Universidade do Minho
Escola de Engenharia
Departamento de Informática

Dissertação de Mestrado
Mestrado em Engenharia Informática

Database Replication in Large Scale Systems

Miguel Gonçalves de Araújo

Trabalho efectuado sob a orientação do
Professor Doutor José Orlando Pereira

Junho 2011

Partially funded by project ReD – Resilient Database Clusters (PDTC / EIA-EIA / 109044 / 2008).

Declaração

Nome: Miguel Gonçalves de Araújo

Endereço Electrónico: miguelaraujo@lsd.di.uminho.pt

Telefone: 963049012

Bilhete de Identidade: 12749319

Título da Tese: Database Replication in Large Scale Systems

Orientador: Professor Doutor José Orlando Pereira

Ano de conclusão: 2011

Designação do Mestrado: Mestrado em Engenharia Informática

Universidade do Minho, 30 de Junho de 2011

Miguel Gonçalves de Araújo

Experience is what you get when you didn't get what you wanted.

Randy Pausch (The Last Lecture)

Acknowledgments

First of all, I want to thank Prof. Dr. José Orlando Pereira for accepting being my adviser and for encouraging me to work on the Distributed Systems Group at University of Minho. His availability, patient and support were crucial on this work. I would also like to thank Prof. Dr. Rui Oliveira for equal encouragement on joining the group.

A special thanks to my parents for all support throughout my journey and particularly all the incentives to keep on with my studies.

I thank to my friends at the group that joined me on this important stage of my studies. A special thank to Ricardo Coelho, Pedro Gomes and Ricardo Gonçalves for all the discussions and exchange of ideas, always encouraging my work. I also thank for the good moments and fellowship to my friends and colleagues at university. Not forgetting all the good moments in the many trips to Braga, I would like also to thank my friend Rui Durães.

To all the past and current members of the Distributed Systems Group I want to thank for the good working environment, fruitful discussions and fundamental opinions. In special, I would like to thank Ricardo Vilaça and Nuno Carvalho for their help.

Although not personally acquainted, I also thank to Kay Roepke and to Jan Kneschke for all the questions clarified and opinions given through IRC.

Finally, thanks to all my friends for their friendship and comprehension during this course and to everyone that read this thesis and contributed with corrections and critics.

Resumo

Existe nos dias de hoje uma necessidade crescente da utilização de replicação em bases de dados, sendo que a construção de aplicações de alta performance, disponibilidade e em grande escala dependem desta para manter os dados sincronizados entre servidores e para obter tolerância a faltas.

Uma abordagem particularmente popular, é o sistema código aberto de gestão de bases de dados MySQL e seu mecanismo interno de replicação assíncrona. As limitações impostas pelo MySQL nas topologias de replicação significam que os dados tem que passar por uma série de saltos ou que cada servidor tem de lidar com um grande número de réplicas. Isto é particularmente preocupante quando as actualizações são aceites por várias réplicas e em sistemas de grande escala. Observando as topologias mais comuns e tendo em conta a assincronia referida, surge um problema, o da frescura dos dados. Ou seja, o facto das réplicas não possuírem imediatamente os dados escritos mais recentemente. Este problema vai de encontro ao estado da arte em comunicação em grupo.

Neste contexto, o trabalho apresentado nesta dissertação de Mestrado resulta de uma avaliação dos modelos e mecanismos de comunicação em grupo, assim como as vantagens práticas da replicação baseada nestes. A solução proposta estende a ferramenta MySQL Proxy com plugins aliados ao sistema de comunicação em grupo Spread oferecendo a possibilidade de realizar, de forma transparente, replicação activa e passiva.

Finalmente, para avaliar a solução proposta e implementada utilizamos o modelo de carga de referência definido pelo TPC-C, largamente utilizado para medir o desempenho de bases de dados comerciais. Sob essa especificação, avaliamos assim a nossa proposta em diferentes cenários e configurações.

Abstract

There is nowadays an increasing need for database replication, as the construction of high performance, highly available, and large-scale applications depends on it to maintain data synchronized across multiple servers and to achieve fault tolerance.

A particularly popular approach, is the MySQL open source database management system and its built-in asynchronous replication mechanism. The limitations imposed by MySQL on replication topologies mean that data has to go through a number of hops or each server has to handle a large number of slaves. This is particularly worrisome when updates are accepted by multiple replicas and in large systems. Noting the most common topologies and taking into account the asynchrony referred, a problem arises, the freshness of the data, i.e. the fact that the replicas do not have just the most recently written data. This problem contrasts with the state of the art in group communication.

