication occurs in only one direction.

For cross datacenter replication (XDCR), all clusters must include the <connection> element pointing to at each one other cluster. If you are establishing an XDCR network with multiple clusters, the <connection> tag can specify hosts from one or more of the other clusters. The participating clusters will coordinate establishing the correct connections, even if the <connection> element does not list them all.

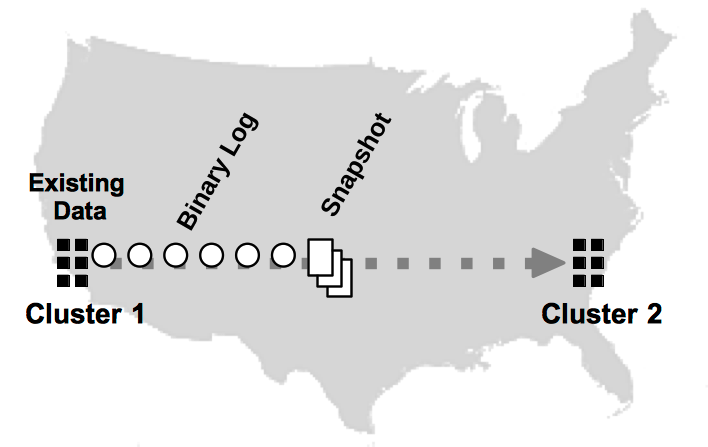
Note that for XDCR, you must specify the attribute role="xdcr" before starting each cluster. For passive DR, you must start the replica database with the role="replica" attribute to ensure the replica is in read-only mode. Once the clusters are configured properly and the schema of the DR tables match in the databases, replication starts.

The actual replication process is performed in multiple parallel streams; each unique partition on one cluster sends a binary log of completed transactions to the other clusters. Replicating by partition has two key advantages:

* The process is faster — Because the replication process uses a binary log of the results of the transaction (rather than the transaction itself), the receiving cluster (or consumer) does not need to reprocess the transaction; it simply applies the results. Also, since each partition replicates autonomously, multiple streams of data are processed in parallel, significantly increasing throughout.
* The process is more durable — In a K-safe environment, if a server fails on a DR cluster, individual partition streams can be redirected to other nodes or a stream can wait for the server to rejoin — without interfering with the replication of the other partitions.

If data already exists in one of the clusters before database replication starts for the first time, that database sends a snapshot of the existing data to the other, as shown in [Figure 11.3, “Replicating an Existing Database”](https://docs.voltdb.com/UsingVoltDB/ChapReplication.php#DbRepSnapshotfig). Once the snapshot is received and applied (and the two clusters are in sync), the partitions start sending binary logs of transaction results to keep the clusters synchronized.

**Figure 11.3. Replicating an Existing Database**



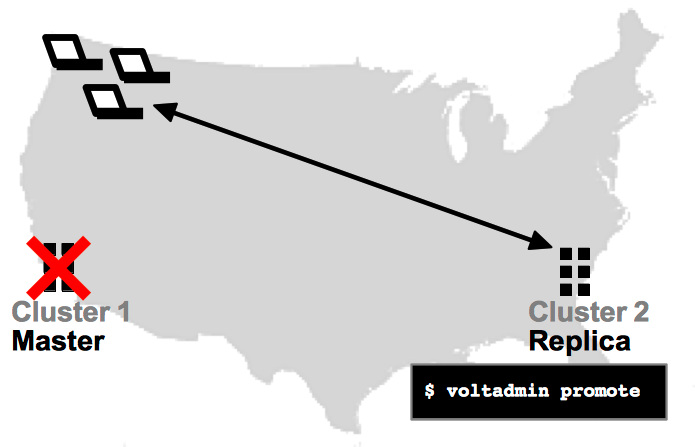
For passive DR, only the master database can have existing data before starting replication for the first time. The replica's DR tables must be empty. For XDCR, the first database that is started can have data in the DR tables. If other clusters contain data, replication cannot start. Once DR has started, the databases can stop and recover using command logging without having to restart DR from the beginning.

### **11.1.2. Database Replication, Availability, and Disaster Recovery**

Once replication begins, the DR process is designed to withstand normal failures and operational downtime. When using K-safety, if a node fails on any cluster, you can rejoin the node (or a replacement) using the **voltdb start** command without breaking replication. Similarly, if a cluster shuts down, you can use **voltdb start** to restart the database and restart replication where it left off. The ability to restart DR assumes you are using command logging. Specifically, synchronous command logging is recommended to ensure complete durability.

If unforeseen events occur that make a database unreachable, database replication lets you replace the missing database with its copy. This process is known as disaster recovery. For cross datacenter replication (XDCR), you simply need to redirect your client applications to the remaining cluster(s). For passive DR, there is an extra step. To replace the master database with the replica, you must issue the **voltadmin promote** command on the replica to switch it from read-only mode to a fully operational database.

**Figure 11.4. Promoting the Replica**



See [Section 11.2.6.3, “Promoting the Replica When the Master Becomes Unavailable”](https://docs.voltdb.com/UsingVoltDB/DbRepHowToPassive.php#DbRepHowToPromote) for more information on promoting the replica database.

