**Multi-master data conflicts - Part 1: understanding the problem**

### What is a conflict?

Readers of this blog know that one of my favorite tools, [Tungsten Replicator](http://tungsten-replicator.org/), can easily create multi-master replication topologies, such as all-masters, star, fan-in. While this is good news for system designers and ambitious DBAs, it also brings some inconvenience. When you allow updates to happen in more than one master, you risk having conflicts. You may have heard this term before. For the sake of clarity, let's define what conflicts are, before analyzing each case in detail.

You have a conflict when **several sources (masters) update concurrently the same data** in asynchronous replication.

It's important to stress that this happens with **asynchronous** replication. In a truly synchronous cluster, where all data is kept consistent through [2-phase commit](http://en.wikipedia.org/wiki/Two-phase_commit_protocol), there is no risk of conflicts.

The above definition is not always correct. You may update data from several sources and end up with something completely legitimate. A better definition should be: **a conflict happens after an unexpected concurrent modification of existing data coming from a remote source.**

For example:

* Both the server A and B insert a record with the same primary key;
* Master A updates record 10 with value 3 while master B updates record 10 with value 4;
* Master A deletes record 10 while master B updates record 10.

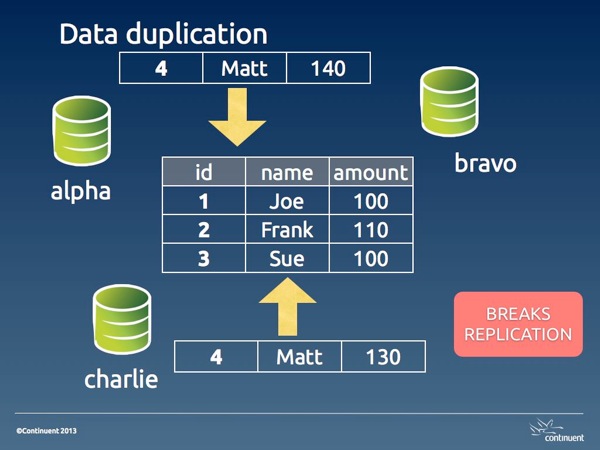
In each of these cases, there is data in one server, which could have been just inserted by a legitimate source, or could have been there for long time. Regardless of the record age, the conflict happens when the data clashes with the latest update. We usually think of conflicts as concurrent events from different sources, because this is the most frequent case of **evident** conflicts, i.e. conflicts that are detected (and hopefully fixed). But there are hidden conflicts that happen (or are discovered) long time after the update.

When we consider the consequences of a conflict, we observe that a conflict creates one or more of the following:

* Duplicate data (unwanted insert)
* Data inconsistency (unwanted update)
* Data loss (unwanted delete)
* Replication break

### Duplicate data

This is the most widely known conflict, because it's often caught immediately after the fact, allowing the DBA to take action. However, it's also one of the less understood cases, as we will see in the next section.



*Image #1. Data duplication*

In its basic shape, data duplication happens when two masters insert the same data. One of the two transactions will be committed before the other, and the second one will fail with a rightful *duplicate key* error. The reason for this occurrence is often an application bug, or a design flaw, or sometimes a human error that the application fails to detect properly (a different application bug).

We will see the possible solution in part 2 of this article. For now, it's enough to know that this kind of error is the best kind of conflicts that can happen to you. Since it breaks replication, it has the positive effect of alerting the DBA about an error.

Depending on which event is applied first, you have different situations. If the remote event is applied first, you have a real conflict, where the legitimate event can't get to the database. This state requires a longer set of actions: you need to clean up both the origin and the destination servers. If the legitimate action is applied first, then you don't have a real conflict, as the wrong event was applied only in one server, and you need to clean up only the origin. If you have more than two masters in your topology, you may find that the damage could be a mix of both cases, as the wrong event may arrive to distant servers before or after the right one.

### auto\_increment\_offsets and hidden data duplication

There is a popular belief that conflicts with multi-master topologies can be avoided using [auto\_increment\_increment](http://dev.mysql.com/doc/refman/5.0/en/replication-options-master.html" \l "sysvar_auto_increment_increment) and [auto\_increment\_offset](http://dev.mysql.com/doc/refman/5.0/en/replication-options-master.html" \l "sysvar_auto_increment_offset). The combination of these two variables makes sure that auto\_increment values are generated with different intervals for each master. For example, if I have three masters and I am using increment 3, the first master will generate 1,4,7,10, the second one will have 2,5,8,11, and the third one 3,6,9,12.

