Advanced Topics of Information Systems - Database Replication
Andreas Dröscher, 04-280-301, autumn 2010

Abstract

Data warehousing is one of the key functionalities of today’s information systems. Nearly every enterprise application uses some kind of database. However for correct and efficient operation every application relays on different key features of the database management system.

Since publication of The Dangers of Replication and a Solution [GHOS96] in 1996 many of the use cases considered may have changed. Even if main statements of this paper remain valid, the conclusion to favor lazy over eager replication isn't valid anymore as Don't Be Lazy, Be Consistent: Postgres-R [PG-R00] has proven.

This report is going to give a short overview of possible database replication concepts including a detailed look at the proof of concept implementation in Postgres-R. As the mathematical concepts presented in [GHOS96] can be considered common knowledge, the derivation and the formulae are not recited.

Introduction

Database replication is used for two key purposes: redundancy and scalability. While redundancy is easy comprehensible - if one server fails the business continues (seamlessly) on the backup system - scalability has much more pitfalls. Read transactions scale well as each server of a cluster may answer any request without creating load on any other server. However write transactions involve every server as all replicas of the involved datasets need to be updated (transaction*server count) or O(n²). Jim Gray and others [GHOS96] have proven that the conflict potential even increases exponentially: "A ten-fold increase in servers gives a thousand-fold increase in failed transactions". To circumvent this situation they suggest allowing cutbacks at the correctness and consistency rules of a database system. In other words they favor lazy replication enhanced with non-transactional replication schemes to form a two-tier replication concept.

The two basic replication models are "eager" and "lazy". Based on a paper from 1996 [GHOS96] most of the research has focused on optimizing lazy replication (stated in [PG-R00]). Also most of the commercial implementation e.g. Oracle 7 [ORACLE] (using a twelve rules based reconciliation concept) were based on lazy replication.

About ten years ago there was a shift back to eager replication. This trend is driven by techniques based on Broadcast Primitives in Replicated Databases [AES98] used in the field of distributed systems. By using broadcast primitives the concurrency control is "outsourced" into lower network layers (ISO/OSI 1-6) and is implemented much more efficient as it could have been done in the application layer (ISO/OSI 7).

In the first part of the paper I will explain and exemplify the basic replication methods and their disadvantage in daily use (based on [GHOS96]). Later on I will show an approach first theoretically [AES98, KA98] and then practically [PG-R00] to deal with the inconvenience of eager replication. To conclude I will look at both papers censoriously and try to estimate the benefit for a real world usage.
Use case “New York Marathon, 2007” [LM08/09]

An application for a performance focused distributed database is the New York Marathon. During the Marathon (a few hours) the webpage got approximately 15 million hits. Simultaneously the database was fed with several intermediate- and final timings of the approximate 38000 participants.

The solution for this application consisted of ten servers (2 load balancers, 6 application servers and 2 primary database servers). The load balancers were used to equally distribute the user queries over the application servers which answered the requests from their local, replicated (slave) database.

![System Architecture](image)

The two primary database servers distributed all incoming time measurements to the application servers. The setup showed that the first approach using Statement-based Replication of MySQL 5.0 did not scale. The second approach using Mixed Replication (MySQL 5.1) did perform well. In the mixed model the database server dynamically switches between statement and row based replication based on a variety of criteria [MYSQL08]. As an additional tweak MySQL’s AutoCommit feature was disabled to postpone the intermediate replication of partial commits between the two master servers. With this option the replication is delayed until the end of the transaction.

There will never be a database replication approach that will suit all possible application scenarios. This use case shows that extreme scenarios have to be analyzed carefully and an appropriate system setup has to be chosen for it. In this example the data has a well defined source producing the updates/inserts and well defined consumers querying the database. Also the amount of queries is approximately fifty times higher as the update count. So the slave databases can be designed for read only operations and a one way replication scheme can be used.

Database basics

One of the key functionality of a database system is assuring data integrity. Every database write should be an atomic action:

1. lock all involved resources - preferably atomically
2. (read) → modify (and check constraints) → write datasets
3. unlock resources.

