Transaction Log Synchronization Protocol

for GraphDB High Availability Cluster

Scope

This is a description of a protocol, which will be used for synchronizing the transaction logs of two or more read-write master nodes in a high-availability cluster.

The scope specifically *excludes*:

- communication between master and worker nodes
- synchronization between a read-write master and a read-only master
- details of solutions based on external implementations of distributed synchronized structures

Node State Variables

Each Transaction Log contains synchronized-part and incoming-part (represented by Queues). The Synchronized Queue is already agreed between the Masters and is an exact replica of the others, while the Incoming part is being negotiated (the order in which to execute the transactions). We need this order to be agreed upon, in order to guarantee eventual consistency between separate datacenters.

Synchronized queue

A list of transactions that have already been queued for execution on a master node's workers.

During normal operation, the synchronized queue is consistent across all peers in the sense that, at any given time, for any given pair of peers, either their synchronized queues are identical, or a tail of one queue is a prefix of the other queue.

The ID of the last transaction in the synchronized queue is used as a key for matching messages in synchronization rounds (**merge base** – see *Normal Operation*.)

Incoming queue

The queue of incoming transactions, any or all of which may be either created locally, as requested by a client, or received from other nodes.

Each transaction is listed with its ID, timestamp, and originating node ID. The transactions are sorted by timestamp and transaction ID in this order. The transaction ID-s are globally unique and are assigned to the transactions upon creation at the node of origin.

Timestamp Counter (not necessarily related to the system time)

The timestamp assigned to the last transaction.

The counter is a sequence mark, and may or may not be based on the system's time counter.

It's essential that:

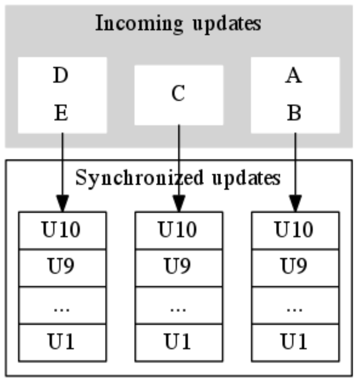
- the counter be maintained so that each new transaction created at this node will receive a timestamp that is *strictly greater* than those for the preceding transactions created there, in order to ensure consistency at the ADD step under normal operation (see below.)
- the counters should be advanced with each synchronization round, and be maintained as close to one another as possible, in order to minimize the time for synchronizing incoming transactions

Normal Operation

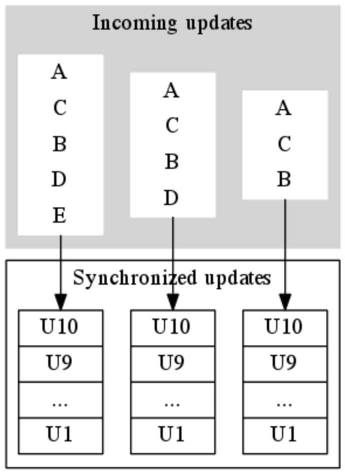
Every participating master node repeatedly executes synchronization rounds, each round consisting of the following steps:

- ➔ **POST** to all other known participating masters a message containing:
  - **merge base** (the ID of the last synchronized transaction)
  - the node's current **timestamp counter**
  - the current contents of its incoming queue (**local queue**)
- ➔ **COLLECT** the incoming queues (**foreign queues**) and **timestamp counters** from incoming messages POST-ed by all other known participating masters for the same **merge base**.
- ➔ **ADD** to the **synchronized queue** the longest common prefix of all collected **foreign queues** and the **local queue**, which contains only transactions with timestamps up to the least (i.e. earliest) of the collected **timestamp counters**.
- ➔ **MERGE** into the **local queue** all unknown transactions from **foreign queues** in their proper sorting places

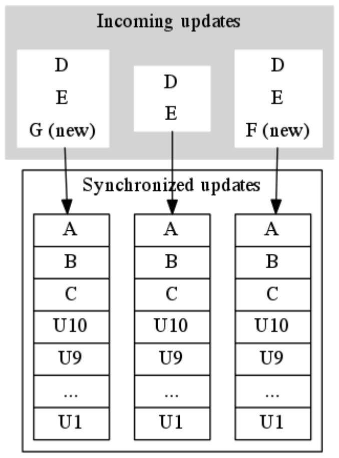
Example operation:



Initial state. The following example represents the Incoming and Sync Queues for three Masters. The start state has U1-U10 in the Sync Queues and the masters have (A, B), (C) and (D, E) as incoming updates.



After the **COLLECT** step, the incoming queues are updated with the updates from other masters. Note that some masters may not contain all the updates (in this case the third master doesn't get D and E), because of the asynchronous operation of the step. After that step, the Incoming queues have the longest common sequence of A, B and C, which is used in the next ADD step.



The final diagram shows the (A, B, C) updates added to the Sync Queue, while the Incoming Queue has (D, E) agreed and two new updates G and F which came asynchronously. The system is ready to perform ADD step for (D, E) to the Sync Queue.

Exceptions

Timeout

- ✖ One or more nodes did not post any status within the round's allotted time.
- ➔ Complete the round between the nodes that did post status using the last known **timestamp counters** for the missing nodes.

It may very well be the case that all remaining known transactions are past the last known counters for the missing nodes. In that case, there will be a sequence of "empty" rounds until the missing nodes reappear. If no progress is marked after a specified amount of time, the remaining nodes continue synchronization without the missing ones (possibly creating a *Split Brain* case.)