Where does this paradigm work? When the primary key is the auto generated column, then the conflict is prevented. For example, in a table that records bug reports, the incrementing value for the bug number is a legitimate primary key. If two masters enter a bug simultaneously, the bug numbers will have different values. (there will most likely be gaps in the bug numbers, and this could be a non-desirable side effect, but then, I am not advocating this system, although I made this mistake many years ago.)

However, if the table has a natural primary key that is not an auto-incremented value, conflicts are possible, and likely. In that case, you will have a duplicated key error, as in the case seen before.

Disaster strikes when the table has a poorly chosen auto\_increment primary key.

For example, let's consider a departments table with this structure:

CREATE TABLE departments (

dept\_id int not null auto\_increment primary key,

dept\_name varchar(30) not null

);

If two masters need to insert a new department named 'special ops', we may end up with this situation:

select \* from departments;

+---------+-------------+

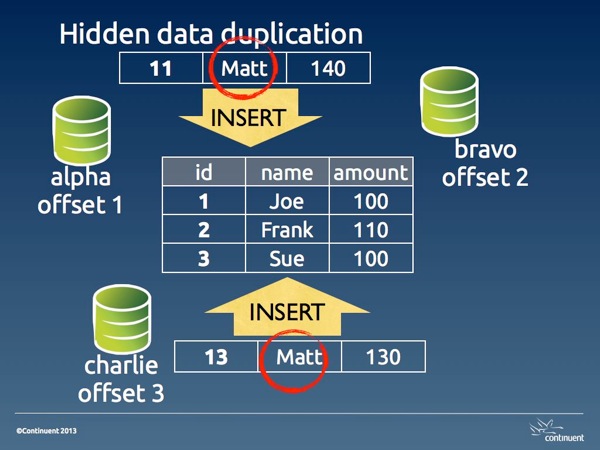
| dept\_id | dept\_name |

+---------+-------------+

| 2 | special ops |

| 4 | special ops |

+---------+-------------+



*Image #2. Hidden data duplication*

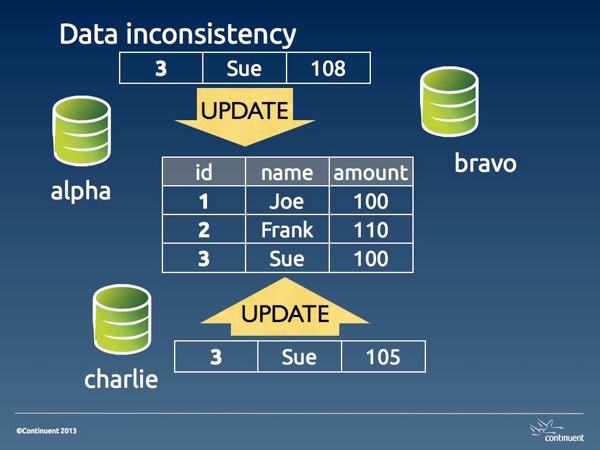
This is what I define **hidden data duplication**, because you have duplicated data, and no errors that may warn you of the problem. Here the issue is aggravated by the fact that 'department' is likely a lookup table. Thus, there will be a table where 'special ops' is referenced using dept\_id 2, and another table where it is used with dept\_id 4.

The reason for hidden data duplication is poor choice of primary key, and failure to enforce unique values in columns that should be such.

### Data inconsistency

When two UPDATE statements are executed on the same record from different sources, there is the possibility of spoiling the data accuracy in several ways. The amount of damage depends on the type of update (with absolute values or calculated ones) and on whether we are using statement-based or row-based replication.

With absolute values, the last value inserted overwrites the previous one.