While such transactions are feasible in a single server environment the whole transaction management is becoming very complex in a multi server environment (Figure 2). One approach is to extend the locking mechanics beyond one server and update the master/slave servers in a single...
transaction. This is called *eager replication*. The other approach is to update the master in a single transaction and propagate the updates to every slave during additional independent transaction, this is called *lazy replication*.

![Figure 2, Replication: single-node vs. multi-node [GHOS96]](image)

**Eager Replication**

As stated earlier eager replication keeps all copies of a server cluster synchronized. Every single transaction is propagated synchronously over all members. Based on this eager replication has the major advantage that there are no concurrency anomalies and no need for reconciliation [GHOS96].

But the paper from ‘96 stated that there are at least three big disadvantages of eager replication. They believed, as every server is involved in every transaction O(server*transaction), that this transactional scheme could not scale. The second disadvantage they listed was the high deadlock rate if transactions were initiated from different servers (multi-master). They thought this would require complex back off algorithms.

Both statements have been proven wrong.

The only still existing disadvantage of eager replication is the need for additional replication concepts to handle offline nodes which synchronize their data rarely (Use case: road warrior).

**Lazy Replication**

Lazy Replication is the completely contrary approach. Every transaction is first performed locally and then propagated to the additional servers in independent transactions. This concept first of all allows offline clients, which would propagate all changes made to the servers after coming online again. Lazy Replication however carries a high risk for *data inconstancy* [GHOS96]. A query executed on two different servers could lead to different results if it is done between the synchronization.

It gets even worse if a entry is modified independently on two different servers simultaneously, as the conflict will only be detected during the propagation. This will lead to the need of a routine or a person for reconciliation of the conflict. An example of such a problem is a bank account: If two withdraws are made simultaneously on two different servers both would show a suitable balance. However during the synchronization period a constraint would block the database update as the account is overdrawn.
In addition to breaking constraints there is also a high risk of system delusion - the database is inconsistent and there is no obvious way to repair it [GHOS96]. This is likely to occur in an environment with group ownership, often called update-anywhere (Figure 3), where every server updates its data and replicates it to the additional members of the cluster. This is in contrast to the master ownership concept where always one server is responsible for holding the primary copy and all replicas are only read.

![Figure 3, Ownership: Group vs. Master [GHOS96].](image)

While in lazy systems the single master replication scheme is used to reduce the risk of system delusion in eager-master replication has its main advantage is the reduction of deadlocks.

**Non-Transactional Replication Schemes**

Non-transactional replication schemes are following another very interesting approach as they are not saving the value itself but the transformation. As an example we can take the iPhone’s calendar application, it saves all manipulations to the calendar and adds timestamps to them, i.e. move appointment X to 10 a.m., appointment Y deleted, appointment Z added. This way the mobile calendar and the office calendar can synchronize properly. When the two devices become connected to each other, the two databases converge.

**Two-Tier Replication**

An ideal replication scheme would achieve four goals [GHOS96]:

- **Availability and scalability:** Provide high availability and scalability through replication while avoiding instability
- **Mobility:** Allow mobile nodes to read and update the database while disconnected from the network
- **Serializability:** Provide single-copy serializable transaction execution
- **Convergence:** Provide convergence to avoid system delusion

The safest transactional replication schemes are eager- or lazy- master systems. These two are the only systems guaranteeing ACID (atomicity, consistency, isolation, durability). All of them however had other disadvantages [GHOS96]:

1. Master objects cannot accept updates if master node is not reachable
2. With increasing load or node count deadlock rates rise quickly
3. With the introduction of master copies also a single point of failure is introduced

Back in '96 the many disadvantages of eager and lazy systems lead to the new approach of the two-tier replication. On one hand it provides high availability and scalability through replication while avoiding instability. Additionally mobile nodes have the possibility to update the database while
being disconnected (something not yet solved with the Group Communication Approach (GCP), [PG-R00] and others).