### **11.1.3. Database Replication and Completeness**

It is important to note that, unlike K-safety where multiple copies of each partition are updated simultaneously, database replication involves shipping the results of completed transactions from one database to another. Because replication happens after the fact, there is no guarantee that the contents of the clusters are identical at any given point in time. Instead, the receiving database (or consumer) "catches up" with the sending database (or producer) after the binary logs are received and applied by each partition.

Also, because DR occurs on a per partition basis, changes to partitions may not occur in the same order on the consumer, since one partition may replicate faster than another. Normally this is not a problem because the results of all transactions are atomic in the binary log. However, if the producer cluster crashes, there is no guarantee that the consumer has managed to retrieve all the logs that were queued. Therefore, it is possible that some transactions that completed on the producer are not reflected on the consumer.

Fortunately, using command logging, when you restart the failed cluster, any unacknowledged transactions will be replayed from the failed cluster's disk-based DR cache, allowing the clusters to recover and resume DR where they left off. However, if the failed cluster does not recover, you will need to decide how to proceed. You can choose to restart DR from scratch or, if you are using passive DR, you can promote the replica to replace the master.

To ensure effective recovery, the use of synchronous command logging is recommended for DR. Synchronous command logging guarantees that all transactions are recorded in the command log and no transactions are lost. If you use asynchronous command logging, there is a possibility that a binary log is applied but not captured by the command log before the cluster crashes. Then when the database recovers, the clusters will not agree on the last acknowledged DR transaction, and DR will not be able to resume.

The decision whether to promote the replica or wait for the master to return (and hopefully recover all transactions from the command log) is not an easy one. Promoting the replica and using it to replace the original master may involve losing one or more transactions per partition. However, if the master cannot be recovered or cannot not be recovered quickly, waiting for the master to return can result in significant business loss or interruption.

Your own business requirements and the specific situation that caused the outage will determine which choice to make — whether to wait for the failed cluster to recover or to continue operations on the remaining cluster only. The important point is that database replication makes the choice possible and significantly eases the dangers of unforeseen events.

<https://docs.voltdb.com/UsingVoltDB/ChapReplication.php>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**How does MySQL Replication really work?**

While we do have many blog posts on replication on our blog, such as on [replication being single-threaded](https://www.percona.com/blog/2010/10/20/mysql-limitations-part-1-single-threaded-replication/), on [semi-synchronous replication](https://www.percona.com/blog/2012/01/19/how-does-semisynchronous-mysql-replication-work/) or on [estimating replication capacity](https://www.percona.com/blog/2010/07/20/estimating-replication-capacity/), I don’t think we have one that covers the very basics of how MySQL replication really works on the high level. Or it’s been so long ago I can’t even find it. So, I decided to write one now.

Of course, there are many aspects of MySQL replication, but my main focus will be the logistics – how replication events are written on the master, how they are transferred to the replication slave and then how they are applied there. Note that this is NOT a HOWTO setup replication, but rather a [howstuffworks](http://www.howstuffworks.com/" \t "_blank" \o "Learn How Everything Works) type of thing.

## Replication events

I say replication events in this article because I want to avoid discussion about different replication formats. These are covered pretty well [in the MySQL manual here](http://dev.mysql.com/doc/refman/5.5/en/replication-formats.html). Put simply, the events can be one of two types:

* **Statement based** – in which case these are write queries
* **Row based** – in this case these are changes to records, sort of row diffs if you will

But other than that, I won’t be going back to differences in replication with different replication formats, mostly because there’s very little that’s different when it comes to transporting the data changes.

## On the master

So now let me start with what is happening on the master. For replication to work, first of all master needs to be writing replication events to a special log called **binary log**. This is usually very lightweight activity (assuming events are not synchronized to disk), because writes are buffered and because they are sequential. The binary log file stores data that replication slave will be reading later.

Whenever a replication slave connects to a master, master creates a new thread for the connection (similar to one that’s used for just about any other server client) and then it does whatever the client – replication slave in this case – asks. Most of that is going to be (a) feeding replication slave with events from the binary log and (b) notifying slave about newly written events to its binary log.

Slaves that are up to date will mostly be reading events that are still cached in OS cache on the master, so there is not going to be any physical disk reads on the master in order to feed binary log events to slave(s). However, when you connect a replication slave that is few hours or even days behind, it will initially start reading binary logs that were written hours or days ago – master may no longer have these cached, so disk reads will occur. If master does not have free IO resources, you may feel a bump at that point.

## On the replica

Now let’s see what is happening on the slave. When you start replication, **two threads** are started on the slave:

**1. IO thread**

This process called IO thread connects to a master, reads binary log events from the master as they come in and just copies them over to a local log file called **relay log**. That’s all.

Even though there’s only one thread reading binary log from the master and one writing relay log on the slave, very rarely copying of replication events is a slower element of the replication. There could be a network delay, causing a steady delay of few hundred milliseconds, but that’s about it.

If you want to see where IO thread currently is, check the following in “show slave status\G”:

* **Master\_Log\_File** – last file copied from the master (most of the time it would be the same as last binary log written by a master)
* **Read\_Master\_Log\_Pos** – binary log from master is copied over to the relay log on the slave up until this position.