*Image #3. Data inconsistency*

With calculated values, the data inconsistency may change with surprising consequences. For example, if we have a table accounts:

select \* from accounts;

+----+--------+

| id | amount |

+----+--------+

| 1 | 1000 |

| 2 | 1000 |

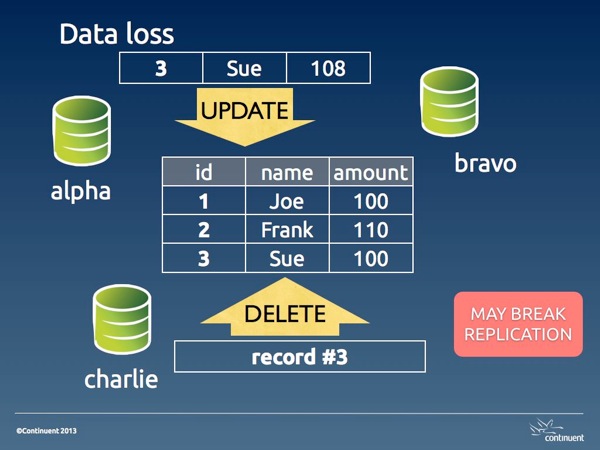
| 3 | 1000 |

+----+--------+

If a statement that doubles the account for ID 2 is executed in two masters, then we will have an amount of 4,000 instead of 2,000. Using row-base replication can protect you against this kind of disaster.

### Data loss

When a DELETE statement is entered for a record that later we want to read, we have lost data. This kind of DELETEs may happen because of bad operations, or more likely because of data inconsistencies that alter the conditions used for deleting.



*Image #4. Data loss*

Unwanted DELETE operations may also break replication when the DELETE happens before an UPDATE on the same record. Either way, a data loss conflict is hard to resolve because the data has gone away. Depending on the amount of the loss, we may need to restore the table completely or partially from a backup.

### Why conflicts happen

To understand why conflicts, happen, let's first see why they don't happen when we try a conflicting operation in the same server.

SESSION\_1 > create table people (id int not null primary key, name varchar(40) not null, amount int);

Query OK, 0 rows affected (0.01 sec)

SESSION\_1 > insert into people values (1, 'Joe', 100), (2, 'Frank', 110), (3, 'Sue', 100);

Query OK, 3 rows affected (0.00 sec)

Records: 3 Duplicates: 0 Warnings: 0

SESSION\_1 > set autocommit=0;

Query OK, 0 rows affected (0.00 sec)

SESSION\_1 > begin;

Query OK, 0 rows affected (0.00 sec)

SESSION\_2 > set autocommit=0;

Query OK, 0 rows affected (0.00 sec)

SESSION\_2 > select \* from people;

+----+-------+--------+

| id | name | amount |

+----+-------+--------+

| 1 | Joe | 100 |

| 2 | Frank | 110 |

| 3 | Sue | 100 |

+----+-------+--------+

3 rows in set (0.00 sec)

SESSION\_2 > begin;

Query OK, 0 rows affected (0.00 sec)

SESSION\_1 > insert into people values (4,'Matt', 140);

Query OK, 1 row affected (0.00 sec)

SESSION\_2 > insert into people values (4,'Matt', 130);

# ... hanging

SESSION\_1 > commit;

Query OK, 0 rows affected (0.01 sec)

SESSION\_2 > insert into people values (4,'Matt', 130);

# ...

ERROR 1062 (23000): Duplicate entry '4' for key 'PRIMARY'

What happens here is that user in session 1 inserts a record at the same time when user in session 2 inserts the same record. When the record is inserted in session 1, InnoDB creates a lock. If you look at the InnoDB locks before SESSION\_1 issues a commit, you will see it:

SESSION\_3 > select \* from information\_schema.innodb\_locks\G

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

lock\_id: 30B:0:307:5

lock\_trx\_id: 30B

lock\_mode: S

lock\_type: RECORD

lock\_table: `test`.`people`

lock\_index: `PRIMARY`

lock\_space: 0

lock\_page: 307

lock\_rec: 5

lock\_data: 4

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

lock\_id: 30C:0:307:5

lock\_trx\_id: 30C

lock\_mode: X

lock\_type: RECORD

lock\_table: `test`.`people`

lock\_index: `PRIMARY`

lock\_space: 0

lock\_page: 307

lock\_rec: 5

lock\_data: 4

2 rows in set (0.00 sec)

You can see that there is an [exclusive lock](http://dev.mysql.com/doc/refman/5.5/en/innodb-lock-modes.html) on the record.

This lock effectively prevents a duplicate.

Now, if you imagine the two sessions happening on different servers, the two users are in a similar situation, i.e. they don't know that a concurrent update on the same record is being attempted. But the difference is that, in asynchronous replication, there is no lock applied on a remote server. If the two transactions are committed at the same instant, both of them will be stored in their local server, and both of them will fail and break replication on the remote server. If the record in session 1 is applied a few seconds before the other, the user in session 2 will not be able to commit, same as it happened with the concurrent insertion in the single server example above. In this case, the conflict looks exactly as it happened in a single server.

