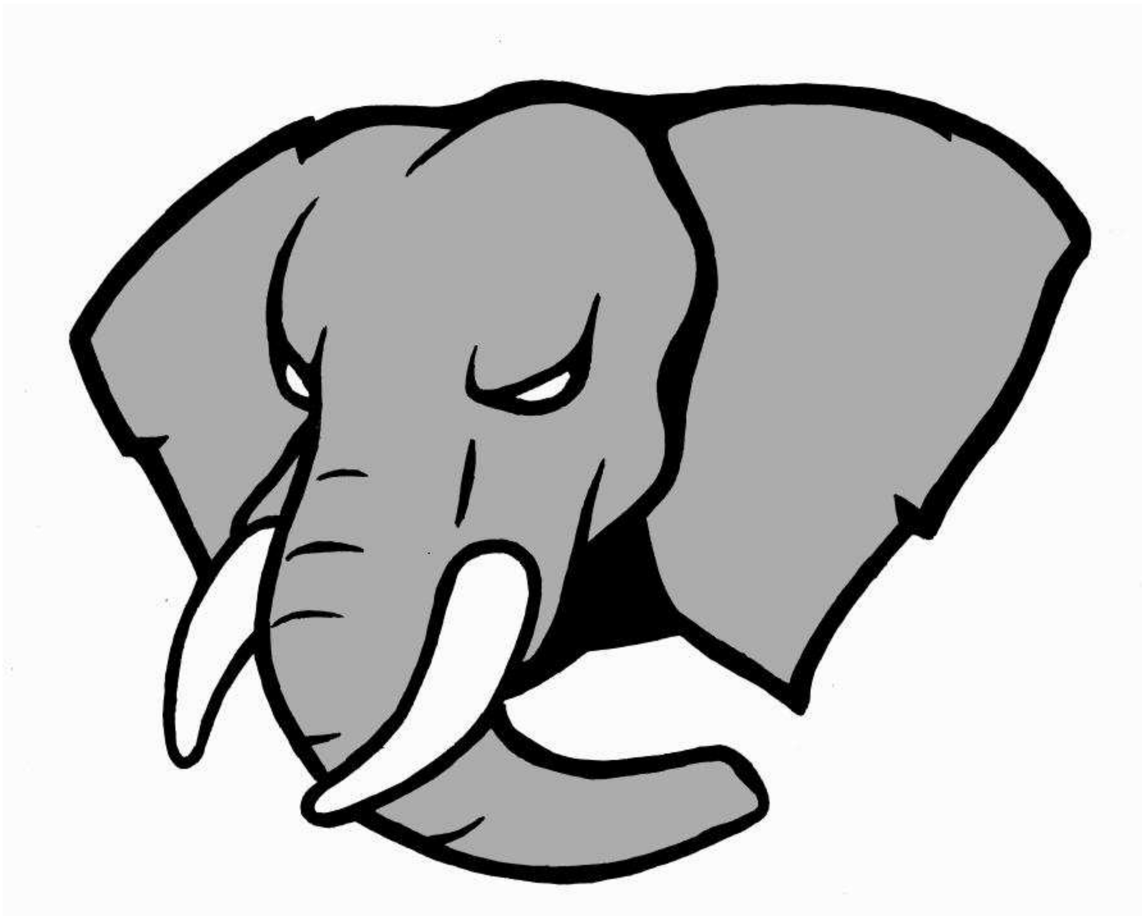

Slony-I
A replication system for PostgreSQL

Implementation details

Jan Wieck

Afilias USA INC.
Horsham, Pennsylvania, USA

ABSTRACT

This document describes several implementation details of the Slony-I replication engine and related components.

Table of Contents

1. Control data 1
1.1. Table sl_node 1
1.2. Table sl_path 1
1.3. Table sl_listen 2
1.4. Table sl_set 2
1.5. Table sl_table 2
1.6. Table sl_subscribe 2
1.7. Table sl_event 2
1.8. Table sl_confirm 2
1.9. Table sl_setsync 3
1.10. Table sl_log_1 3
1.11. Table sl_log_2 3
2. Replication Engine Architecture 4
2.1. Sync Thread 4
2.2. Cleanup Thread 4
2.3. Local Listen Thread 4
2.4. Remote Listen Threads 5
2.5. Remote Worker Threads 5

