A Proposal for a Multi-Master Synchronous Replication System

Neil Conway (neilc@samurai.com), Gavin Sherry (gavin@alcove.com.au)

January 12, 2006

Contents

1 Introduction

This document describes the design of the **Slony** Π replication system for **PostgreSQL**, the status of the **Slony** Π prototype, and outstanding issues with the system.

2 Design goals

Database replication is best viewed not as a single problem, but as a set of related problems. These related problems have significantly different requirements; in our view, no package can realistically attempt to provide a solution to all the possible variants of database replication. Therefore, our system is designed under the following assumptions:

- multi-master updates are required: any node in the cluster should be able to accept update queries
- synchronous replication is required: SQL isolation levels should be respected as far as possible, and ACID guarantees should be maintained (i.e. a committed transaction should never subsequently be aborted)
- the network is (mostly) reliable and fully connected, and most of the nodes are online most of the time. When a node is not connected to the cluster, it cannot process write transactions
- the members of the cluster are connected via a high-speed network interlink
- the members of the cluster have reasonably similar hardware
- the cluster contains on the order of 1–100 nodes

In addition to basic replication functionality, the system will also:

- allow the replication of many DDL commands. This is part of the goal that migration from a single-node **PostgreSQL** installation to a **Slony** Π cluster should require as few application-level changes as possible
- allow for the efficient recovery of crashed or disconnected nodes: when a node joins the cluster, it will automatically synchronize itself with the current state of the cluster
- survive the destruction of any one or more nodes in the cluster without stopping operation
- survive network partitions and intermittent network failures. Where possible, the cluster should continue to accept transactions during these events
- provide the necessary interfaces to allow monitoring tools and administrative utilities to be written
- integrate with heartbeat applications, load balancers, and similar systems as appropriate

3 Algorithm summary

The **Slony** Π algorithm is derived from Postgres-R [WK05] and related research by Professor Bettina Kemme and colleagues. In this section, we describe the basic ideas of the algorithm. [KA00] and [KBB01] contain a more thorough discussion of the theoretical grounding of this work.

3.1 Problem definition

In multi-master database replication, a cluster consists of a set of nodes connected via a network. Each node contains a copy of the entire database. Queries can be submitted to any node in the cluster. Some of these queries may *conflict*: for example, if two concurrent transactions attempt to modify the same row, both updates cannot be applied. Therefore, a multi-master replication system has two essential requirements:

1. The changes submitted to any node in the cluster must be broadcast to the other members of the cluster.
2. Conflicts between updates must be detected and resolved in a consistent manner.

3.2 Representing changes

Before we can communicate the changes made at one node to the rest of the cluster, we need a format to represent those changes. Section 5 describes the format we have chosen, known as a *writeset*.

3.3 Generating writesets

As a transaction executes, **Slony** Π collects the set of rows modified by the transaction. When the transaction is ready to commit, we check whether it has modified any rows. If it has not, it is allowed to commit: read-only transactions incur no overhead.[1] If the transaction has modified one or more rows, those rows are serialized to writesets and then sent to each member of the cluster.

3.4 Communicating changes

The nodes in a **Slony** Π cluster communicate primarily via a Group Communication System (GCS) [CKV01]. A GCS provides some useful primitives for sending messages between the members of a cluster:

- each message is *multicast* to the entire cluster at once
- all nodes in the cluster are guaranteed to receive the same messages in the same order — the GCS defines a *total order* over all messages

The requirements we have from the GCS are further described in section 11.

[1] This is a valuable optimization, as a high proportion of many real-world workloads consists of read-only transactions.

3.5 Resolving conflicts

If two concurrent transactions attempt to modify the same row, they cannot both be allowed to commit. In general, this situation is called a *conflict*; conflicts are handled by committing the "earlier" transaction and aborting the "later" one, where the ordering between transactions is the total order defined by the GCS.

4 Architecture

Slony Π is implemented as a modification to **PostgreSQL**. A node in a **Slony** Π cluster is a modified installation of **PostgreSQL**. It accepts input from two sources:

local connections: normal PostgreSQL connections are accepted via the usual means (Unix domain sockets or TCP sockets).

replication data: data about the state of the cluster and the writesets generated by concurrent transactions on other nodes are delivered completely separately, using the communications API defined by the GCS.

To handle communication with the GCS, we have added a new component to PostgreSQL, the *replication manager* (RM). This component is responsible for:

- connecting to the cluster and initializing the initial state of the node. Recovery may be required, as described in section 9.
- reading data from the GCS, and applying the writesets it receives to the local node. Writeset application is performed by one or more modified `postgres` backends, known as "replication backends".
- sending data to the GCS that describes transactions executed on the local node, and then determining if and when those transactions can be allowed to commit.

Thus, the RM serves as an intermediary between the local PostgreSQL instance (which is largely unmodified) and the rest of the **Slony** Π cluster.

5 Writesets

The modifications that a transaction makes to SQL data are collected as the transaction executes and then serialized into a format called a *writeset*. This format is intended to allow the changes made by the transaction to be applied at any node in the cluster. Furthermore, it should be as efficient as possible to apply a writeset, as many thousands of writesets will typically be generated and applied per minute on a busy **Slony** Π cluster.

It is worth noting that the writeset format cannot be identical to the SQL query strings executed by the transaction itself. One reason for this is that SQL is non-deterministic in general. For example, queries such as:

INSERT INTO *table* **VALUES** (random());

are not guaranteed to yield the same results at all nodes. Instead, a writeset describes the tuples modified by a transaction relative to the primary key of each modified row.[2] For instance, an SQL

[2] **Slony** Π is only capable of replicating tables with a primary key.