And then you can compare it to the output of “show master status\G” from the master.

**2. SQL thread**

The second process – SQL thread – reads events from a relay log stored locally on the replication slave (the file that was written by IO thread) and then applies them as fast as possible.

This thread is what people often blame for being single-threaded. Going back to “show slave status\G”, you can get the current status of SQL thread from the following variables:

* **Relay\_Master\_Log\_File** – binary log from master, that SQL thread is “working on” (in reality it is working on relay log, so it’s just a convenient way to display information)
* **Exec\_Master\_Log\_Pos** – which position from master binary log is being executed by SQL thread.

**Replication lag**

Now I want to briefly touch the subject of replication lag in this context. When you are dealing with replication lag, first thing you want to know is which of the two replication threads is behind. Most of the time it will be the SQL thread, still it makes sense to double check. You can do that by comparing the replication status variables mentioned above to the master binary log status from the output of “show master status\G” from the master.

If it happens to be IO thread, which, as I mentioned many times already, is very rare, one thing you may want to try to get that fixed is [enabling slave compressed protocol](http://dev.mysql.com/doc/refman/5.5/en/replication-options-slave.html#sysvar_slave_compressed_protocol).

Otherwise, if you are sure it is SQL thread, then you want to understand what is the reason and that you can usually observe by vmstat. Monitor server activity over time and see if it is “r” or “b” column that is “scoring” most of the time. If it is “r”, replication is CPU-bound, otherwise – IO. If it is not conclusive, mpstat will give you better visibility by CPU thread.

Note this assumes that there is no other activity happening on the server. If there is some activity, then you may also want to look at [diskstats](https://www.percona.com/doc/percona-toolkit/2.1/pt-diskstats.html" \t "_blank" \o "pt-diskstats - best tool to analyze disk stats) or even do a query review for SQL thread to get a good picture.

If you find that replication is CPU bound, [this maybe very helpful](https://www.percona.com/blog/2008/09/22/fighting-mysql-replication-lag/).

If it is IO bound, then fixing it may not be as easy (or rather, as cheap). Let me explain. If replication is IO bound, most of the time that means that SQL thread is unable to read fast enough because reads are single threaded. Yes, you got that right – it is **reads** that are limiting replication performance, not writes. Let me explain this further.

Assume you have a RAID10 with a bunch of disks and write-back cache. **Writes**, even though they are serialized, will be fast because they are buffered in the controller cache and because internally RAID card can parallelize writes to disks. Hence replication slave with similar hardware can write just as fast as master can.

Now **Reads**. When your workset does not fit in memory, then the data that is about to get modified is going to have to be read from disk first and this is where it is limited by the single-threaded nature of the replication, because one thread will only ever read from one disk at a time.

That being said, one solution to fix IO-bound replication is to increase the amount of memory so working set fits in memory. Another – get IO device that can do much more IO operations per second even with a single thread – fastest traditional disks can do up to 250 iops, SSDs – in the order of 10,000 iops

<https://www.percona.com/blog/2013/01/09/how-does-mysql-replication-really-work/>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**MySQL Limitations Part 1: Single-Threaded Replication**

I recently mentioned [a few of the big “non-starter” limitations Postgres has overcome](http://www.xaprb.com/blog/2010/10/19/i-wish-i-could-be-at-pgwest/) for specific use cases. I decided to write a series of blog posts on MySQL’s unsolved severe limitations. I mean limitations that really hobble it for major, important needs — not in areas where it isn’t used, but in areas where it is used and thus is used very wastefully compared to its potential.

The first glaring problem is single-threaded replication. It is severe and getting much worse as servers get more and more CPUs and CPU cores. The replication replay process executes in a single thread on the replicas, and thus has no hope of keeping up with even a moderately busy write load on the primary, where many updates are occurring concurrently.

In a lot of Web applications, this isn’t really seen as a huge limitation. That’s because these apps are mostly read traffic, so a single primary can delegate the read workload across several replicas, and the write workload is still a relatively small fraction of the server’s total capacity. But eventually, it does become a problem if the app gets large enough, no matter how large the read-to-write ratio is.

What are some workarounds? Here are a few I’ve seen:

* Use DRBD replication instead of MySQL replication. Problem: you end up with an idle standby server, which can’t be used for serving reads, only for disaster recovery. That’s costly.
* Shard. Write workload is the single most legitimate reason for sharding. It’s too bad that “the replica can’t keep up with the write workload” becomes the reason to shard. I wish we could avoid that.
* Do the replication in the application. I know of applications where they just don’t use built-in replication. When they modify some data, they do it in both places. That’s a headache.
* Try obscure techniques such as external processes to prefetch the data the replica is trying to modify, so it can do it faster. This rarely works.

I’m not criticizing anyone who does these things — there really isn’t much of a good choice. It’s all a matter of picking the solution that’s least evil.

Why isn’t there multi-threaded replication? I think that a) it’s not as easy as it seems, and b) there are a hundred edge cases that make it nearly impossible to choose a solution that works for all situations. For example, mixtures of transactional and non-transactional tables are a nightmare.