To achieve this, the two-tier replication scheme assumes two kinds of nodes [GHOS96]:

**Mobile nodes:** disconnected most of the time
- have a local replica of the database
- may originate tentative transactions

**Base nodes:** always connected
- store a replica of the database
- most items are mastered at base nodes

and that mobile nodes may hold two versions of the data [GHOS96]:

**Master version:**
- the most recent value received from the object master
- the version at the object master is the master version however disconnected or lazy replicated nodes may have older versions

**Tentative version:**
- the local object may be updated by tentative transactions the most recent value, due to local updates, is maintained as tentative value

this also leads to two types of transactions [GHOS96]:

**Base Transactions**
- work only on master data and produce new master data

**Tentative Transactions**
- work on local tentative data and produce new tentative versions
- additionally they produce a base transaction to be run on the base nodes at a later time

The basic idea behind the two-tier replication is that when a node is offline it may involve objects mastered on base nodes and mastered at the mobile node. When the mobile node is connected to the network all base nodes will be available and all tentative transactions are processed a second time as base transactions. In this second transaction all master copies will be updated. If during this ("real" transaction) a transaction fails due to unmet constraints the user has the opportunity to reconcile the changes. Within this phase the mobile node also updates all its master versions of the base nodes.

In a cluster environment with two-tier replication between two servers one server would see himself as a mobile node holding some master versions and the other server as base node for other master versions (and vice versa). The system would behave like a lazy master replication with the additional restriction that no transaction may update data mastered at more than one "mobile" node.

This two tier transaction scheme meets most of the four rules mentioned before. This scheme however exploits like the most "lazy" schemes the possibility that a query may have different results on different nodes. As however the merging phase is obvious to the user it also allows to yell for user intervention in a case of inconsistency, the user can easily be included in the reconciliation process.

Clearly if this protocol presented in [GHOS96] is feasible is HIGHLY application specific. It could work for an SBB conductor or ticket vending machines, but not very well for online booking of flights.
Influence of Group Communication Primitives

Since the publication of [GHOS96] there has been a strong believe that eager protocols were not feasible due to bad scalability. Scared from deadlocks database developers were willing to accept inconsistencies and complex reconciliation processes to circumvent eager protocols. With the introduction of Broadcast Primitives in Replicated Databases ([AES98] and others) a possibility to successfully implement eager replication has been found.

Generally it can be stated that, as weaker as the ordering requirements are, as more efficient the group communication protocol. However as weaker the ordering (of the broadcast primitives) requirements are, more has to be implemented in the replication protocol.

The precondition for the following protocols is that the communication between two nodes is FIFO. If one node sends several messages to another node the messages will arrive in the same order as they were sent.

Reliable broadcast protocol

A reliable broadcast protocol is a simple broadcast essentially ensuring delivery to all sites. A reliable broadcast protocol must have the following three properties [AES98]:

1. **Validity:** If a correct process broadcasts a message m, then all correct processes eventually deliver m
2. **Agreement:** If a correct process delivers a message m, then all correct processes eventually deliver m.
3. **Integrity:** For any message m every correct process delivers m at most once, and only if m was previously broadcasted by the sender of m.

As the reliable broadcast protocol ensures that all messages are delivered only the commit has to be synchronized between the nodes. The initiating node of consecutive writes does not have to wait for all sites to acknowledge each write command. They can be sent out in a non-blocking manner. This is achieved by moving the needed guarantees from the application layer (ISO/OSI 7) to the protocol layer (ISO/OSI 1-6). However it requires a two-phase commitment protocol to ensure correctness. This protocol is based on the two-phase locking concept: First acquire read locks, then write locks and hold all locks until the termination of the transaction.

1. All read operations are executed locally.

   All locks are kept to detect conflicting transactions.

2. All writes are broadcasted to all sites. As the protocol guarantees delivery, several write operation can be sent side by side without blocking.

   Write locks are required and kept to detect conflicting transactions.

3. The two phase commit protocol is executed:
   a) The initiating node broadcasts a commit request.
      i) If any of the servers has any conflicting/blocking write or read transaction it will broadcast an abort message that leads to abortion of the transaction on all nodes.
      ii) Otherwise every server has to broadcast a positive acknowledge.
   b) After all servers have sent an positive acknowledge and clearly also received (n to n) The transaction is committed and all locks are freed.
While this protocol works flawlessly, when all write operations originate from one server there may be conflicting transactions, if write operations are initiated from different sites. These conflicts are only detected at commit time.