`UPDATE` statement is represented as:

- the values of the primary keys for each modified row
- the columns modified by the `UPDATE`
- the new values of each modified column for each modified row

The format of writesets for `DELETE` and `INSERT` statements follows naturally.[3] An update writeset in this format could *conceptually* be applied at a remote site as follows:

UPDATE *table*
WHERE *pk* = ...
SET *col1* = ... **AND** *col2* = ... **AND** *colN* = ...;

(In practice, writeset application should be implemented by parsing the writeset format itself, *not* by first translating the writeset back into SQL.)

Writesets are only communicated to the cluster at transaction-commit time at the originating node. This avoids the need to send a message to the network for each command executed in the transaction.

6 Conflict resolution

Slony II allows the execution of read and write transactions on any node in the cluster at any time. Write-write conflicts are resolved according to the rules of the "serializable" isolation level defined by the SQL standard: if a transaction T_j updates data that has been updated by a transaction T_i that committed after T_j began, T_j is aborted.

When a transaction containing one or more write operations attempts to commit, its writeset is broadcast to the cluster via the GCS. One of the guarantees provided by the GCS is a *total order* over the messages sent to the cluster: therefore, all nodes will receive all the writesets in the same order. Conflicts between replicated writesets can be resolved consistently by obeying the total order, without the need for explicit concurrency control over the network (e.g. distributed locking or two-phase commit).

6.1 Concurrency control in PostgreSQL

In "serializable" isolation level, the system emulates the serial execution of transactions. If the system cannot hide the effect of one concurrently-executing transaction from another (for example, due to a write-write conflict between the transactions), one of the transactions must be aborted.

To implement concurrency control on a single system, **PostgreSQL** uses transaction identifiers (XIDs). Every tuple stores two XIDs: that which created it (`t_xmin`) and that which invalidated it (`t_xmax`).

When a read operation occurs, **PostgreSQL** knows that tuples with a `t_xmin` equal to a transaction which is still in progress cannot be visible. Likewise, tuples with a `t_xmax` equal to a transaction which is still in progress have not yet been deleted and as such are visible. When a write operation occurs, we fetch the latest version of the tuple. If we find a tuple whose `t_xmin`

[3] DDL statements are represented differently.

or `t_xmax` is a concurrent transaction, we must block to see the outcome of the transaction. If the transaction aborts, then we can proceed. If it commits, then we must abort because there is a conflict.

6.2 Distributed concurrency control

Implementing concurrency control in a distributed environment imposes some additional challenges. On a single machine, there is a single shared disk; if one transaction attempts to modify a tuple that another transaction has updated, the conflict is immediately visible. In contrast, in a replicated environment we have no knowledge of concurrent transactions on other nodes: replicated transactions communicate their writesets only when they commit.

PostgreSQL resolves conflicts between local transactions by aborting the transaction that started later. In **Slony** Π, we use a similar system: if two writesets conflict, we abort the writeset that comes later in the total order defined by the GCS. Specifically, a writeset can conflict with another transaction in two ways:

- if the writeset conflicts with an uncommitted local transaction at the local node, we resolve the conflict by aborting the local transaction[4]
- if the writeset conflicts with another writeset, we resolve the conflict by aborting the writeset that comes later in the total order.

Note that these rules will ensure that a transaction is either committed or aborted in the same fashion on all the nodes in the cluster: in the first case, the local transaction has not yet communicated its writeset, so it can be aborted without affecting consistency. In the second case, the total order is the same on all nodes; therefore, each node will resolve conflicts in the same manner.

6.3 Committing writesets

Because conflicts are only detected at commit-time, we *cannot* allow a replicated transaction to commit unless we are certain it will not be aborted by another transaction. Therefore, before we commit a transaction at a node, that node must be able to verify that each transaction that might cause it to abort does not conflict with it. How can we verify this?

The key observation is that *a replicated transaction can only be aborted by a transaction that comes before it in the total order*, because conflicts are resolved by aborting the writeset that occurs later in the total order. Therefore, once a node has applied each writeset that precedes a writeset w in the total order, it knows that w can commit — any transactions that conflict with w will be aborted because they must come later in the total order.

Therefore, committing a transaction at its originating node requires the following steps:

1. Serialize the rows modified by the transaction into a writeset w
2. Broadcast w via the GCS. This communicates w to the rest of the cluster and simultaneously determines w's position in the total order.

[4]Otherwise, the local transaction would eventually broadcast its writeset to the GCS. At that point it would come later in the total order, so the conflict would otherwise be resolved by aborting it. Aborting the local transaction earlier is just an optimization.

3. Read writesets back from the GCS. For each such writeset w':
    - If $w' = w$, we know that w does not conflict with a writeset that precedes it in the total order, so we can commit it.
    - If w' conflicts with w, we know that w' precedes w in the total order (since we have not seen w yet). Therefore w should be aborted.
    - If w' conflicts with an uncommitted local transaction, abort the local transaction.
    - Otherwise, commit w' and read the next writeset from the GCS.

Applying writesets at other nodes in the cluster requires a similar procedure:

1. The replication manager at each cluster reads writesets from the GCS.
2. For each such writeset w:
    - If w conflicts with an uncommitted local transaction, abort the local transaction.
    - If w conflicts with a writeset w' that precedes this transaction in the total order, abort w.
    - If w conflicts with a writeset w' that follows this transaction in the total order, abort w' and commit w.
    - Otherwise, commit w and read the next writeset from the GCS.