Here are a few ideas I’ve either heard, or discussed, or thought of myself:

1. One thread per database. If the application is built such that each database is fully independent, then on the replica, we could start up a thread for each database we see in the binary log, and simply pass the replication events to the appropriate thread.
2. A pool of threads on the replica, and a coordinating thread that hands work to each of them. The coordinator would read from the relay log until it has a complete transaction’s worth of events (not a single statement), and hand the whole transaction to the worker thread. The worker thread would run up until the COMMIT, but not actually commit, and then report back to the coordinator thread. The coordinator thread would ensure that all of the transactions begin in the same order as defined in the relay log, and commit in the same order. If any error occurred, such as a deadlock or lock wait timeout, then the coordinator would instruct the workers to roll back, and either retry or make them execute the problematic transactions in serial instead of concurrently.
3. Multiple binary logs on the primary, one per database; one replication process per binary log on the replica. This would have the advantage that it would allow a replica to subscribe to multiple masters, which is currently impossible.

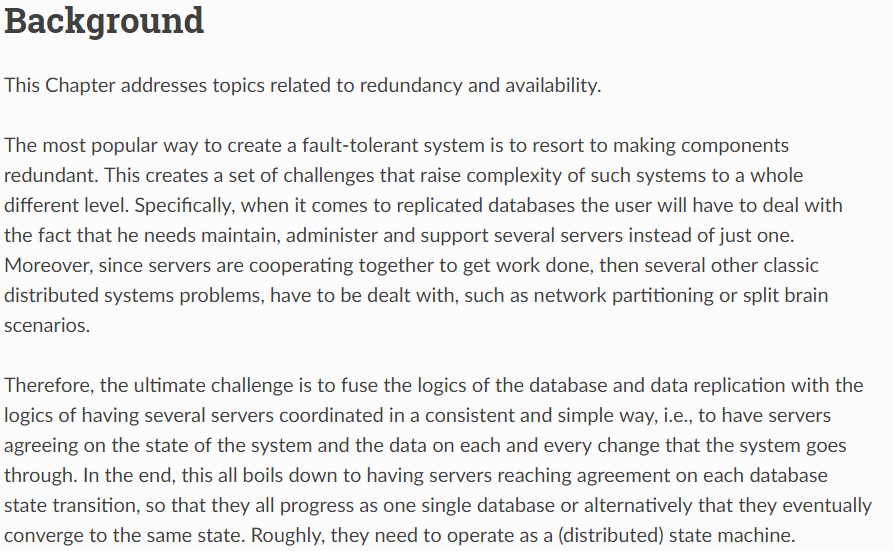
These solutions represent different types of trade-offs. For example, solution 1) only works for specific uses, and I don’t think it’s general-purpose enough. Solution 2) has potentially complex behavior that might not work well in some cases, such as when deadlocks are common; but it is overall the least disruptive or different from the user’s point of view. Solution 3) requires modifying the binary logging code, which is risky. It also requires maintaining many master.info files on the replica, and new SQL syntax for administering replicas, and is generally not something I personally want to administer (replication is fragile enough… imagine recovering after a crash when you have to fix multiple threads that have forgotten where they should be reading in multiple binlogs?).

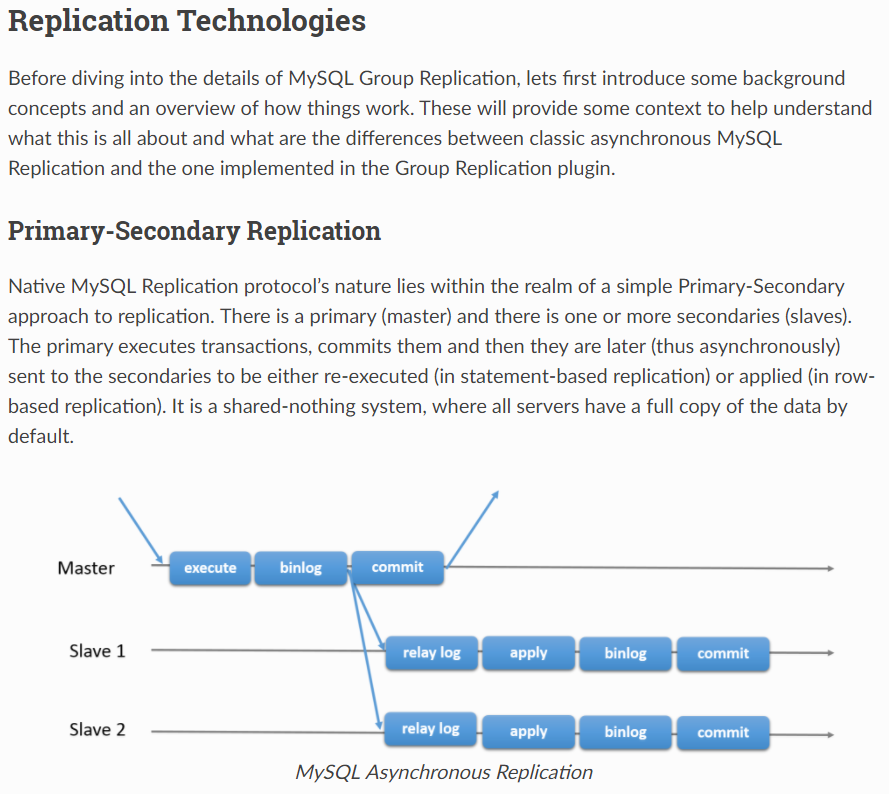
Regardless of the solution, it is certain that nothing is going to work in all cases; the most common cases will require use of InnoDB with the proper transaction isolation level, at a minimum. This behavior is going to have to default to single-threaded as replication currently does, and only enable the multi-threaded behavior if the user configures it to do so. I would be in favor of solution 2) with an option to configure it to behave as solution 1).