However, if both commits happen at the same time, both users will have a positive feedback, since their transaction will return success, and both are happy, at least temporarily. Unknown to both, though, their transaction has failed on the remote server, and replication is broken on both servers, leaving each with a bit of mess to clean up.

These examples show that conflicts are often a matter of chance. Depending on the timing of the operations, we might catch them as they happen and take action before the conflict spreads its side effects, or we only notice later on, when replication fails, and the conflict has already spoiled our data.

### Summing up

Conflicts in multi-master topologies are the consequence of unwanted or unexpected operations. The effects of a conflict range from data inconsistency to data loss, and may also cause replication to break.

The most desirable outcome for a conflict is a replication error, because it prevents further spreading of the error and alerts the DBA about a possible issue.

In the second part of this article, we will look at some of the methods to deal with conflicts in various scenarios

<http://datacharmer.blogspot.in/2013/03/multi-master-data-conflicts-part-1.html>

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

### Multi-master data conflicts - Part 2: dealing with conflicts

**Pessimistic locking (or: conflicts won't happen)**

*Applicability:****synchronous clusters with 2pc***

We've covered this topic in the previous article, but it's worth repeating. If you use a synchronous cluster, you don't have conflicts. For example, MySQL Cluster ensures consistent data with updates coming from different nodes. However, MySQL Cluster is not a replacement for a MySQL server, and it has severe limitations.

**Optimistic locking**

*Applicability:****synchronous clusters without 2pc (Galera)***

Conflicting transactions proceed on different nodes with local locking. The last one then rolls back when it discovers a prior transaction got in first on the same data. For a more detailed analysis of this handling method, see [this article by Jay Janssen](http://www.mysqlperformanceblog.com/2012/11/20/understanding-multi-node-writing-conflict-metrics-in-percona-xtradb-cluster-and-galera/)

**Conflict resolution after-the fact**

*Applicability:****[EnterpriseDB](http://www.enterprisedb.com/products-services-training/products-overview/xdb-replication-server-multi-master) (none so far for MySQL)***

Asynchronous replication is hard for conflicts. A conflict in this state means that the data has been applied to the wrong node or to the wrong object, and something must be done to solve the issue.

Typical remedies offered for conflict resolution are:

* **Earliest** or **Latest Timestamp**: This method says that the oldest or the latest record prevails when a conflict happens. This is hardly a reliable resolution. It's the easiest method to implement, and thus it is offered. But it often results in a hidden data inconsistency problem, where we may find data that we don't expect. The current data was applied simply because it was updated later than the correct record. Also, timestamp calculation requires time synchronization across servers, and possibly across time zones, which calls for extra effort to keep the system functioning.
* **Node Priority**: There is a hierarchy of nodes, with different ranks. When a conflict occurs, the node with the highest rank prevails. This method requires the data origin to be stored alongside the contents, and to be easily searchable when conflicts occur. It must also take into account offline nodes, and therefore it should keep the conflict resolution metadata until the offline nodes are back in synch.

Methods that could be implemented in a more advanced technology may include:

* **Origin enforcement**: data coming from authorized nodes will be preserved. Data from wrong origin will be dropped, and a consolidation event will be generated and sent to the other nodes. This method would be possible in systems (like Tungsten) that keep track of the event origin.
* **Data merge**: If possible and desirable, data from two different sources can be preserved, and merged in the destination table. This rule should also originate a new event to fix the data in the other nodes.

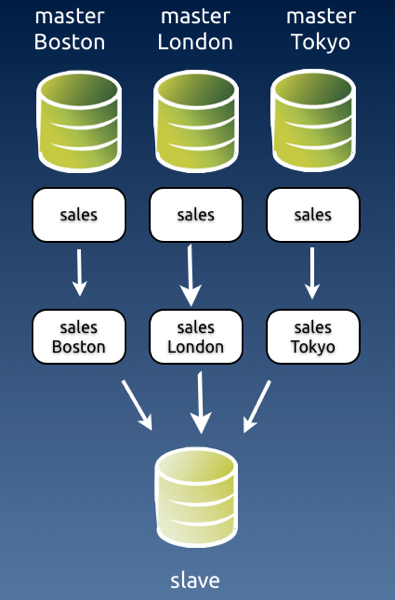
**Schema renaming**

*Applicability:****fan-in topologies***



*Image #1 - Fan-in topology with likely conflicts.*

A fan-in topology is easy to implement with Tungsten Replicator, but not easy to maintain. By its nature, fan-in is a conflict waiting to happen. Assuming that all the masters have the same structure, they will replicate multiple changes into the same schema, and it is quite likely that some changes will clash. For this reason, the simple solution often adopted is renaming the schema before the data reaches the slave.