In this context, the work presented in this Master's thesis is the result of an evaluation of the models and mechanisms for group communication, as well as the practical advantages of group-based replication. The proposed solution extends the MySQL Proxy tool with plugins combined with the Spread group communication system offering, transparently, active and passive replication.

Finally, to evaluate the proposed and implemented solution we used the reference workload defined by the TPC-C benchmark, widely used to measure the performance of commercial databases. Under this specification, we have evaluated our proposal on different scenarios and configurations.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Internet-based services have become a standard in our information society, supporting a wide range of economic, social, and public activities. And in this globalized era, since large organizations are present in different places all over the world, information must be always online and available. The loss of information or its unavailability can lead to serious economic damages.

Availability has recently became critical due to large amounts of data being captured and used each day with the emerging online services. Large companies such as Google, eBay, or Amazon handle exabytes of data per year. Facebook claims to be one of the largest MySQL installations running thousands of servers handling millions of queries, complemented by its own Cassandra data store for some very specific queries. So, high-availability, performance, and reliability are all critical requirements in such systems.

Both of these challenges are commonly addressed by means of the same technique, namely data replication. Application components must be spread over a wide area network, providing solutions that enable high availability through network shared contents.

Data replication has become a rising study topic in many areas, specially in distributed systems, mainly for fault tolerance proposes, and in databases, mainly for performance reasons. So, for this reasons, since databases are more and more deployed over clusters and workstations, replication is a key component. Replicating data improves fault-tolerance since the failure of a site does not make a data item inaccessible. Available sites can take over the work of failed ones. It also improves performance since data access can be localized over the database network, i.e. transaction load is distributed across the replicas, achieving load balancing. On the other hand, it can be used to provide more computational resources, or allow data to be read from closer sites reducing the response time and increasing the throughput of the system.

However, replication introduces a trade-off between consistency and performance.

Due to this, it is important to use adequate replication mechanisms.

Actually replicated databases, are usual solutions used in datacenter's or local area networks. And so, most of solutions adopt a model where data consistency is relaxed in favor to better performance, i.e. most replicated databases do not ensure data consistency among replicas.

1.1 Problem Statement

Most database management systems implement asynchronous master-slave replication. These systems provide mechanisms for master-slave replication that allows configuring one or more servers as slaves of another server, or even to behave as master for local updates. MySQL in particular allows almost any configuration of master and slaves, as long as each server has at most one master. This usually leads to a variety of hierarchical replication topologies, but includes also a ring which allows updates to be performed at any replica, as long as conflicts are avoided. Being this engine widely used, open-source, and fast, it becomes a very interesting topic of investigation and contribution.

Being replicated asynchronously, data is first written on the master server and then is propagated to the respective slaves, and so, specially in the case of thousands or hundreds of servers, the nodes will not obtain the most recent data. This method of disseminating data combined with the impossibility of having more than one master makes it impossible to spread data rapidly to a large number of replicas.

This problem contrasts with the state of the art in group communication having in account the characteristics inherent to this. Guarantees such as reliability, order, and message stability, as also message delivery guarantees as for example reliable messaging or fully ordered messages.

1.2 Objectives

The central objective of this work is to improve the scalability and fault-tolerance of MySQL proposing, implementing and evaluating a mechanism of updates distribution that allows thousands or hundreds of replicas. For that, it is necessary to understand firstly the MySQL's replication mechanism, and also the data freshness measurement. Moving next to the main objective to improve this with the use of a group communication protocol.

1.3 Contributions

This thesis proposes a new approach to MySQL replication that enables state-machine replication and primary-backup replication by combining the software tool MySQL Proxy and the Spread Group Communication System. The key to our implementation is to take advantage of the guarantees of reliability, order, message stability and message delivery guarantees for reliable messaging or fully ordered messaging of group communication, to build an mechanism of active and passive replication for the MySQL database management system.

In detail, we make the following contributions:

- **Evaluation and measuring of data freshness in scenarios of large scale replicated databases**
  This contribution addresses the difficulty of measure accurately the impact of replication in data freshness by introducing a tool that can accurately measure replication delays for any workload and then apply it to the industry standard TPC-C benchmark [1]. We also evaluate data freshness by applying the tool to two representative MySQL configurations with a varying number of replicas and increasing workloads using the industry standard TPC-C on-line transaction processing benchmark [1].

- **Documentation and analysis of the software tool MySQL Proxy**
  We fully document, analyze and discuss the components and working of the software tool MySQL Proxy.

- **Development of plugins for group based replication using MySQL Proxy**
  We propose a solution to implement group based replication using the software tool MySQL Proxy. The proposal exploits the plugin based architecture of MySQL Proxy to implement plugins to use the Spread Group Communication Toolkit for both active and passive replication.