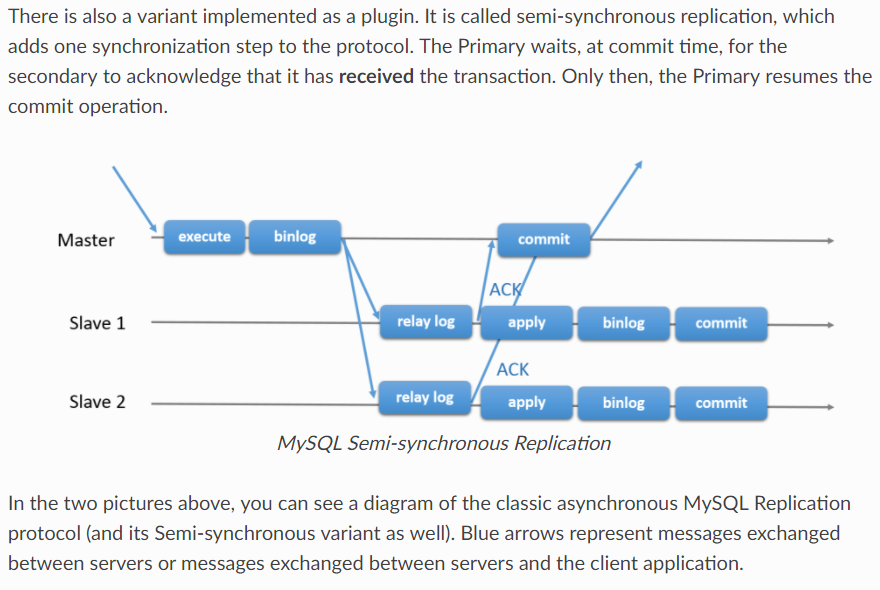
<https://www.percona.com/blog/2010/10/20/mysql-limitations-part-1-single-threaded-replication/>

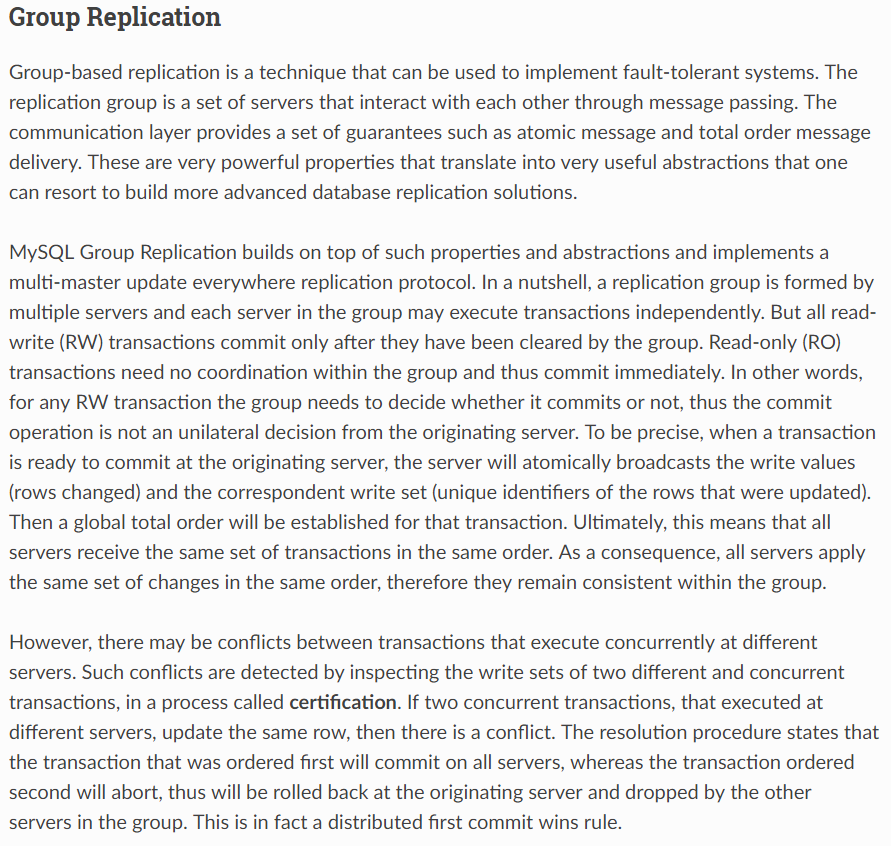
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