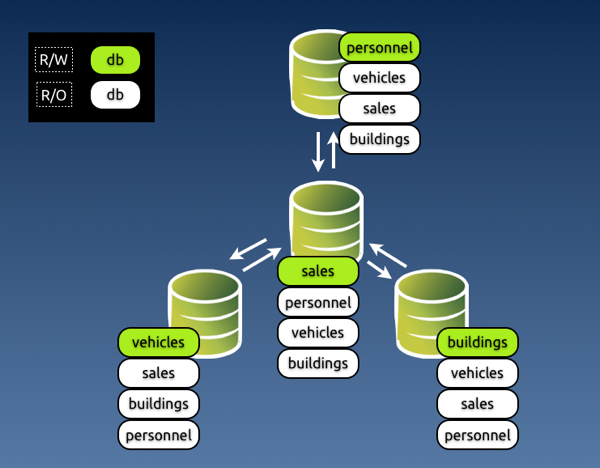
*Image #2 - Fan-in topology with schema renaming.*

I know of at least one user who has successfully applied this technique for a cluster made of 70 masters and one slave.

**Conflict prevention: Discipline**

*Applicability:****all topologies***

A simple way of preventing conflicts, and one that would make life easier for all is **discipline**. The organization decides which entry points can update which data, and conflicts are not possible, because the data is inserted or modified only in the places where it is supposed to be.



*Image #3 - Preventing conflicts with discipline in a star topology.*

**Conflict prevention: Enforced discipline**

*Applicability:****all topologies***

If you have worked in any large organization, either public or private, you know that discipline alone is the worst method you can rely on for something so delicate and valuable as your data. The reasons why this paradigm could fail are many: it could be because some people dislike discipline, or because someone makes a mistake, or because there are too many rules and they don't remember, or because of an application bug that lets you update what you shouldn't.

Either way, you end up with a system that has conflicts and nobody knows what happened and how to fix them. However, there is a way of **enforcing** this system based on discipline.

This is the "poor-man's" conflict avoidance system. It is based on simple technology, available in most database servers. If you can install a multi-master topology, using either native MySQL (circular) replication or [Tungsten Replicator](http://tungsten-replicator.org/) topologies, you can also apply this method.

The key to the system is to **grant different privileges for every master**. Looking at *image #3*, you can enforce discipline by granting different privileges to the application user in every master.

In master #1, where we can update *personnel*, app\_user will have SELECT privileges on all databases, and all privileges on *personnel*.

In master #2, where we can update *sales*, app\_user will have all privileges on *sales* and read only access to the other databases, and so on.

The key to make this system work well is that you should assign the privileges and not let the GRANT statement being replicated. It should work like this:

# master 1

GRANT SELECT on \*.\* to app\_user identified by 'my password';

# This is good for all masters. Let it replicate

# master 1

SET SQL\_LOG\_BIN=OFF;

GRANT ALL on personnel.\* to app\_user; # This won't replicate

# master 2

SET SQL\_LOG\_BIN=OFF;

GRANT ALL on sales.\* to app\_user;

# master 3

SET SQL\_LOG\_BIN=OFF;

GRANT ALL on vehicles.\* to app\_user;

# master 4

SET SQL\_LOG\_BIN=OFF;

GRANT ALL on buildings.\* to app\_user;

This method works quite well. Since updates for a given schema can be applied only in one master, there is little chance of any mischief happening. Conflicts are not completely removed, though. There are super users and maintenance users who can, consciously or not, introduce errors. For these cases, you may want to look at the next section.

**Enforced discipline with certified origin**

*Applicability:****all Tungsten topologies***

Discipline based on granted privileges is often robust enough for your needs. However, if you want to keep track of where the data comes from, you should look at a [System Of Records](http://en.wikipedia.org/wiki/System_of_record) technology, where the origin of each piece of data can be traced to its origin.