6.4 Alternative algorithms

The algorithm we have chosen is *not* the traditional approach to multi-master replication, which is Two-Phase Commit (2PC) [BHG87]. While it is beyond the scope of this report to discuss 2PC in detail, we find Gray's analysis of 2PC's scalability problems to be convincing [GHOS96]. As Gray argues,

> The probability of deadlocks, and consequently failed transactions rises very quickly with transaction size and with the number of nodes. A ten-fold increase in nodes gives a thousand-fold increase in failed transactions (deadlocks).

In **Slony** Π, only a single multicast message is required for each replicated transaction. Since the total order guarantees that each node will resolve conflicting updates in the same manner, there is no need for an explicit "transaction committed" protocol message. One of the goals of the prototype is to verify the performance and scalability of the algorithm.

Another advantage of this algorithm is its simplicity. Participating in a cluster merely requires that a node apply the data modifications according to the total order. This makes recovery simple, as described in section 9.

7 Global Transaction Identifiers (GIDs)

We are often interested in the position of a writeset in the total order defined by the GCS. To allow this to be determined simply, we assign each replicated transaction a Global Transaction Identifier (GID), which is a cluster-unique 64-bit identifier. GIDs serve two purposes:

- they uniquely identify writesets. Furthermore, a writeset has the same GID on all nodes in the cluster.
- they describe the position of a writeset within the total order: iff $GID(w) < GID(w')$, w precedes w' in the total order.

GIDs in **Slony** Π serve a similar purpose to transaction identifiers (XIDs) in normal **PostgreSQL**. Unlike XIDs in **PostgreSQL**, we neglect to account for GID wraparound: our assumption is that a 64-bit counter will not wraparound within a reasonable amount of time.

7.1 Assigning GIDs

There are various ways to ensure that the GID sequence is synchronized among the nodes in the cluster, but the easiest and most efficient approach is to use the total order defined by the GCS. Whenever a node receives a network message that requires a GID, it increments its local GID counter and sets the GID of the message to the new value of the counter. Since all connected nodes will receive the same set of messages in the same order, this ensures GIDs will be kept in sync across the cluster. Dealing with synchronizing the GID counter on nodes that join the cluster is discussed in section 8.1.

Note that a writeset's GID is defined by its position in the total order. That position is only determined when the writeset is received back from the GCS—when a node sends a writeset, it does not yet know what the GID of that writeset is going to be. Therefore, a writeset does not contain its own GID, as that is not known when the message is constructed. Instead, each node independently assigns a GID to each message it receives, as discussed above.

8 Cluster membership

Slony Π allows nodes to join and leave the cluster at any time. Membership in a **Slony** Π cluster is similar but not identical to membership in a particular group of the GCS: all the nodes in a **Slony** Π cluster are members of the same GCS group, but all members of that group are not necessarily members of the cluster.

When a new client connects to the GCS group that is being used by a **Slony** Π cluster, it is *connected*; when it leaves the group, it is *disconnected*.[5] The GCS delivers group membership changes as part of the same total order that applies to data messages — therefore, we can assume that all nodes see the same set of group membership changes in the same order.

If a connected member of the group completes the handshake process with the existing members of the cluster (see section 8.1), it is has *joined* the cluster.

8.1 Joining the cluster

The first step in joining a cluster is connecting to the cluster's group using the GCS.[6] If the connection attempt is successful, the new node is informed of the members of the group, and the existing members of the group are informed that a new node has connected. The newly-connected

[5]The GCS defines the low-level details of group membership. For example, it might be possible that a node is considered to be disconnected if it does not respond to network messages fast enough. Our assumption is that the group communication system handles this atomically: a node is either a member of the group or it is not.

[6]The mapping of logical cluster names to GCS group names is currently done manually in the configuration file.

node will receive all subsequent messages sent to the group, but because it has not yet joined the **Slony** II cluster, it will not yet apply any writesets (it may choose to buffer them, however).

Next, the newly-connected node checks whether there are any other members of the cluster. If there are not, it must *bootstrap* the cluster, as described in section 8.2. In this section, we assume that the cluster contains at least one other member.

To join a **Slony** II cluster, a newly-connected node must obtain three pieces of information:

1. Meta-data about the cluster itself
2. The current GID
3. The current state of the database

Cluster meta-data is the simplest: the newly-connected node n picks a random member of the group n'. It asks n' (via a private message) to send it the cluster meta-data. If n' disconnects from the group before n receives a response, n merely picks another random member of the group.

Determining the current position of the GID sequence is slightly more complex: as described in section 7.1, the GID of a writeset is not stored in the writeset itself but is implied by the ordering of messages sent to the network. Therefore a new node node cannot request the "current" GID, since that may change by the time the message containing it is received back by the joining node. A simple solution to this problem is to pick a member of the cluster and send a message to the group, asking for that node to send back the GID *of that message itself*. Since the GID request is sent globally, it is part of the total order and has a well-defined GID. The joining node can combine the GID it receives back from the cluster with the stream of messages it reads from the GCS to correctly initialize the GID counter.

Synchronizing the state of the database is the most complex part of joining a cluster, and is fully discussed in section 9. Note that since a node is not synchronized with the cluster until the recovery process is complete, it cannot accept transactions or apply remote writesets.

8.2 Bootstrapping

When a node connects to a group and determines that it is empty, it must then *bootstrap* the cluster. This means that it must initialize the state needed to allow other nodes to join the cluster, and to allow the cluster to begin accepting transactions. Once a single node has bootstrapped the cluster, it should be fully operational (although some of the cluster's members may not yet have joined).