- **Evaluation and performance analysis of the proposed solution**
  We evaluate the developed solution using realistic workloads based on the industry standard TPC-C benchmark [1]. We analyze the behaviour of the solution under different conditions and configurations comparing it to the MySQL standard replication mechanism.

1.4 Dissertation Outline

This thesis is organized as follows: Chapter 2 describes the state of the art in database replication; Chapter 3 introduces and discusses group-based replication; Chapter 4 presents the performance tests and the efforts done in order to measure the replication propagation delay in the MySQL Database Management System; Chapter 5 presents and documents the software tool MySQL Proxy; Chapter 6 presents the proposed approaches and solutions; Chapter 7 evaluates the solution implemented using realistic workloads; and finally Chapter 8 concludes the thesis, summarizing its contributions and describing possible future work.

Related Publications

Portion of the work presented in this thesis has been previously published in the form of conference and workshop papers:

- M. Araújo and J. Pereira. Evaluating Data Freshness in Large Scale Replicated Databases. In *INForum*. 2010.

Chapter 2

Database Replication

Database replication is a technique that allows taking a database and making an exact copy of it on another site. In a replicated database system each site stores a copy of the database. This copies can be total (full replication) or partial (partial replication). Data access is done via transactions. A transaction represents a unit of work (read or write operation) performed against a database.

Database replication is in charge of ensuring concurrent and consistent transaction execution. This is made by concurrency control and replica control mechanisms. Concurrency control isolates concurrent transactions with conflicting operations, while replica control coordinates the access to different copies. Replication protocols are the ones in charge of performing this task.

2.1 Classification Criteria

Classification of replication protocols can be done according to where and when can updates be performed [17]. Regarding to when can updates be propagated we have **lazy** replication protocols, also known as asynchronous protocols, and **eager** replication protocols, also known as synchronous replication protocols. Regarding to where can updates be performed we have two approaches, **primary-copy** and **update-everywhere** [17].

2.1.1 Eager vs Lazy Replication

Eager replication keeps all replicas synchronized at all nodes by updating all the replicas as part of one atomic transaction [17]. This is comparable to the Two-Phase Commit protocol. Eager protocols propagate updates to remote copies within the transaction boundaries and coordinate the different sites before the transaction commits [35]. With this,

if the database management system is serializable, serializable execution is achieved - there are no concurrency anomalies. Strong consistency and fault-tolerance are achieved by ensuring that updates are stable at multiple replicas before replying to clients [21]. And so, crash detection is also allowed. These protocols are also flexible since they, in contrast with lazy replication, allow updates to any copy in the system.

But this type of replication has some disadvantages. Despite consistency achieved in these models it is expensive in terms of message overhead and response time. So the performance is reduced and transaction response times are increased because extra messages are added to the transaction, also mobile nodes cannot use an eager scheme when disconnected and the probability of deadlocks and failed transactions rises very quickly with transaction size and number of nodes.

Lazy replication propagates replica updates asynchronously to other nodes after the transaction commits. The other nodes are updated later by capturing updates in the master, distributing and applying them. This mechanism as an impact on user visible performance, specially on transaction latency that is reduced.

Lazy schemes update replicas using separate transactions, in contrast to eager schemes that distribute updates to replicas in the context of the original updating transaction. The eager method makes it easy to guarantee transaction properties, such as serializability. However, since such transactions are distributed and relatively long-live, the approach does not scale well. [13].

Due to the complexity and performance of eager replication, there is a wide spectrum of lazy schemes. Lazy replication reduces response times since transactions can be executed and committed locally and only then updates are propagated to the other sites [22].

But asynchronous replication also has shortcomings, being the major one stale data versions. Even allowing a wide variety of optimizations, copies are allowed to diverge so inconsistencies among copies might occur [34]. This kind of replication is also not suitable for fault-tolerance by fail-over while ensuring strong consistency because updates can be lost after a failure of the master.

Lazy schemes reduce response times, however durability cannot be guaranteed. If a node fails before it propagates the update of a committed transaction T to the other sites, then T is lost.

2.1.2 Primary-copy vs Update-Everywhere

The other classification parameter referred by [17] is about who can perform updates, **primary-copy** vs **update-everywhere** replication.

In the **primary copy** approach all the updates are initially performed at one copy, called master or primary copy. After this step the updates are propagated and executed in the other copies (replicas). All replicas must contact the same server to perform updates. To notice that after the execution of the transaction, the local server (master) sends the response back to the client, and only after the commit the updates are propagated to the other sites. This allows a reduction on the communication overhead. The Agreement Coordination phase [34], is relatively simple because all the ordering of the transactions takes place on the primary copy and the replicas need only to apply the propagated updates. This introduces a single point of failure and a potential bottleneck, but simplifies replica control.