The write order is still ensured because all write operation are broadcasted to all sites and concurrent conflicting writes that are delivered in different order at different sites are eventually aborted by the purposed two phase commit protocol. Additionally the use of two phase locking and the fact, that write operations are executed at all sites, ensure that all executions are one-copy-serializable.

**Casual broadcast protocol**

Two events are casually related if they are both executed on the same node or if one of them is the sending of the message and the other is the delivery of the same message. The casual broadcast protocol must have all properties of the RBP and additionally following [AES98]:

4. **Casual delivery:** If the broadcast of a message m casually precedes the broadcast of another message m', then no correct process delivers m' unless it has previously delivered m.

The casual relation of two messages can be exploited to eliminate the need for positive acknowledgements during commitment by exploiting the given information as follows [AES98]. If a node receives casually dependent messages on a commitment of a transaction from all the sites in the network then the receiving node can commit the transaction. On the other hand if the node receives a negative acknowledgement the transition is aborted. Note: This protocol can only be implemented if the causality information is exported by the communication layer.

**Atomic broadcast protocol**

It ensures that all messages are delivered in the same order at all sites (total ordering). The atomic broadcast protocol must have all properties of the CBP and additionally following [AES98]:

5. **Ordered delivery:** If sites p and q deliver broadcast messages m and m' then m and m' are delivered in the same order at all sites.

As this protocol ensures that all messages are delivered in the same order, this can be exploited to implement transactions completely with local commits. There is no need for any confirmation or abort messages.

As however conflicting read operations happen only locally such conflicts - in contrast to the two examples shown above - would not have been detected globally anymore. The best approach to circumvent this problem is to reorder the transaction priorities and allow write transactions to abort read transactions. This way conflicts will not lead to inconsistencies and also no additional synchronization systems have to be introduced.

With atomic broadcasts (offering total order) the full synchronization between two servers can be outsourced into the broadcast protocol to form an efficient and stable replication protocol. We will review an approach based on this idea using [KA98] and [PG-R00] closer.
**Postgres-R - A Proof of Concept Implementation ...**

Postgres-R shows that an eager replication protocol can be implemented in practice. Even so the protocol used for the proof of concept implementation looks similar to the *Atomic broadcast protocol* additionally features like scalable write operations were taken into consideration.

The transaction is executed on a shadow copy and intermediate updates are kept local. This "private" execution moreover allows checking constraints, capturing write-read dependencies and processing triggers. If the shadow copy write does not yell any errors the changes are propagated to all sites. This can be done in two ways: The initial SQL statement is broadcasted and executed on each node or the changed rows sent. Depending on the network throughput and the amount of modified data one of the two possibilities will be preferable.

The *atomic broadcast protocol* is only used to broadcast information requiring ordering. All non-critical communication is handled with ordinary broadcasts to reduce overhead even further (The propagation of decision messages out of order also reduces the deadlock rate).

The protocol used for Postgres-R executes transactions in four phases [PG-R00]:

1. **Local Read Phase:** Perform all read operations locally
   - Execute write operations on shadow copies
   - Acquire the appropriate lock before executing the operation

2. **Send Phase:**
   - If \( T_i \) is read-only then commit
   - Else bundle all writes into write set \( WS_i \) and multicast it to all sites including the sending (this ensures the same delivery order at all sites)

3. **Lock Phase:**
   - Upon delivery of \( WS_i \) on item \( X \) in \( WS_i \):
     1. For each operation \( w_i(X) \) on item \( X \) in \( WS_i \):
        a) Perform a conflict test:
           - If a local transaction \( T_j \) has granted lock on \( X \) and \( T_j \) is still in its read or send phase abort \( T_j \)
           - If \( T_i \) is in its send phase then multicast the decision message "abort" for \( WS_i \) (as mentioned earlier, not ordered)
        b) If there is no lock on \( X \), grant the lock to \( T_i \).
           Otherwise enqueue the lock request directly after all locks from transactions that are beyond their lock phase.
   - 2. If \( T_i \) is a local transaction multicast the decision message "commit"

4. **Write Phase:**
   - Whenever a write lock is granted apply the corresponding update.
     - A local transaction can commit and release all locks once all updates have been applied to the database.
     - A remote transaction must wait until the decision message arrives and terminate accordingly.