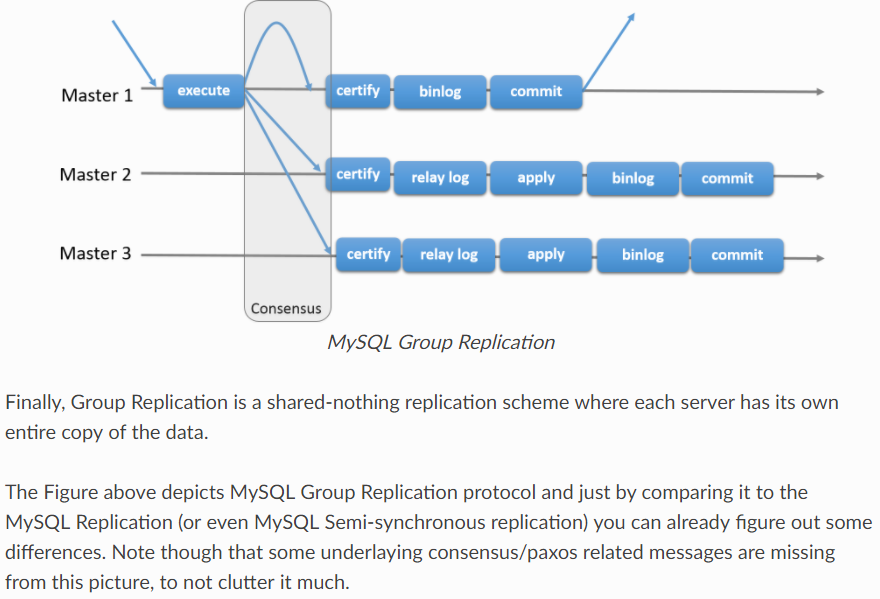
MySQL Group Replication

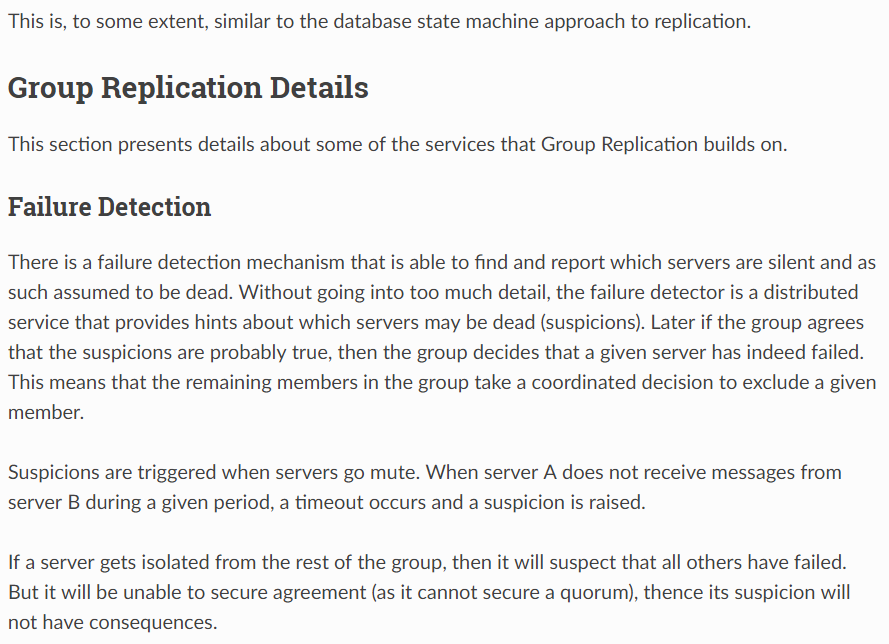


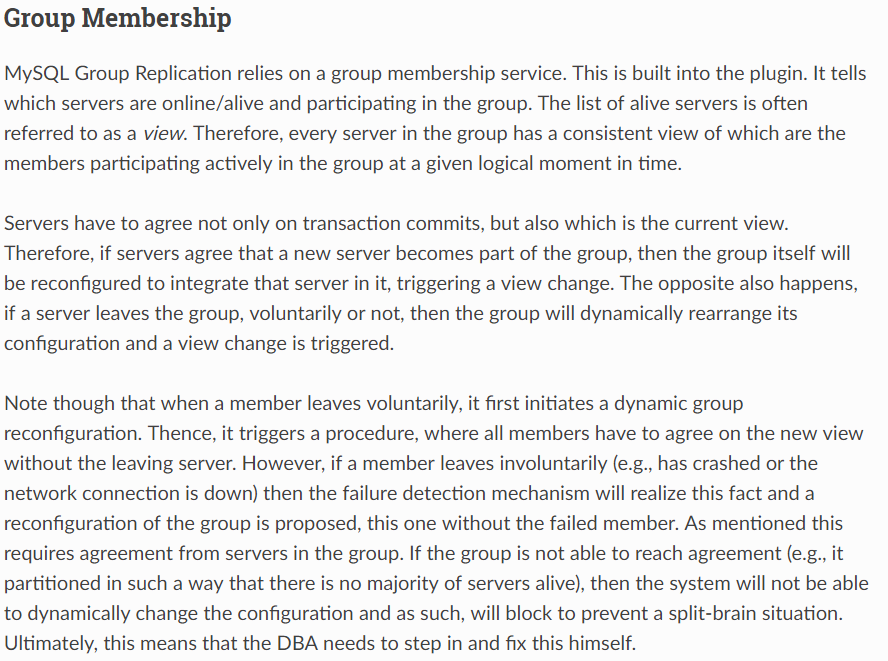


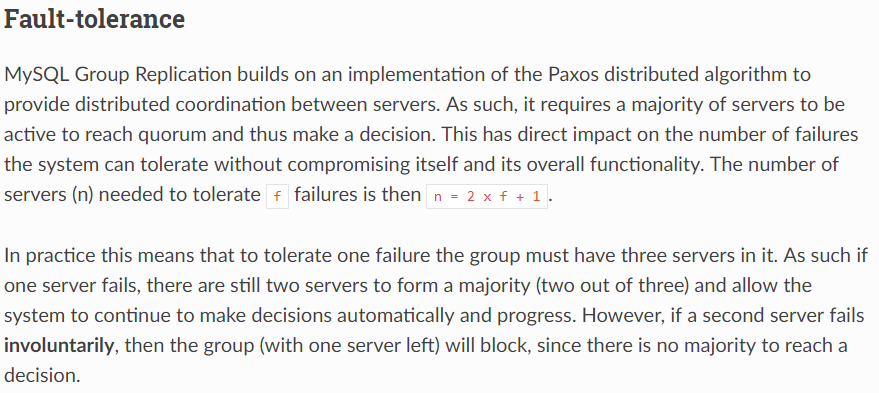


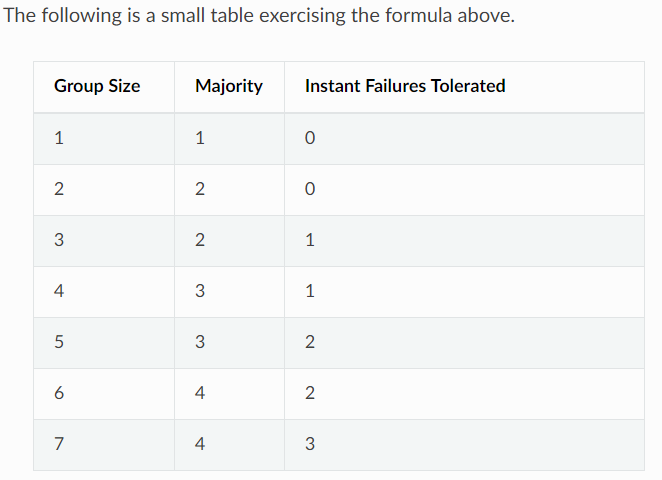












<http://mysqlhighavailability.com/gr/doc/background.html>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

### **MySQL Semi-Synchronous Replication**

MySQL replication by default is asynchronous. The master writes events to its binary log but does not know whether or when a slave has retrieved and processed them. With asynchronous replication, if the master crashes, transactions that it has committed might not have been transmitted to any slave. Consequently, failover from master to slave in this case may result in failover to a server that is missing transactions relative to the master.

Semisynchronous replication can be used as an alternative to asynchronous replication:

* A slave indicates whether it is semisynchronous-capable when it connects to the master.
* If semisynchronous replication is enabled on the master side and there is at least one semisynchronous slave, a thread that performs a transaction commit on the master blocks after the commit is done and waits until at least one semisynchronous slave acknowledges that it has received all events for the transaction, or until a timeout occurs.
* The slave acknowledges receipt of a transaction's events only after the events have been written to its relay log and flushed to disk.