In contrast, the **update-everywhere** method allows any copy to be updated, it speeds up data access but makes replica coordination more complex and expensive. In this case the Agreement Coordination phase, is much more complex than in the primary copy approach. Since any copy can perform updates, conflicting transactions may occur at the same time between replicas. So, the copies on the different sites may not only be inconsistent but also stale. Reconciliation is needed to decide what transactions should be performed and those that should be undone.

Update-Everywhere

This approach, also called **Lazy Group** Replication [17], works by sending a transaction to every node in order to apply the root transaction's update to the replicas at the destination node, when a transaction commits. It is possible for two nodes update the same object and race each other to install their updates to other nodes. So, the replication mechanism must detect this and reconcile the two transactions so that their updates are not lost.

The method commonly used to detect and reconcile transaction updates is the use of timestamps. Each object carries the timestamp of its most recent update. Each replica update carries the new value and is tagged with the old object timestamp. Each node detects incoming replica updates that would overwrite earlier committed updates. The node tests if the local replica's timestamp and the update's old timestamp are equal. If so, the update is safe. The local replica's timestamp advances to the new transaction's timestamp and the object value is updated. If the current timestamp of the local replica does not match the old timestamp seen by the root transaction, then the update may be dangerous. In such cases, the node rejects the incoming transaction and submits it for reconciliation.

Transactions that would wait in an eager replication system face reconciliation in a lazy-group replication system. Waits are much more frequent than deadlocks because it

takes two waits to make a deadlock. So, if waits are a rare event, deadlocks are even a more rare event. Eager replication waits cause delays while deadlocks create application faults. With lazy replication, waits are much more frequent; this is what determines the reconciliation frequency.

Primary-Copy

This approach, also called **Master** Replication [17] is the most common method used in lazy replication.

Master replication assigns an owner to each object and the owner stores the object's correct current value. Updates are first done by the owner and then propagated to other replicas. Different objects may have different owners.

When a transaction wants to update an object, it sends an RPC (remote procedure call) to the node owning the object. To get serializability, a read action should send read-lock RPCs to the masters of any objects it reads.

Simplifying, the node that originates the transactions, broadcasts the replica updates to all the slaves after the master transaction commits. The originating node sends one slave transaction to each slave node. Slave updates have timestamps to assure that all the replicas converge to the final state. If the record timestamp is newer than a replica update timestamp, the update is "stale" and can be ignored. Alternatively, each master node send replica updates to slaves in sequential commit order.

Lazy-Master replication is not suitable for mobile applications. If a node wants to update an object it must be connected to the object owner and participate in an atomic transaction with it.

Lazy-master systems have no reconciliation failures, conflicts are resolved by waiting or deadlock. The deadlock rate for a lazy-master system is similar to a single node system with much higher transaction rates. Transactions operate on master copies of objects. The replica update transactions do not really matter, because they can abort and restart without affecting the user. The main issue relies on how frequently the master transactions deadlock.

This is a better behavior than lazy-group replication. Lazy-master replication sends fewer messages during the base transaction and so completes more quickly. Nevertheless, all of these replication schemes have troubling deadlock or reconciliation rates as they grow to many nodes.

In summary, lazy-master replication requires contact with object masters and so is not useable by mobile applications. Lazy-master replication is slightly less deadlock prone than eager-group replication primarily because the transactions have shorter duration.

2.2 Consistency Criteria

Replica consistency is a key issue to achieve fault tolerance. The consistency property ensures that the database remains in a consistent state before and after the transaction is over.

A correct behaviour in a replicated system must ensure the strictest correctness criterion: linearizability. The correctness criterion linearizability, also called one-copy equivalence, gives the illusion that a replicated database systems is single, i.e. non-replicated. The effect of transactions performed by clients on replicated objects should be the same as if they had been performed one at-a-time on a single set of objects. That is a desirable property because it preserve's the program's semantics [6].

Most of replications techniques were designed for serializable database management systems (DBMS) in order to obtain One-Copy Serializability Replication (1CS). However, there is an increasing popularity of Snapshot Isolation (SI) level where a transaction obtains the latest committed snapshot version of the database as of the time it starts [16,24].

The main goal of providing transactional guarantees weaker than 1CS, such as SI, is that the database system can achieve increased concurrency by relaxing the isolation requirements on transactions. This means that concurrently executing transactions may see each others indirectly through their effects on the database. However, SI does not guarantee serializability. It allows update transactions to read old data.