Tungsten Replicator implements this technology with several topologies. The [theory of this matter](http://scale-out-blog.blogspot.it/2011/08/system-of-record-approach-to-multi.html) is beautifully explained by Robert Hodges in an article written some time ago. Here I would like to look at the practical stuff.

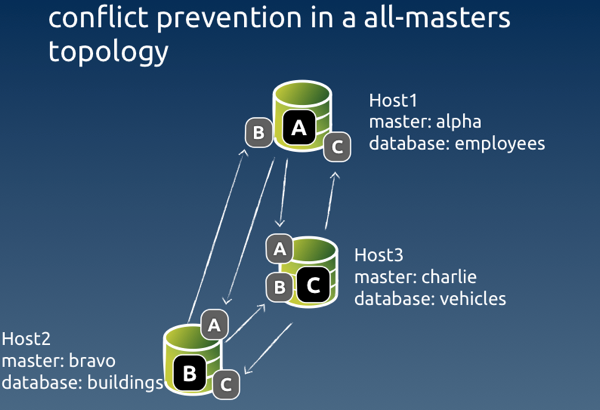
To implement a System of Records in Tungsten, you decide where you want to update each schema (which is defined as a **shard** in our lingo,) assign that schema to a service, and the replicator will enforce your rules.

Once you have defined the shards, you can set the rules. When an event comes to a slave from an **UNKNOWN** shard, i.e. a shard that was not among the defined rules, you can:

* Accept the event; (not recommended, really)
* Drop the event silently
* Drop the event with a warning in the logs;
* Generate an error that will break replication (recommended)

You can choose among the above actions when setting a rule for events that come from **UNWANTED** shards, i.e. a shard that is not the one designated to update that schema.

Here's an example of a shard definition based on an all-masters schema with three nodes:



*Image #4 - Sample conflict prevention in an all-masters topology*

# Options to add during installation

--svc-extractor-filters=shardfilter

# policy for unknown shards

--property=replicator.filter.shardfilter.unknownShardPolicy=error

# policy for unwanted shards

--property=replicator.filter.shardfilter.unwantedShardPolicy=error

# Whether the policy for unwanted shards is activated or not

--property=replicator.filter.shardfilter.enforceHomes=false

# whether we allow whitelists to be created

--property=replicator.filter.shardfilter.allowWhitelisted=false

# Loading the rules set

$ trepctl -host host1 -service charlie shard -insert < shards.map

$ cat shards.map

shard\_id master critical

employees alpha false

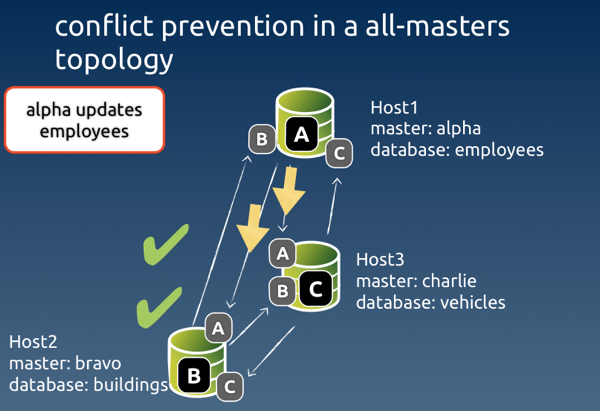
buildings bravo false

vehicles charlie false

test whitelisted false

The rules are set by service, rather than host name. The schema 'employees' can be updated by the service named 'alpha', which has its master in host #1. Similarly, 'buildings' can be updated by 'bravo', with a master in host #2, and 'vehicles' is updated by 'charlie' master service in host #3. Remember that in Tungsten each replication stream from one master to many slaves is a separate service. This way we can keep track of the events origin. Even if the event is routed through a hub in a star topology, it retains its origin in the metadata.

The last line of the rules says that the schema 'test' is whitelisted, i.e. it can be freely updated by any master. And this means that conflicts can happen there, so be careful if you use this feature!



*Image #5 - Example of a legitimate event coming through*

When an **expected** event comes through, all is well. Each node checks that the event was originated by the authorised master, and the event is applied to the slave service.