* If a timeout occurs without any slave having acknowledged the transaction, the master reverts to asynchronous replication. When at least one semisynchronous slave catches up, the master returns to semisynchronous replication.
* Semisynchronous replication must be enabled on both the master and slave sides. If semisynchronous replication is disabled on the master, or enabled on the master but on no slaves, the master uses asynchronous replication.

While the master is blocking (waiting for acknowledgment from a slave after having performed a commit), it does not return to the session that performed the transaction. When the block ends, the master returns to the session, which then can proceed to execute other statements. At this point, the transaction has committed on the master side, and receipt of its events has been acknowledged by at least one slave.

Blocking also occurs after rollbacks that are written to the binary log, which occurs when a transaction that modifies nontransactional tables is rolled back. The rolled-back transaction is logged even though it has no effect for transactional tables because the modifications to the nontransactional tables cannot be rolled back and must be sent to slaves.

For statements that do not occur in transactional context (that is, when no transaction has been started with [START TRANSACTION](https://dev.mysql.com/doc/refman/5.5/en/commit.html) or [SET autocommit = 0](https://dev.mysql.com/doc/refman/5.5/en/set-variable.html)), autocommit is enabled and each statement commits implicitly. With semisynchronous replication, the master blocks after committing each such statement, just as it does for explicit transaction commits.