At a minimum, a bootstrapping node must initialize the GID counter to 0. However, this is probably insufficient due to the possibility of network failure. Consider the following scenario:

- Node A joins the group and bootstraps the cluster
- Node B joins the cluster as described in section 8.1
- The cluster processes transactions; suppose that the GID is now 100
- Node A crashes, and is then disconnected from the network due to a network failure of some kind

- When A is restarted, it joins the group and discovers that it is once again empty (since it is disconnected from the network, and hence cannot see B)
- A bootstraps its own cluster and begins accepting transactions
- The GID sequences of A and B are now out of sync. Suppose that A and B both process 100 (different) transactions, leaving their GID at 200. If both nodes now rejoin the same network, they will communicate their current GIDs and incorrectly believe they are synchronized

There are two ways to avoid this problem:

- Prevent A from bootstrapping the cluster a second time. The easiest way to implement this is to require administrator intervention (e.g. a command-line flag to the `postmaster`) to bootstrap a cluster. For some installations this is desirable anyway, but probably not all.
- When A and B rejoin the same network, arrange for them to realize that their GID counters are no longer synchronized. This can be arranged quite simply: as part of the cluster metadata, we can include a *cluster identifier*. When a node bootstraps a new cluster, it generates a new, globally-unique cluster identifier. When a node joins a cluster, it takes that cluster's identifier. During the handshake process, we check whether the node's previous cluster identifier matches the current cluster's identifier; if it does not, the joining node must perform a total recovery (see section 9.2).

We will probably implement both of these techniques.

8.3 Leaving the cluster

A node can disconnect from a **Slony** Π cluster at any time. There is no need for a node to send an explicit "disconnect" message to the group: the GCS will inform the other nodes that the disconnected node has left the group. This applies to both "graceful" disconnections (when **Slony** Π is shutdown on a node intentionally) as well as sudden disconnections caused by node failures.

Once a node has disconnected from the cluster, it cannot commit any transactions, since we would be unable to replicate the changes made by those transactions and resolve any resulting conflicts. Any pending writesets produced by a disconnected node can still be applied according to the normal conflict resolution rules. If the node did not see its own writesets before it disconnected, it will apply them once it rejoins the cluster (see section 9).

8.4 Network partitions

A *network partition* occurs when the network splits into two or more components. The members of each partition can communicate with one another, but not with the members of any other partitions. This can occur in practice: for example, if the nodes in the cluster are located on two different LANs connected by a gateway, the network will partition if the gateway crashes. It is also possible for partitions to *merge* — for example, when the gateway recovers, the two partitions will merge back to a single network.

Network partitions present a problem for **Slony** Π. Because each remains online, clients can still connect to nodes and submit transactions. However, writesets cannot be broadcast to the entire cluster because of the partition, so a global total order cannot be formed. If multiple partitions

continue to accept transactions, the partitions will begin to diverge. If the partitions subsequently merge, **Slony** Π will be unable to synchronize all the nodes to a consistent state.

We assume that detection of network partitions is provided by the GCS. When a partition occurs, the GCS delivers a message in the total order to the nodes in each partition. This message describes how the cluster has been partitioned (i.e. how many partitions have formed and which nodes belong to each partition). **Slony** Π now has two options:

- the partition can be treated as temporary: **Slony** Π assumes that the nodes will eventually be merged back into a single network. One partition is selected to be the *primary* partition and is allowed to continue processing transactions. All other transactions are considered *secondary*, and cannot allow transactions that modify SQL data to commit.[7] When the partitions merge, all the nodes can be synchronized correctly: the nodes in secondary partitions will replay the transactions executed by the primary partition using the recovery mechanism described in 9.

  All the nodes must make the same decision about which partition is chosen to be the primary one. To allow this, the decision should be deterministic function of information that is available to every node. One reasonable way to make this decision is to make the primary partition the one with the most members. If two partitions have the same number of members, the tie can be broken by some other deterministic criteria (for example, by selecting the partition containing the node that joined the cluster most recently).

- the partition can be accepted as permanent. The partitions are treated as unrelated **Slony** Π clusters which can each accept and commit transactions. However, just like any other set of arbitrary **Slony** Π clusters, they cannot be subsequently merged together to form a single cluster.

Whether a network partition is considered to be temporary or permanent could depend on the configuration of **Slony** Π, administrative intervention, or other factors.

9 Recovery

When a node attempts to join a **Slony** Π cluster (as described in section 8.1), it must synchronize the content of its local database with the current replicated database. This process is called *recovery*. We provide two related recovery mechanisms: *partial recovery* and *total recovery*.

9.1 Partial recovery

This recovery method is used when the joining node was a member of the cluster in the recent past. Therefore, the local state on the joining node is probably similar to the current state of the cluster, so we only want to transfer the differences between the two. This can be achieved by reusing the totally-ordered writeset sequence produced by the GCS: by applying the writesets that were generated while the node was disconnected, the node can be synchronized with the current state of the cluster. Note that there is no need for any additional conflict resolution: nodes are not allowed to commit transactions unless they are members of the **Slony** Π cluster, so exactly the same sequence of transactions will be applied to the joining node as was applied by the other members of the cluster.

[7] They can continue to accept read-only transactions, however.

To perform partial recovery, we require two pieces of information:

- the GID of the last writeset that was committed at the recovering node. This will be recorded in the Write Ahead Log, as described in section 9.4.
- each writeset that falls between that GID and the current GID of the cluster.

In order to perform partial recoveries, **Slony** II archives a certain number of writesets on disk at each node in the *writeset log* (described in section 9.5). Partial recovery can be used if the writeset log on any machine contains enough writesets to synchronize the node with the current state of the cluster. If this is not the case, total recovery must be used.