This protocol requires that the local server sends two messages per transaction: One with the write-set and another to confirm or abort the transaction. The only disadvantage of this protocol is the need to abort reads if they conflicting with writes, to be able to keep read-only transactions local. This could however be circumvented by additionally implementing SI or cursor stability CS.

**Cursor Stability (CS)**

CS is a technique allowing to avoid the need to abort ongoing read operations when a write operation arrives, basically the read-locks are divided up into short- and long read-locks. Short reads
are read only transactions and immediately released after the read operation, often as soon as the cursor has jumped to the next row. Long-locks are often precursor to write operations and held until the end of the transaction.

**Snapshot isolation (IS)**

Snapshot isolation guarantees that all read operations made in a transaction will happen on the same state/version of the database (consistency). This can be achieved by labeling every row of a table with a version number or a timestamp of the transaction.

... the test setup ...

As Figure 4 illustrates the test setup consisted of several servers (up to 15) and some real and virtual clients for load generation. Each server was running PostgreSQL with Postgres-R on top of it.

PostgreSQL was enhanced with a *Replication Manager* that was responsible for handling the synchronization messages and was keeping track of the four transaction phases. The total ordering was achieved by using [ENSEMBLE] a group communication toolkit.

![Figure 4, Postgres-R Architecture [PG-R00]](image)

... and the performance analysis

The performance analysis (Figure 5) shows that the multicast approach leads to a well scalable system in terms of read throughput. Also the server count has no effect on the throughput as long as small datasets are modified. The third diagram shows that by broadcasting the altered rows instead of the SQL statement even write commands are scalable.

![Figure 5, Performance Analysis [PG-R00]](image)

The assumption made in [PG-R00] that the slowest machine has no effect on the performance of the cluster as long as the global serialization order has been determined can be considered to be wrong. The queue/buffer of the slowest machine will eventually overflow which would lead to a full resynchronization of that machine. The resynchronization would have a negative impact on the already saturated cluster of which, the peak load situation, lead to this overflow on the first hand.
Conclusion

Both replication approaches - eager and lazy - have many disadvantages in their pure form. Lazy replication may look very promising and easier to handle at the first glance; however the implementation of reconciliation rules often has to be done application-specific and requires a huge effort. Eager replication may circumvent the need for reconciliation but at the price of having a large synchronization and messaging overhead.

The paper [AES98] clearly shows that GCP can be used to outsource serializations and conflict detection mechanisms to lower layers. The implementation of the full logic still has to be made on one of the layers. On the bottom line however there is a high potential to implement such serialization much more efficient on the network layer than it could have been done with server-to-server communication on the application layer.

The approach demonstrated in [PG-R00] is streamlined on the application perspective; it however relies on a network layer guarantying total ordering. Situations like an outage on a link between two sites of a company (e.g. Bern and Zurich) was not yet considered in Postgres-R implementation - this situation is however handled in the two-tier approach presented.

The usage of GCPs is very promising but still requires some additional effort. This may also explain why Oracle and others still relay on protocols based on lazy replication.

References

[GHOS96] Jim Gray, Pat Helland, Patrick E. O'Neil, Dennis Shasha
The Dangers of Replication and a Solution
SIGMOD Conference 1996: 173-182

[PG-R00] Bettina Kemme, Gustavo Alonso
Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement DB Replication
VLDB 2000: 134-143

[ORACLE] Oracle 7 Server Distributed System: Replicated Data
Oracle part number A21903, March 1994

[KA98] Bettina Kemme, Gustavo Alonso
A Suite of Database Replication Protocols based on Group Communication Primitives
ICDCS 1998: 156-163

[LM08/09] Bernd Erk, Netways GmbH (Nürnberg, Germany)
Spitzenleistung, Linux Magazin 2008/09, Linux New Media AG: 074-076

[MYSQL08] MySQL (now Oracle Corporation)
MySQL 5.1 Reference Manual - 16.1.2. Replication Formats

Using broadcast primitives in Replicated Databases
IEEE Int. Conf. on Distributed Computing Systems May, 1998

[ENSEMBLE] Developed at Cornell University and Hebrew University of Jerusalem
Group Communication Toolkit based on Horus
http://www.cs.technion.ac.il/dsl/projects/Ensemble/