1. Control data

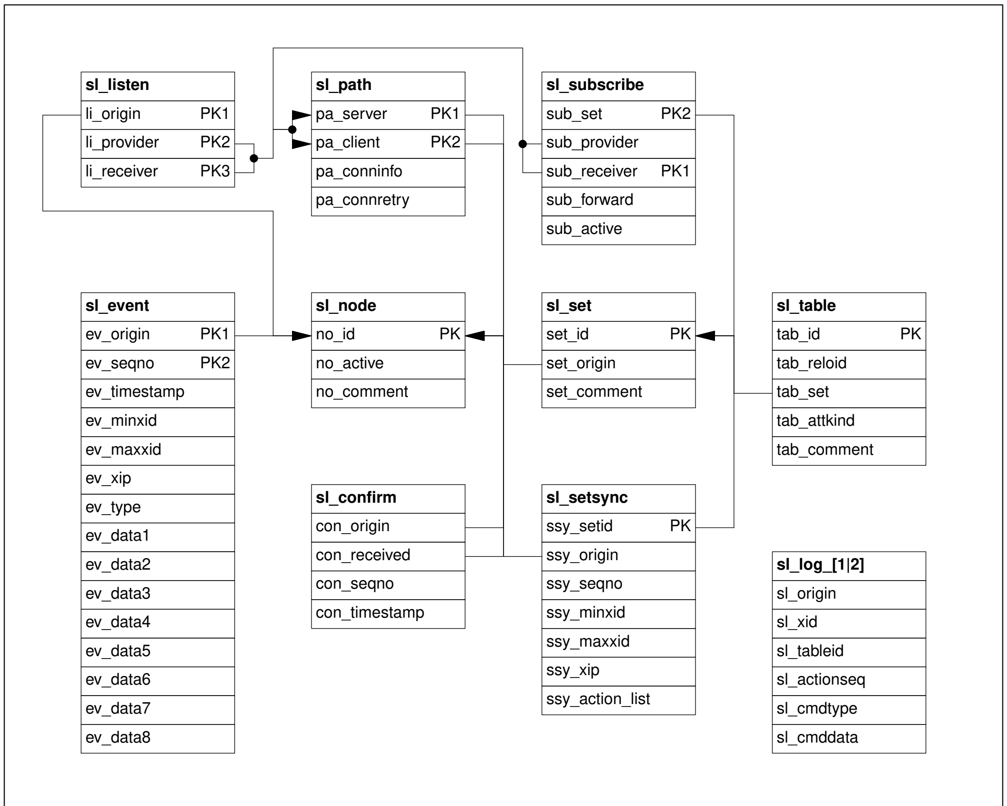

Figure 1

Figure 1 shows the Entity Relationship Diagram of the Slony-I configuration and runtime data. Although Slony-I is a master slave replication technology, the nodes building a cluster do not have any particular role. All nodes contain the same configuration data and are running the same replication engine process. At any given time, a collection of tables, called set, has one node as its origin. The origin of a table is the only node that permits updates by regular client applications. The fact that all nodes are functionally identical and share the entire configuration data makes failover and failback a lot easier. All the objects are kept in a separate namespace based on the cluster name.

1.1. Table sl_node

Lists all nodes that belong to the cluster. The attribute no_active is NOT intended for any short term enable/disable games with the node in question. The transition from disable to enable of a node requires full synchronization with the cluster, resulting possibly in a full set copy operation.

1.2. Table sl_path

Defines the connection information that the pa_client node would use to connect to pa_server node, and the retry interval in seconds if the connection attempt fails. Not all nodes need to be able to connect to each other. But it is good practice to define all possible connections so that the configuration is in place for an eventual failover. An sl_path entry alone does not actually cause a connection to be established. This requires sl_listen and/or sl_subscribe entries as well.

1.3. Table sl_listen

Specifies that the li_receiver node will select and process events originating on li_origin over the database connection to the node li_provider. In a normal master slave scenario with a classical hierarchy, events will travel along the same paths as the replication data. But scenarios where multiple sets originate on different nodes can make it necessary to distribute events more redundant.

1.4. Table sl_set

A set is a collection of tables and sequences that originate on one node and is the smallest unit that can be subscribed to by any other node in the cluster.

1.5. Table sl_table

Lists the tables and their set relationship. It also specifies the attribute kinds of the table, used by the replication trigger to construct the update information for the log data.

1.6. Table sl_subscribe

Specifies what nodes are subscribed to what data sets and where they actually get the log data from. A node can receive the data from the set origin or any other node that is subscribed with forwarding (cascading).

1.7. Table sl_event

This is the message passing table. A node generating an event (configuration change or data sync event) is inserting a new row into this table and does Notify all other nodes listening for events. A remote node listening for events will then select these records, change the local configuration or replicate data, store the sl_event row in its own, local sl_event table and Notify there. This way, the event cascades through the whole cluster. For SYNC events, the columns ev_minxid, ev_maxxid and ev_xip contain the transactions serializable snapshot information. This is the same information used by MVCC in PostgreSQL, to tell if a particular change is already visible to the transaction or considered to be in the future. Data is replicated in Slony-I as single operations on the row level, but grouped into one transaction containing all the changes that happened between two SYNC events. Applying the last and the actual SYNC events transaction information according to the MVCC visibility rules is the filter mechanism that does this grouping.

1.8. Table sl_confirm

Every event processed by a node is confirmed in this table. The confirmations cascade through the system similar to the events. The local cleanup thread of the replication engine periodically condenses this information and then removes all entries in sl_event that have been confirmed by all nodes.

1.9. Table sl_setsync

This table tells for the actual node only what the current local sync situation of every subscribed data set is. This status information is not duplicated to other nodes in the system. This information is used for two purposes. During replication the node uses the transaction snapshot to identify the log rows that have not been visible during the last replication cycle. When a node does the initial data copy of a newly subscribed to data set, it uses this information to know and/or remember what sync points and additional log data is already contained in this actual data snapshot.