9.2 Total recovery

This recovery method is used when the joining node is quite dissimilar from the current state of the cluster. This might be because the node has never been a member of the cluster before, or it might be that the node has been disconnected from the cluster for a considerable period of time (days or weeks). Total recovery is an expensive operation, and should not be performed frequently — the writeset logs on each node should be large enough that most joining nodes can use partial recoveries to synchronize their state.

In a total recovery, the recovering node is sent a compressed snapshot of the current state of the database at one node in the cluster, as well as the GID that the snapshot corresponds with. The recovering node basically replaces its own database content, if any, with the snapshot and sets its current GID to the GID of the snapshot. Then it performs a partial recovery to synchronize itself with the current state of the cluster, if needed.

In practice, this might be implemented by creating the snapshot using `pg_dump`, along with some additional information (including global data such as users and databases and the GID of the snapshot). Due to the size of many production databases, we anticipate the need for users to transfer these snapshots via offline means (such as tape drives). This should be possible.

9.3 Recovery algorithm

Recovery is performed as part of the handshake processed described in section 8.1. Once a node has obtained the cluster configuration and the current cluster GID, it can perform the following steps to synchronize its database with the current state of the cluster:

1. determine the GID of the last committed writeset at the local node, g_0 (see section 9.4)
2. send a message to the cluster, asking each node to reply with the GID range of archived writesets available at that node
3. if one or more nodes contains all writesets greater than g_0, we can use partial recovery; otherwise, we need to use total recovery
4. select a *recovery partner*. This is a member of the cluster that will be used to update the joining node's database. The method for selecting the recovery partner has not been decided — it may be random, based on configuration, or perhaps based on some autonomic mechanism for selecting the "least busy" node.

5. ask the recovery partner for the information needed by the recovery mechanism. For partial recovery, this would be a range of writesets; for total recovery, this is a snapshot of database content and an associated GID.

   Note that from the perspective of the recovery partner, this is a stateless protocol: the joining node selects a partner and asks it for information. The recovery partner merely returns the requested information. The state of the recovery process is recorded only on the joining node.

Any writesets that the recovering node receives during this process should be buffered, and applied (using the normal conflict resolution rules) once recovery is complete. The node can then begin accepting transactions.

9.4 Integration with Write Ahead Logging

In **PostgreSQL**, the Write Ahead Log (WAL) is used to record modifications to the database. To atomically commit a transaction, a record is written to the WAL describing the changes made by that transaction, and then the WAL is flushed to disk. If the machine crashes before the changes made by the transaction are written to disk, the WAL is consulted to replay the changes and restore the database to a consistent state.

As part of the recovery process, **Slony** II must determine the GID of the last writeset to be applied at the recovering node. This information must be correct even in the face of unexpected node crashes, such as power failures. The easiest way to implement this is by modifying the WAL to include the GID associated with each committed transaction, if any. By "piggybacking" on the existing disk flush that is performed for the WAL, we can record this information durably without the need to flush another file to disk at commit-time.

9.5 Writeset log

As discussed in section 9.1, each node in the cluster may archive the writesets it receives to a *writeset log*, to allow partial recoveries to be performed.

The writeset log will be an on-disk ring buffer. The size of the buffer will be supplied by the replication manager and will be user configurable. The buffer should probably be broken up into separate files, the size of which is a compile time constant (perhaps 16MB by default).

For performance reasons, writesets will be written as they are received by the replication manager. Each database being replicated will have its own writeset log, to avoid unnecessary contention.

9.5.1 Writeset durability

The writeset log is used to recover the state of *other* nodes after they have crashed. Because the writeset log is not needed to recover the local node, it does not need to be flushed to disk as part of the transaction commit process (unlike the WAL). Avoiding an additional forced disk write for every committed transaction is essential for good performance.

9.5.2 Validating writesets

Given section 9.5.1, we cannot assume that the content of the WS log is valid after a crash has occurred. If a power failure occurs while we are writing out a page in the writeset log (a "torn page"), the content of the WS log might be incomplete or corrupt. Unless this situation is detected,

we might serve corrupted content to other nodes if a node with a torn page is selected as a recovery partner.

This problem can be resolved in several ways. A checksum (e.g. CRC32) could be recorded on each page of the writeset log before writing it to disk. Before using a page from the writeset log, the replication manager could verify that the checksum is correct. Alternatively, the entire content of the writeset log could be discarded when a node is restarted; since the node may have been disconnected for a considerable period of time, the content of the formerly crashed node's writeset log may be of little relevance anyway. In any case, this is unlikely to present a major problem in practice: the writeset log contains highly structured data, so a torn page is likely to result in a syntax error rather than silent data corruption.

9.5.3 Locking

There are times when concurrent access to the writeset log will be required — for example, when a peer node is assisting a recovering node, it will be reading its own log and writing to it. A read/write locking mechanism should suffice.

The granularity of locking presents a bit of a problem. It would be pretty simple to implement file-level locking; more granular locking may offer better performance, but will be significantly more complex to implement. This is a point for further research.

9.5.4 Log file reuse

Since this is a circular buffer, files will be reused. When a writer needs to reuse a log file, all contents are lost. That is, the old data in the file, which is effectively the data at the end of the buffer, cannot be used for recovery since it may have been partially overwritten, etc.

10 DDL statements

SQL statements can be divided into several groups based on the type of operation they perform. The operations that modify SQL data (such as `INSERT`, `UPDATE`, and `DELETE`) are called the Data Modification Language (DML). The operations that change the definition of SQL objects (such as `CREATE TABLE`) are called the Data Definition Language (DDL).