Another correctness criteria is Strong serializability. This criterion ensures that a transaction that starts after a previous transaction has finished is serialized after is predecessor. But recently, [12] demonstrated that this criteria is too strong for lazy replicated systems, and proposed Strong Session One-Copy Serializability Replication (Strong Session 1SR).

Strong Session 1SR is a generalization of One-Copy Serializability Replication (1SR) and Strong One-Copy Serializability (Strong 1SR) that allows important transaction ordering constraints to be captured and unimportant ones to be ignored, improving data freshness. However, it has been proved that Strong 1SR is very difficult to achieve as the propagation latencies increase while the Strong Session 1SR can be maintained almost as efficiently as 1SR.

Concluding, higher degrees of transaction isolation guarantee fewer anomalies but with larger performance penalties. There is a range of solutions to achieve transaction isolation guarantees, each one introducing a trade-off between performance and data consistency.

2.3 Replication in Large Scale Databases

Current database replication techniques have attained some degree of scalability; however, there are two main limitations to existing approaches. Firstly, most solutions adopt a full replication model where all sites store a full copy of the database. The coordination overhead imposed by keeping all replicas consistent allows such approaches to achieve only medium scalability. Secondly, most replication protocols rely on the traditional consistency criterion, 1-copy-serializability, which limits concurrency, and thus scalability of the system [31].

The main problem of the protocols that ensure serializability is that all concurrency conflicts must be considered, like read/write and write/write transactions. Read/write conflicts are very frequent and limit the amount of potential concurrency in the system, resulting in lack of scalability.

The protocols studied are fully replicated, then updates have to be executed at all replicas. So, in eager protocols, the replicated database does not scale under update workloads, because all sites do the same work.

On the other hand, lazy replication updates all the copies in separate transactions, so the latency is reduced in comparison with eager replication. A replica is updated only by one transaction and the remain replicas are updated later on by separate refresh transactions [28].

Although there are concurrency control techniques and consistency criterion which guarantee serializability in lazy replication systems, these techniques do not provide data freshness guarantees. Since transactions may see stale data, they may be serialized in an order different from the one in which they were submitted.

So, asynchronous replication leads to periods of time that copies of the same data diverge. Some of them have already the latest data introduced by the last transaction, and others have not. This divergence leads to the notion of data freshness: The lower the divergence of a copy in comparison with the other copies already updated, the fresher is the copy [29].

Actually have been proposed some consistency techniques to improve data freshness, but having a trade-off between consistency and performance.

Recently, some refresh strategies have also been proposed. The first one to be mentioned is the ASAP model, in which the updated are propagated from the source to the replicas as soon as possible [5, 7, 11]. Another strategy used in data warehouses, is to refresh replicas periodically, as in [9,25]. In [32] a refresh strategy was proposed, which consist in maintaining the freshness of replicas by propagating updates only when a replica is too stale.

Mixed strategies were also proposed. An approach to improve data freshness has been proposed in which the data sources push updates to caches nodes when their freshness level is too low [26]. If needed, cache nodes can also force refreshment. Another strategy discussed in [23] states that an asynchronous Web cache maintains materialized views whit an ASAP strategy while regular views are regenerated on demand. In these approaches, refresh strategies are not chosen having in concern the performance related to the workload in question.

2.4 MySQL

In this work we take the MySQL case study to systematize and evaluate the replication mechanisms. MySQL database management system implements asynchronous master-slave replication. The system provides mechanism to configure master-slave replication that allows configuring one or more servers as slaves (replicas) of another server, or even to behave as master for local updates.

The configuration of replication allows an arrangement of masters and slaves in different topologies. It is possible to replicate the entire server, replicate only certain databases or to choose what tables to replicate.

2.4.1 Replication Formats

MySQL uses the Primary-Copy Replication method, and supports two kinds of replication, statement-based and row-based.

Statement-Based Replication

In the statement-based approach, every SQL statement that could modify the data is logged on the master server. After this those statements are re-executed on the slave against the same initial dataset and in the same context. It generally requires less data to be transferred between the master and the slave, as well as taking up less space in the update logs. It does not have to deal with the format of the row. The compactness of the data transfer will generally allow it to perform better. On the other hand, it is necessary to log a lot of execution context information in order for the update to produce the same results on the slave as it did originally on the master. In some cases it is not possible to provide such a context. Statement-based replication is also more difficult to maintain, as the addition of new SQL functionality frequently requires extensive code updates for it to replicate properly.