1.10. Table sl_log_1

The table containing the actual row level changes, logged by the replication trigger. The data is frequently removed by the cleanup thread after all nodes have confirmed the corresponding events.

1.11. Table sl_log_2

The system has the ability to switch between the sl_log_1 and this table. Under normal circumstances it is better to keep the system using the same log table, with the cleanup thread deleting old log information and using vacuum to add the free'd space to the freespace map. PostgreSQL can use multiple blocks found in the freespace map to actually better parallelize insert operations in high concurrency. In the case nodes have been offline or fallen behind very far by other means, log data collecting up in the table might have increased its size significantly. There is no other way than running a full vacuum to reclaim the space in such a case, but this would cause an exclusive table lock and effectively stop the application. To avoid this, the system can be switched to the other log table in this case, and after the old log table is logically empty, it can be truncated.

2. Replication Engine Architecture

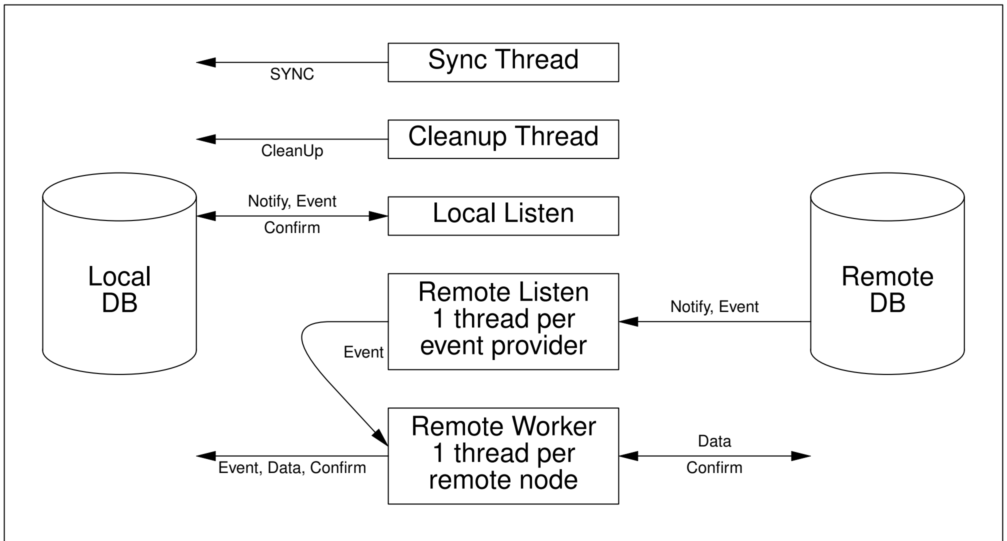

Figure 2

Figure 2 illustrates the thread architecture of the Slony-I replication engine. It is important to keep in mind that there is no predefined role for any of the nodes in a Slony-I cluster. Thus, this engine is running once per database that is a node of any cluster and all the engines together build "one distributed replication system".

2.1. Sync Thread

The Sync Thread maintains one connection to the local database. In a configurable interval it checks if the action sequence has been modified which indicates that some replicable database activity has happened. It then generates a SYNC event by calling CreateEvent(). There are no interactions with other threads.

2.2. Cleanup Thread

The Cleanup Thread maintains one connection to the local database. In a configurable interval it calls the Cleanup() stored procedure that will remove old confirm, event and log data. In another interval it vacuums the confirm, event and log tables. There are no interactions with other threads.

2.3. Local Listen Thread

The Local Listen Thread maintains one connection to the local database. It waits for "Event" notification and scans for events that originate at the local node. When receiving new configuration events, caused by administrative programs calling the stored procedures to change the cluster configuration, it will modify the in-memory configuration of the replication engine accordingly.

2.4. Remote Listen Threads

There is one Remote Listen Thread per remote node, the local node receives events from (event provider). Regardless of the number of nodes in the cluster, a typical leaf node will only have one Remote Listen Thread since it receives events from all origins through the same provider. A Remote Listen Thread maintains one database connection to its event provider. Upon receiving notifications for events or confirmations, it selects the new information from the respective tables and feeds them into the respective internal message queues for the worker threads. The engine starts one remote node specific worker thread (see below) per remote node. Messages are forwarded on an internal message queue to this node specific worker for processing and confirmation.

2.5. Remote Worker Threads

There is one Remote Worker Thread per remote node. A remote worker thread maintains one local database connection to do the actual replication data application, the event storing and confirmation. Every Set originating on the remote node the worker is handling, has one data provider (which can but must not be identical to the event provider). Per distinct data provider over these sets, the worker thread maintains one database connection to perform the actual replication data selection. A remote worker thread waits on its internal message queue for events forwarded by the remote listen thread(s). It then processes these events, including data selection and application, and confirmation. This also includes maintaining the engines in- memory configuration information.