Most of the research on database replication only deal with DML, as it constitutes the bulk of typical database workloads. We feel that implementing support for replication of DDL statements in **Slony** II will be useful — in practice, production databases frequently require schema changes while the database is deployed.

The concurrency control techniques discussed in section 6 are insufficient to replicate DDL statements. That approach to dealing with DML is based on the assumption that two-phase commit would be too expensive, due to its poor scalability and the rate at which DML statements are submitted in many applications. However, these conditions do not apply to DDL statements: in a well-designed application, the definitions of database objects should not need to be changed frequently. Therefore, we can afford to use two-phase commit to replicate DDL statements.

10.1 DDL in transaction blocks

PostgreSQL supports DDL in transaction blocks. This means that a transaction can contain a mix of DML and DDL statements. To support this in **Slony** Π, we must do one of the following:

1. Start a transaction block on each node, issue the DML statement inside the transaction block on each node and then commit using two-phase commit
2. Group DDL commands into the existing writeset structure but flag the whole writeset as needing to be processed in two-phase commit mode
3. Not allow DDL inside transaction blocks

Option 2 is the simplest to implement but option 1 is also possible to implement. Option 3 is the least attractive.

10.2 Which DDL is replicated

Not all DDL can or should be replicated. Some DDL only applies to the local node (such as `CREATE TEMP TABLE`), so it does not make sense to replicate it. Replicating some other DDL statements, such as `VACUUM` or `ANALYZE`, is probably not wise: it would significantly affect the throughput of the cluster and provide little benefit.[8]

11 Group Communication Systems

As discussed above, the Group Communication System plays a fundamental role in **Slony** Π. Therefore, the quality and performance of the GCS we use will significantly affect the quality and performance of **Slony** Π itself. For the implementation of the prototype, we chose to use the open-source Spread GCS system developed at the Center for Networking and Distributed Systems at John Hopkins University [ADS00].

We have begun auditing the Spread source code to assess its suitability for **Slony** Π. We have found a number of performance and scalability related issues but no bugs. We intend on integrating these fixes into the publicly available code and encouraging a community of **Slony** Π enthusiasts to help maintain Spread. As yet, we have found no bugs which would undermine the correctness of the replication mechanism. It must be said that we have taken only a cursory look at the GCS protocol handling code, however.

We have not yet decided whether to use Spread in the production implementation, switch to another GCS, or write our own GCS from scratch.

12 Implementation issues

12.1 GID storage

As described in section 7, every writeset has an associated GID (a 64-bit integer). When applying a writeset at a node, we are often interested in the GID of the writeset that last modified a particular

[8] It would be useful to allow administrators to schedule the execution of a shell command or maintenance DDL on some or all nodes of the cluster simultaneously. This can also be done with many existing tools for managing Unix clusters.

row. That is, we want to find the GID of the writeset that created or invalidated a particular tuple version last. Therefore, an important consideration in the design of **Slony** Π is how the GID associated with a particular tuple version will be stored and how that storage will be managed.

We considered two main alternatives for storing GIDs:

1. Storing GIDs in an auxiliary data structure that maps XIDs to GIDs. Since the heap tuple header already contains the XID of the creating and invalidating transactions of a particular tuple version, we can use this data structure to find the GIDs associated with those transactions.

   There are two obvious ways to implement the lookup table:

   (a) As a system catalog. This is the technique used by [KA00], but we feel this solution is not appropriate for a production environment. Inserting into a system catalog is expensive: it requires doing both a heap update and updating at least one index. This technique also creates additional overhead because of the need to `VACUUM` the system catalogs to reclaim expired tuples.

   (b) As a hash table in shared memory. This is more plausible, but it raises a number of difficult issues. The hash table would need to be updated whenever a GID is assigned to an XID, which happens quite often (e.g. when applying remote writesets, or when a local transaction derives its GID from the total order). Therefore, updates to the hash table would likely become a source of lock contention.

   Another problem would be how to bound the size of the hash table. Shared memory is statically sized; furthermore, it is undesirable to store a hash table entry for each active XID, as this could consume an enormous amount of memory. A reasonable compromise might be some kind of hybrid scheme: we could keep an entry in the hash table for each uncommitted transaction — if a transaction has committed, we know that it must precede a currently-running transaction in the total order, and therefore we need not know its GID.

   This also raises the question of whether the XID to GID map needs to be persisted across system crashes.

2. Storing GIDs in the heap tuple header. This would avoid the possible lock contention associated with an auxiliary lookup table, and would nicely resolve questions of crash recovery (since the GID would be part of each heap tuple, it would be persisted in the case of a crash through the usual WAL mechanism).

   One downside is that it would require changing the heap tuple format. Introducing a single GID would add 8 bytes to the tuple header; it may be necessary to store two GIDs, in which case 16 additional bytes would be required. Furthermore, this would make migration of **PostgreSQL** systems to **Slony** Π more difficult.

   An additional consideration is that this technique would require updating heap tuples twice when executing a transaction on its originating node: once, to perform the update itself, and a second - time to set the GID on the modified tuples (the GID cannot be set when the update is performed since it is derived from the total order, and this information is only available when the completed writeset is sent to the network). This could be done reasonably cheaply,

as the originating node would know the TID of the tuples it has modified, but it would still be an additional cost.[9]

12.2 Concurrent application of writesets

One of the significant shortcomings of the mechanism described in [WK05] is that it assumes serial application of writesets at any given site. Given that our chief scaling mechanism in **PostgreSQL** and **Slony** II is parallelism, serialising this part of the algorithm is not acceptable.