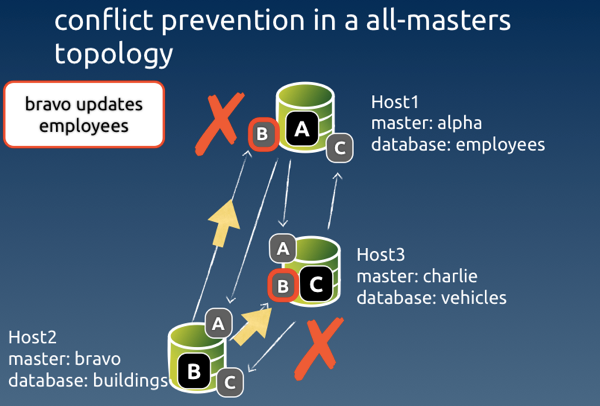


Image #6 - Example of an event originated from an unauthorised node

When the event comes from a node that was not authorised, Tungsten looks at the rules for such case. In our setup, the rule says 'error', and therefore replication will break at the receiving end of the service 'bravo' in host #1 and host #3.

mysql #2> create table employees.nicknames( ... )

# Only server #2 creates the table

# slave service 'bravo' in host1 and host3 get an error

# No table is created in hosts #1 and #3

To detect the error, we can ask for the list of services in host #1 and host #3. What we will see is something like this.

#3 $ trepctl services | simple\_services

alpha [slave]

seqno: 7 - latency: 0.136 - ONLINE

bravo [slave]

seqno: -1 - latency: -1.000 - OFFLINE:ERROR

charlie [master]

seqno: 66 - latency: 0.440 - ONLINE

This Listing says that replication was stopped with an error in slave service 'bravo'. To determine what happened exactly, we ask for the status of that service:

#3 $ trepctl -service bravo status

NAME VALUE

---- -----

appliedLastEventId : NONE

appliedLastSeqno : -1

appliedLatency : -1.0

(...)

offlineRequests : NONE

pendingError : Stage task failed: q-to-dbms

pendingErrorCode : NONE

pendingErrorEventId : mysql-bin.000002:0000000000001241;0

pendingErrorSeqno : **7**

pendingExceptionMessage: **Rejected** event from **wrong shard**:

seqno=7 **shard ID**=employees **shard master=alpha** service=bravo

(...)

This status gives us quite a lot of useful information:

* The event with Global transaction ID (GTID) # 7 was **rejected**;
* The reason for rejection was because it came from the **wrong shard**;
* The expected shard master (i.e. the authorized service) was **alpha**;
* The event was instead originated from service **bravo**.

With the above information, we can take action to fix the event. We know that GTID 7 is wrong, so we can skip it in both servers where the error occurred. To clean up the error, we can simply generate the correct event in the authorized master

#host #1

$ trepctl -service bravo online -skip-seqno 7

mysql #1> drop table if exists employees.nicknames;

mysql #1> create table if exists employees.nicknames ( ... ) ;

#3 $ trepctl -service bravo online -skip-seqno 7

**Statement-based vs row-based replication**

As a general note about conflict solving, I need to mention that, in most cases, using row-based replication vs. statement based will help identifying conflicts, making them easier to clean up.

Even when the conflict involves a deleted row, row-based events will contain enough information that will allow us to identify the critical data needed to recover information.

Be aware that, if you use [binlog-row-image=minimal](http://dev.mysql.com/doc/refman/5.6/en/replication-options-binary-log.html" \l "sysvar_binlog_row_image) in MySQL 5.6, the binary log entry for a DELETE event will only include the primary key.

**More about filters**

We have seen at least in two examples (server renaming and conflict prevention) that you can help avoid conflicts with filters. This is a powerful feature that should be taken into account when planning a multi-master topology.

MySQL native replication offers very little in matter of data transformation through filtering. Tungsten Replicator, instead, allows you to define filters at several stages of the replication process: when extracting the data, after transporting it to the slaves, before applying it. You can write your own filters in JavaScript, and do with the data pretty much everything you want. If you have creative ideas about solving conflicts by manipulating data in transit, there is a good chance that you can implement them using filters. This topic deserves more than a paragraph, and probably I will come back to it soon with a full-fledged article.

**Parting thoughts**

Multi master topologies are much coveted features. However, they often introduce the risk of conflicts.

Dealing with conflicts becomes somewhat easier if you understand how they happen and what kind of problems they generate.

There is no silver bullet solution for conflicts, but recent technology and good organization can help you ease the pain.

<http://datacharmer.blogspot.in/2013/03/multi-master-data-conflicts-part-2.html>