Row-Based Replication

In the row-based approach, every row modification gets logged on the master and then applied on the slave. No context information is required. It is only necessary to know which record is being updated, and what is being written to that record. Given a good code base, the maintenance of a row-based replication is also fairly simple. Since the logging happens at a lower level, the new code will naturally execute the necessary low-level routines that modify the database, which will do the logging with no additional code changes. However, on a system that frequently executes queries such as UPDATE customer SET status='Current' WHERE id BETWEEN 10000 and 20000, row-based replication produces unnecessarily large update logs and generates a lot of unnecessary network traffic between the master and the slave. It requires a lot of awareness of the internal physical format of the record, and still has to deal with the schema modifications. In some situations the performance overhead associated with the increased I/O could become unacceptable.

2.4.2 Replication Mechanism

The replication mechanism of MySQL, works at a high level in a simple three-part process:

- The master records changes to its data in its binary log (these records are called binary log events).
- The slave copies the master's binary log events to its relay log.
- The slave replays the events in the relay log, applying the changes to its own data.

Briefly, after writing the events to the binary log, the master tells the storage engine to commit the transactions. The next step is for the slave to start a I/O thread to start the dump. This process reads events from the master's binary log. If there are events on the master, the thread writes them on the relay log. Finally, a thread in the slave called SQL thread reads and replay events from the relay log, thus updates slave's data to match the master's data. To notice that the relay log usually stays in the operating system's cache, having very low overhead.

This replication architecture decouples the processes of fetching and replaying events on the slave, which allows them to be asynchronous. That is, the I/O thread can work independently of the SQL thread. It also places constraints on the replication process, the most important of which is that replication is serialized on the slave. This means updates

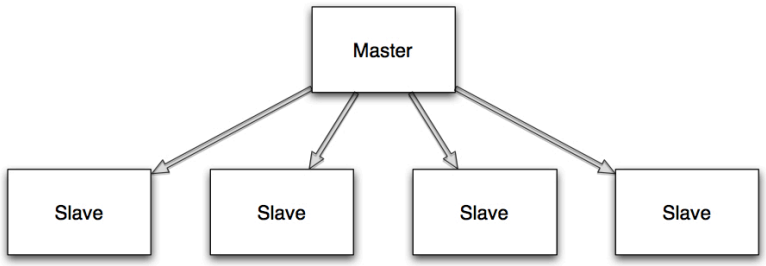

Figure 2.1: Master and Multiple Slaves Replication

that might have run in parallel (in different threads) on the master cannot be parallelized on the slave. However, this is a performance bottleneck for many workloads.

2.4.3 Replication Topologies

It is possible to setup MySQL replication for almost any configuration of masters and slaves, with the limitation that a given MySQL slave instance can have only one master.

The simplest topology besides **Master-Slave** is **Master and Multiple Slaves** (Figure 2.1). In this topology, slaves do not interact with each other at all, they all connect only to the master. This is a configuration useful for a system that has few writes and many reads. However, this configuration is scalable to the limit that the slaves put too much load on the master or network bandwidth from the master to the slaves becoming a problem.

Other possible configuration is **Master-Master in Active-Active Mode**. This topology involves two servers, each configured as both a master and slave of the other. The main bottleneck in this configuration resides on how to handle conflicting changes.

A variation on master-master replication that avoids the problems of the previous is the **Master-Master in Active-Passive** mode replication. The main difference is that one of the servers is a read-only "passive" server. This configuration permits swapping the active and passive server roles back and forth very easily, because the servers configurations are symmetrical. This makes failover and failback easy.

The related topology of the previous ones is **Master-Master with Slaves**. The advantage of this configuration is extra redundancy. In a geographically distributed replication topology, it removes the single point of failure at each site.

One of the most common configuration in database replication, is the **Ring** topology (Figure 2.2). A ring has three or more masters. Each server is a slave of the server before it in the ring, and a master of the server after it. This topology is also called circular replication. Rings do not have some of the key benefits of a master-master setup, such as symmetrical configuration and easy failover. They also depend completely on every