To understand what the “semi” in “semisynchronous replication” means, compare it with asynchronous and fully synchronous replication:

* With asynchronous replication, the master writes events to its binary log and slaves request them when they are ready. There is no guarantee that any event will ever reach any slave.
* With fully synchronous replication, when a master commits a transaction, all slaves also will have committed the transaction before the master returns to the session that performed the transaction. The drawback of this is that there might be a lot of delay to complete a transaction.
* Semisynchronous replication falls between asynchronous and fully synchronous replication. The master waits after commit only until at least one slave has received and logged the events. It does not wait for all slaves to acknowledge receipt, and it requires only receipt, not that the events have been fully executed and committed on the slave side.

Compared to asynchronous replication, semisynchronous replication provides improved data integrity. When a commit returns successfully, it is known that the data exists in at least two places (on the master and at least one slave). If the master commits but a crash occurs while the master is waiting for acknowledgment from a slave, it is possible that the transaction may not have reached any slave.

Semisynchronous replication also places a rate limit on busy sessions by constraining the speed at which binary log events can be sent from master to slave. When one user is too busy, this will slow it down, which is useful in some deployment situations.

Semisynchronous replication does have some performance impact because commits are slower due to the need to wait for slaves. This is the tradeoff for increased data integrity. The amount of slowdown is at least the TCP/IP roundtrip time to send the commit to the slave and wait for the acknowledgment of receipt by the slave. This means that semisynchronous replication works best for close servers communicating over fast networks, and worst for distant servers communicating over slow networks

<https://dev.mysql.com/doc/refman/5.5/en/replication-semisync.html>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**How Does Semi-Synchronous MySQL Replication Work?**

The first thing to understand is that despite the name, **semi-synchronous replication is still asynchronous**. Semi-synchronous is actually a pretty bad name, because there is no strong coupling between a commit on the master and a commit on the replicas. To understand why, let’s look at what truly synchronous replication means. In truly synchronous replication, when you commit a transaction, the commit does not complete until all replicas have also committed successfully. In MySQL’s semi-synchronous replication, the commit completes before the transaction is even sent to any of the replicas. Therefore, by definition the transaction cannot have committed on any of the replicas. If there’s any problem after the commit happens on the master, it’s possible that the replicas won’t get the transaction, and even after they do, there’s no guarantee they can apply and commit it successfully themselves (duplicate key error, anyone?). If any of these problems happens, it’s too late–the commit is already permanent on the master, and can’t be rolled back.

What should semi-synchronous replication be called instead? I believe that it should be called **delayed-acknowledgment commits**, because this is what actually happens. When a transaction commits on the master, the commit proceeds as normal, and the transaction is sent to the replicas as normal, but the client connection to the master is not told that the commit has completed until after at least one replica has acknowledged receiving the transaction.

Another way to look at the same thing is that **semi-synchronous replication actually forces the client to be synchronized**, not the replicas. The client is forced to wait until the transaction has been sent to one of the replicas, but the commit on the master is not forced to wait at all, nor are replicas forced to do anything. The commit has already happened on the master, so the cat’s out of the bag and there’s no way to force replicas to do anything. As a result, the effect is that the client’s activity is throttled so that it cannot outpace the replica’s ability to fetch updates from the master. Have you seen the bumper sticker that says, “don’t drive faster than your Guardian Angel can fly?” That is the effect of this throttling.

Semi-synchronous replication also **does not guarantee that your replicas will not become delayed**. The client connection is forced to wait until at least one of the replicas has retrieved the transaction, but not until the transaction has actually been applied to the replica. As you probably know, it is perfectly possible to send a very long transaction to the replica in a matter of milliseconds. The replica will take a long time to apply this transaction to its own data, and during that time, it will be delayed relative to the master. However, other transactions can continue committing and sending their changes to the replica, because the process of retrieving changes from the master and applying them run in separate threads on the replica.

Finally, **semi-synchronous replication does not provide strong guarantees against data loss**. What do I mean by a strong guarantee against data loss? I consider the safety of my data strongly guaranteed when at least one other server must have a copy of the data before it can be committed on the master. However, that is not what happens in semi-synchronous replication. And if there is an error in semi-synchronous replication, such as a crash at the wrong moment, or a timeout, then even the throttling is abandoned, and everything defaults back to the traditional mode of replication.

What does semi-synchronous replication guarantee me then? If there are no errors or timeouts, then the guarantee is essentially that only one transaction per client is likely to be lost if the master crashes.

I do not mean to sound negative, or to send the message that semi-synchronous replication is not useful. It is useful, but if you misunderstand it, you could be relying on a strong guarantee that is not actually provided.

If you want to learn more about this, then I encourage you to read the relevant section of the [MySQL manual](http://dev.mysql.com/doc/refman/5.5/en/replication-semisync.html). But read carefully, for example, the following sentences:

When a commit returns successfully, it is known that the data exists in at least two places (on the master and at least one slave). If the master commits but a crash occurs while the master is waiting for acknowledgment from a slave, it is possible that the transaction may not have reached any slave.

<https://www.percona.com/blog/2012/01/19/how-does-semisynchronous-mysql-replication-work/>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*