It seems that if we accept that we must ship GIDs with writesets, then we have enough information at all sites to determine if a tuple is the same tuple we saw at our originating site; or, if it is different and the modification is made before or after us in the total order.

Now, a simple comparison will tell us if the modification is in the past or future with respect to the total order time line. If the modification is earlier in the total order, we conflict and must abort. If the modification is later in the total order then that transaction must be aborted. This can and must be evaluated in the same way at all sites.

We must introduce one other mechanism to ensure that this works: a replication backend cannot commit until all writesets which come before it in the total order have committed. This requirement solves two problems: firstly, it ensures that all potentially conflicting writesets will still be in progress at the time the conflict must be resolved (i.e. when a backend must be killed); secondly, it ensures we do not break the SI serialisation rules, by ensuring that modifications become visible to the system in total order.

This mechanism is theoretical but will be prototyped in the future.

12.3 Aborting conflicting transactions

Write-write conflicts are handled by aborting one the conflicting transaction that is is latest in the total order. There are two specific circumstances in which an abort is required:

1. When applying a writeset at a remote site that conflicts with an uncommitted local transaction[10]

2. When applying a remote writeset that conflicts with another remote writeset, and the other writeset follows this writeset in the total order[11]

For our present purposes, these two cases are the same: we need to abort the current transaction of another backend. This will be implemented similarly to the existing code for cancelling running queries: to abort a backend's transaction, a Unix signal is sent to that backend. Since all the available signals are currently being used in **PostgreSQL** for other purposes, we will also add some information to the backend's `PGPROC` entry. The backend periodically polls to see if it has received a signal instructing it to abort its transaction; when it notices that this has occurred, it will abort itself when it is safe to do so.

[9] A subtle point is that since no lock will be held on the heap page containing a modified tuple, a concurrent lazy `VACUUM` could change the TIDs on the page between the time the update is performed and the transaction is committed. However, lazy `VACUUM` is guaranteed not to move the tuple off the page, so it should still be cheap to locate via a sequential scan of the page.

[10] By "local transaction", we mean a transaction that is executing on its originating node and has not yet broadcast its writeset to the network.

[11] As described in section 12.2, this can only occur if remote writesets are applied in parallel.

12.4 Sequences

Consider the following two transactions:

```
T1: BEGIN
T1: INSERT INTO t1 VALUES (nextval('foo_seq'));
T2: BEGIN
T2: INSERT INTO t1 VALUES (nextval('foo_seq'));
T1: COMMIT
T2: COMMIT
```

If these transactions were executed concurrently on a single node, the locking performed internally by `nextval` would ensure that both transactions would receive distinct sequence values. Therefore they would not conflict with one another, and both transactions could commit. In the replication algorithm we have discussed, a transaction's writeset is only sent to the network when it commits, so there is no opportunity to perform synchronization between `nextval` calls. In this case this would result in a spurious conflict between these two transactions.

The conflict could be avoided by adding synchronization for `nextval` calls, but this is undesirable: sequence generators are frequently used, and synchronization over the network is extremely expensive.

An alternative is to redefine the semantics of sequence generators.

The intended use of sequences is as a means of number generator. They are generally employed to produce an arbitrary key for a table. There is a perception that sequences must be monotonically increasing. The behaviour implemented by **PostgreSQL** and required the SQL standard is the numbers be successive. That is, three increments of the sequence produce three numbers n, o, p such that $n < o < p$. Implementing this in **Slony** Π is difficult.

12.5 Partial replication

12.6 Security

12.7 Large field values

In **PostgreSQL**, most field values are stored "inline" (as part the heap pages of the table), but fields above a certain size are "TOASTed" and moved to out-of-line storage. It would be a useful enhancement to be able to detect when an `UPDATE` does not modify a field, so that we could avoid resending the value of that field over the network. This is particularly important when dealing with large TOAST fields (as their size can be up to 1GB).

12.8 Roles

Roles present a problem as we must decide how and when to replicate users and groups to the cluster. There are two reasons this is non-trivial:

1. We want to avoid modifying **PostgreSQL** SQL syntax where possible
2. Even though a table may be replicated, an administrator may want to restrict access for a single user to a single node.

At the present time, there does not seem to be any way to implement both of these requirements. One idea would be to add an extra parameter to `CREATE ROLE` and `ALTER ROLE` to indicate whether a role should be replicated. If the role is replicated, operations affecting the role (such as granting permissions to the role or revoking them from the role) should also be replicated.

13 Prototype

We have developed a prototype of the replication system described by this report. It is intended to be a "proof of concept": it implements the algorithms described above, but is not appropriate for use in a production environment. The prototype is implemented as a series of modifications and additions to a recent snapshot of the development version of PostgreSQL. The prototype is written in Ruby, a dynamically-typed, object-oriented programming language [Mat02].

13.1 Ruby backend API

In order to implement the prototype in Ruby, we had to allow some parts of the internal **PostgreSQL** APIs to be accessed from code written in Ruby. This was done using C and the Ruby extension API to produce a shared library. This shared library is loaded by **PostgreSQL** at backend startup, and allows any Ruby code that uses the library to call into certain parts of the backend.

13.2 Writeset capture

As a transaction executes, the prototype needs to collect the tuples modified by the transaction so that they can later be serialized into a writeset and broadcast over the network. This is done using the Ruby backend API, as well as some modifications to **PostgreSQL** itself. Every time a row is modified by a backend, a Ruby callback is invoked (and passed the modified tuple). The callback saves a copy of the tuple. At the end of the transaction, the set of modified tuples is sent to the replication manager.