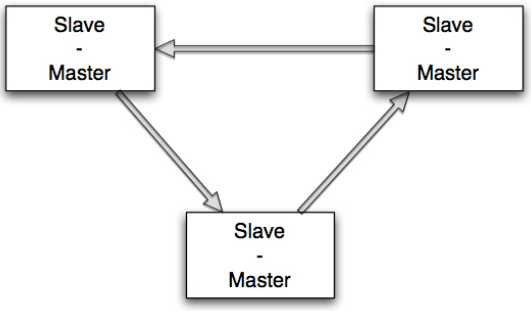

Figure 2.2: Ring Topology

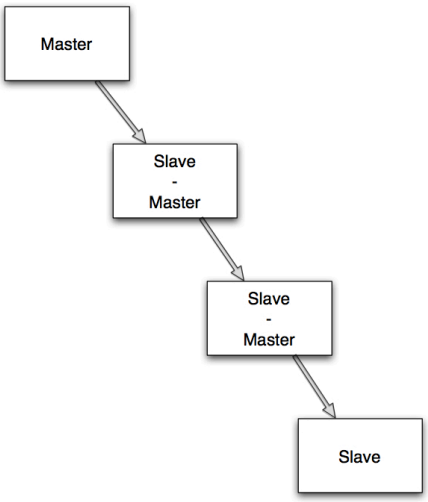

Figure 2.3: Chain Topology

node in the ring being available, which greatly increases the probability of the entire system failing. And if you remove one of the nodes from the ring, any replication events that originated at that node can go into an infinite loop. They will cycle forever through the topology, because the only server that will filter out an event based on its server ID is the server that created it. In general, rings are brittle and best avoided. Some of the risks of ring replication can be decreased by adding slaves to provide redundancy at each site. This merely protects against the risk of a server failing, though.

Another possibility, regarding some certain situations where having many machines replicating from a single server requires too much work for the master, or the replication is to spread across a large geographic area that chaining the closest ones together gives better replication speed, is the **Daisy Chain** (Figure 2.3). In this configuration each server is set to be a slave server to one machine as as master to another in a chain. Again, like the ring topology the risk of losing a server can the decreased by adding slaves to provide

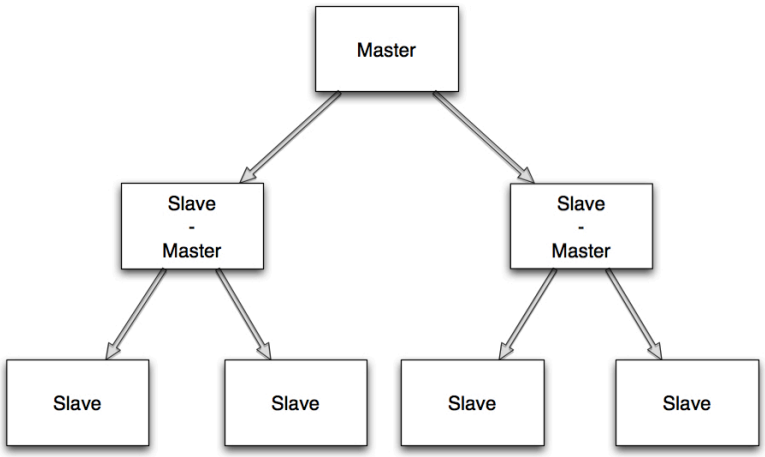

Figure 2.4: Tree Topology

redundancy at each site.

The other most common configuration is the **Tree or Pyramid** topology (Figure 2.4). This is very useful in the case of replicating a master to a very large number of slaves. The advantage of this design is that it eases the load on the master, just as the distribution master did in the previous section. The disadvantage is that any failure in an intermediate level will affect multiple servers, which would not happen if the slaves were each attached to the master directly. Also, the more intermediate levels you have, the harder and more complicated it is to handle failures.

2.4.4 Replication Latency

In theory, replication speed should be extremely fast (i.e. , bounded only by the network speed). The MySQL binlog dump process does not poll the master for events, which would be inefficient and slow. Instead, the master notifies the slave of events. Reading a binary log event from the master is a blocking network call that begins sending data practically instantaneously after the master logs the event. Thus, it's probably safe to say the event will reach the slave as quickly as the slave thread can wake up and the network can transfer the data.

However, since MySQL uses the Primary-Copy Replication method, it lacks of scalability since updating transactions are executed by a single replica and this compromises its performance.

Noting both the replication topologies and the behaviour of MySQL's replication mechanism, one can deduce that in these, several hops are made by updates in order to reach all replicas. The update delay will increase proportional to the number of hops,

having a major impact on large scale systems' data freshness.

2.5 Summary

In this chapter we have introduced database replication presenting for a start the different main replication protocols and the consistency criteria. Afterwards asynchronous replication have been described with detail leading us to draw some conclusions about its application on large scale scenarios. The data freshness problem is stated motivating the work to achieve a solution to this problem. It is interesting to note that even though lazy replication models reduce latency taking advantage of the fact that the replicas are updated in separate transactions, it does guarantee data freshness since they lead to period of time where copies of the same data diverge.

The chapter ends with a description and discussion around the MySQL database management system replication mechanisms. It allow us to draw some conclusions about replication speed on MySQL and on different topologies focusing on the data freshness problem. The limitation of having more than one master restricts the dissemination of the updates to a large number of replicas. This is the basis for the definition of group communication primitives and database replication based on group communication in the following chapter.