Laggers

- ✖ One or more nodes reported **merge bases** that are in the tails of the synchronized queues of other nodes. (All synchronized queues are still consistent with one another, i.e. each two are either identical, or a tail of one is a prefix of the other.)
- ➔ One or more of the forerunners update the lagers with the most up-to-date version of the synchronized queue.

Split Brain

The split-brain scenario happens when two parts of the cluster become disconnected for prolonged time (A typical case for multiple data centers (in e.g. UK and US) is when the UK/US connection is lost).

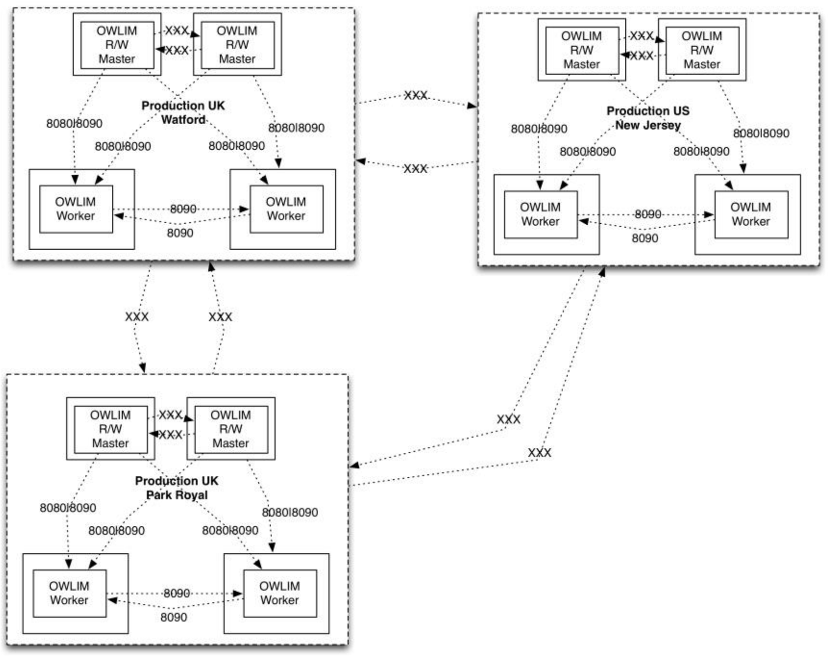
The following happens:

- The masters become disconnected
- For certain period, then continue to collect updates, but don’t execute them, in hope that the connection will become available (this timeout will be a parameter which can be specified from the management console)
- After the timeout is reached, both parts of the cluster do: 1) backup of the database using one of the workers; 2) continue to execute updates against the workers, while synchronizing the updates across the reachable masters

Split brain recovery (after the connection is re-established):

- All masters decide on the correct transaction log using majority vote
- The minority masters: 1) restore the database from the backup; 2) get all the updates from the TxLog after the backup; 3) start executing the updates from the transaction log; 4) continue participating in the TxLog sync
- The majority masters – negotiate the minority updates and add them to their Sync Queues

Split-brain use case



This use case occurs when the UK data centers become disconnected from the US data center.

Note that, in case of a net adapter failure (only one connection between the sub-clusters is lost) the system will continue to function, because the updates that are communicated with the reachable master will be send to the unreachable master on the next **ADD** step.

After the split brain occurs (incl. the timeout):

- UK and US parts of the cluster will backup their DBs and continue to operate independently
- Both databases are inconsistent for the time being –a client routed via the US-ClientAPI will see different results than a client routed via the UK-Client API

Split-brain recovery:

- After the connection is reestablished, the US masters will report different **merge base** that the UK masters
- The majority vote will decide that the UK part is “correct”
- US-cluster will restore its database
- US-cluster will negotiate its updates as new updates
- US-cluster will start executing updates to catch up with the UK-cluster

- Both UK data centers will continue executing updates, because the US cluster reports a known/older **merge base** and it is obviously lagging behind

Additional Notes

Post/Collect Timing and Matching

A node should be prepared to collect status posted from other nodes before it posted its own. It is also possible that one node's synchronization round will include a different set of messages than those of a peer's round. Even so, the nodes are still guaranteed to reach agreement under normal operation, as long as timestamp counters are properly processed and maintained.

Idle/Busy Mode

While there are unsynchronized transactions in the incoming queues, the nodes should execute synchronization rounds back-to-back (**busy mode**) in order to synchronize the queues in the shortest possible time.

There could be periods of missing or low activity, during which the nodes would be exchanging empty incoming queues (idle rounds.) In such cases, the nodes should enter **idle mode** and execute synchronization rounds less often (e.g. once a second to once a minute.)

Nodes should switch to **idle mode** on the first idle round, and back to **busy mode** on the first non-empty status posted by any peer, or on an incoming transaction.

Transaction Exchange

Only the minimum information about transactions will be exchanged in the synchronization protocol. This will prevent bulky updates (e.g. large INSERT statements) from clogging the protocol with data that is irrelevant to synchronization. Nodes will acquire transaction data from the originating peer asynchronously through a separate interface (whose details are irrelevant to the synchronization protocol.)

Acquired Transactions Only

Any node should be able to execute a transaction that has already been merged into the synchronized queue. In order to guarantee this, nodes will only post transactions that they have been able to acquire (see *Transaction Exchange*.)