13.3 Replication manager

The replication manager is implemented as a separate process that is forked by the `postmaster` when **PostgreSQL** it started. When the replication manager starts, it performs the following tasks:

1. spawns a set of *replication backends* that are used to apply writesets produced by other nodes (see section 13.4)
2. connect to the GCS
3. begin logging writesets (see section 9.5)
4. perform the "handshake" required to join the cluster (see section 8.1)

The local node has now joined the cluster. The replication manager then enters the "main loop", in which it waits for events on a set of IO handles. The replication manager is essentially a state machine that is driven by input from three different sources:

- messages received from the GCS. These can contain several types of data:
  - replicated writesets produced by a node in the cluster.
  - group configuration changes. These describe nodes joining and leaving the GCS group, as well as network partitions and similar events.
  - handshake and recovery requests sent by other members of the group.
- incoming writesets delivered by local backends. These contain writesets generated at the local node that should be broadcast to the rest of the cluster via the GCS.
- metadata produced by one of the replication backends. When a writeset is delivered to a replication backend, it might commit or abort that writeset, depending on whether it conflicts with another writeset.

13.4 Replication backend

A replication backend is a modified **PostgreSQL** backend that is used to apply writesets produced by other nodes in the cluster. Replication backends are forked from the `postmaster`, like normal `postgres` backends, but they are controlled by the replication manager directly and use a modified version of the frontend-backend protocol.

14 Testing

In general, testing concurrent software is a challenging task. This rule will apply to **Slony** II; not only is it highly concurrent, but it also has a number of obscure failure conditions that will need to be separately tested and verified (for example, ensuring correct behavior when a network partition occurs).

To reduce the complexity of the testing and to help verify the software in a wider range of network conditions, "network simulation" software such as [Hem] will be helpful. This software allows the creation of unusual network conditions: network partitions, heavy packet loss, node disconnects, and so forth.

Another useful tool for testing **Slony** II will be the ability to create and destroy network nodes with ease, and to arrange these nodes into arbitrary network configurations. For this task, a processor emulator / virtual machine such as [Bel] will be helpful. We plan to create a ready-made Qemu image that contains a minimal Linux installation, Slony II, Spread, and any other required software. Using this, we will be able to simply create n virtual machines into a **Slony** II cluster with n nodes.

15 Remaining work

There are four main tasks which need to be finished in our prototype:

- Application of writesets following conflict semantics detailed in [WK05]
- Finish interface to transaction abort signalling
- Testing for deficiencies and scalability of replication mechanism

- Integrate recovery into the prototype replication manager

References

[ADS00] Yair Amir, Claudiu Danilov, and Jonathan Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and Networks*, pages 327–336, Washington, DC, USA, 2000. IEEE Computer Society.

[Bel] Fabrice Bellard. QEMU processor emulator. `http://fabrice.bellard.free.fr/qemu/`.

[BHG87] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[CKV01] Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, 2001.

[GHOS96] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, June 1996.

[Hem] Stephen Hemminger. Netem network emulator. `http://developer.osdl.org/shemminger/netem/`.

[KA00] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *The VLDB Journal*, pages 134–143, 2000.

[KBB01] Bettina Kemme, Alberto Bartoli, and Ö. Babaoğlu. Online reconfiguration in replicated databases based on group communication. In *DSN '01: Proceedings of the 2001 International Conference on Dependable Systems and Networks*, pages 117–130, Washington, DC, USA, 2001. IEEE Computer Society.

[Mat02] Y. Matsumoto. *The Ruby Programming Language.* Addison Wesley Professional, 2002.

[WK05] Shuqing Wu and Bettina Kemme. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, pages 422–433, Washington, DC, USA, 2005. IEEE Computer Society.

A Replicated DDL Statements

As discussed in section 10.2, it would be unwise for **Slony** Π to blindly attempt to replicate every SQL statement. The following DDL statements could reasonably be replicated:

The following DDL must be replicated:

ALTER AGGREGATE	ALTER CONVERSION	ALTER DOMAIN
ALTER FUNCTION	ALTER INDEX	ALTER LANGUAGE
ALTER OPERATOR	ALTER OPERATOR CLASS	ALTER SCHEMA
ALTER SEQUENCE	ALTER TABLE	ALTER TRIGGER
ALTER TYPE	COMMENT	CREATE AGGREGATE
CREATE CAST	CREATE CONSTRAINT TRIGGER	CREATE CONVERSION
CREATE DOMAIN	CREATE INDEX	CREATE LANGUAGE
CREATE OPERATOR	CREATE OPERATOR CLASS	CREATE RULE
CREATE SCHEMA	CREATE TABLE AS	CREATE TABLE
CREATE TRIGGER	CREATE TYPE	CREATE VIEW
DROP AGGREGATE	DROP CAST	DROP CONVERSION
DROP DOMAIN	DROP FUNCTION	DROP INDEX
DROP LANGUAGE	DROP OPERATOR	DROP OPERATOR CLASS
DROP RULE	DROP SCHEMA	DROP SEQUENCE
DROP TABLE	DROP TRIGGER	DROP TYPE
DROP VIEW	TRUNCATE	

The following DDL can be replicated, with some important limitations or caveats:

SQL statement	**Comments**
CREATE FUNCTION	Depending on the language, this may or may not be replicated. Replicating the creation of a C-language function (where the CREATE FUNCTION command only gives the location of the shared object file) will not be possible, but replicating most other procedural language definitions should be.

The following DDL cannot or should not be replicated:

SQL statement	**Comments**
ANALYZE	
ALTER DATABASE	
ALTER TABLESPACE	
CREATE DATABASE	
CREATE TABLESPACE	
DROP DATABASE	
DROP TABLESPACE	
REINDEX	
VACUUM	