Chapter 3

Group-based Replication

High-availability, performance, and reliability requirements are mostly achieved by the data replication technique. Database replication it is commonly implemented using group communication primitives. These primitives provide a framework that reduces the complexity of the implementation. Replication commonly addresses the linearizability issue with two main models: primary-backup, also called passive replication, or state-machine, also called active replication.

3.1 Group Communication

A distributed system consists of multiple processes that communicate via communication links using message passing. These processes can behave according to their specification if they are correct or crash or behave maliciously if they are incorrect [19]. This set of processes is known as group. A process group has the ability to control the distribution of messages and signals, i.e., a message sent to a process group is delivered to all the other processes.

A group represents a set of processes, as it can address all the processes into a single entity. For example, consider a replicated object x. A group Gx can represent the set of replicas of x. As so, Gx can be used to address a message to all the replicas of x [18]. A group can be used to send messages to all the constituents of it without naming them explicitly, i.e. the process addresses the message to the logical group address.

Since group communication protocols are based on groups of processes, i.e. recipients, communication takes into account the existence of multiple receivers for the messages. As so, message passing within the group must ensure properties such as reliability, and order.

Group Communication provides group membership management to track the dy-

namic constitution of groups. Groups can be of two different kinds: static or dynamic [18]. Groups are considered static if the membership is not changed during the system life-time. All initial members of the group remain with the membership even if they crash. If a recover is possible, the member remains member of the group. On the other hand, dynamic groups are the opposite, membership can change during the life-time of the system. If a replica crashes it leaves the group and if it recovers at any time it can rejoin the group. This states the notion of group membership and view. A group membership maintains group views, i.e., the set of processes believed to be correct at the moment. For the crashing process example, when it crashes it is removed from the Group and when it recovers it rejoins, the history of the group membership is constituted by the views [19]. The group membership service is responsible for tracking correct and incorrect processes, creating and destroying groups, to add or withdraw processes to and from a group and to notify process members of membership changes. Group membership can be defined by the following properties [10]:

Self inclusion:
: Every view installed by a process includes itself, i.e. if a process p installs view V, then p is a member of V

Local monotonocity:
: If a process p installs view V after installing view V' then the identifier of V is greater than that of V'

Agreement:
: Any two views with the same identifier contains the same set of processes.

Linear membership:
: For any two consecutive views there is at least one process belonging to both views.

The definition of a group communication protocol involves properties such as reliability, order and atomicity. In order to obtain reliability in message passing, group communication use reliable multicast. A reliable multicast primitive can be defined as follows: If process p is correct and reliably multicasts message m, then every correct recipient eventually delivers m [20].

Sometimes there is a need to coordinate message transmission with the group membership service. This is achieved by *view synchrony*. View synchrony synchronizes processes on membership changes. The definition is as follows [10]: any two processes that install two consecutive views will deliver the same set of messages multicast between these views.

To multicast the messages View Synchrony defines two primitives: *VSCAST* and *VSDELIVER*. Virtual Synchronous Multicast *(VSCAST)* satisfies the following properties [10]:

Integrity:
If a process p delivers *(VSDELIVER)* a message m, then message m was previously *VSCAST(m, g)*;

No Duplication:
If a process q delivers *(VSDELIVER)* m and m', then m $\neq$ m';

View Synchrony:
If processes p and q install two consecutive views, V and V', then any message delivered *(VSDELIVER)* by p in V is also delivered *(VSDELIVER)* by q in V;

Termination:
If a process p is correct and *VSCAST(m, g)* in view V, then each member q of V either delivers *(VSDELIVER)* m or installs a new view V' in V.

However virtual synchrony multicast is not enough in some particular cases, where there is a need to deliver messages sent to a set of processes at each site in the same order. *TOCAST* provides a group communication primitive that guarantees that a message m, sent to a group g (*TOCAST(m,g)*) is delivered (*TODELIVER*) in the same order at every member of group g. Total Order Multicast is defined as following [15]:

Integrity:
If a process p delivers *TODELIVER* a message m, it does it so at most once and only if m was previously *TOCAST(m, g)*;

Validaty:
If a process p *TOCAST* a message m, then a correct process p' eventually delivers *(TODELIVER)* m;

Agreement:
If a process p *TOCAST* a message m, and a correct process p' delivers *(TODELIVER)* m then all correct processes eventually also delivers *(TODELIVER)* m;

Total Order:
If processes p and a q *TOCAST(m, g)* and *TOCAST(m', g)*, respectively, then two correct processes r and s deliver *(TODELIVER)* m and m' in the same order.

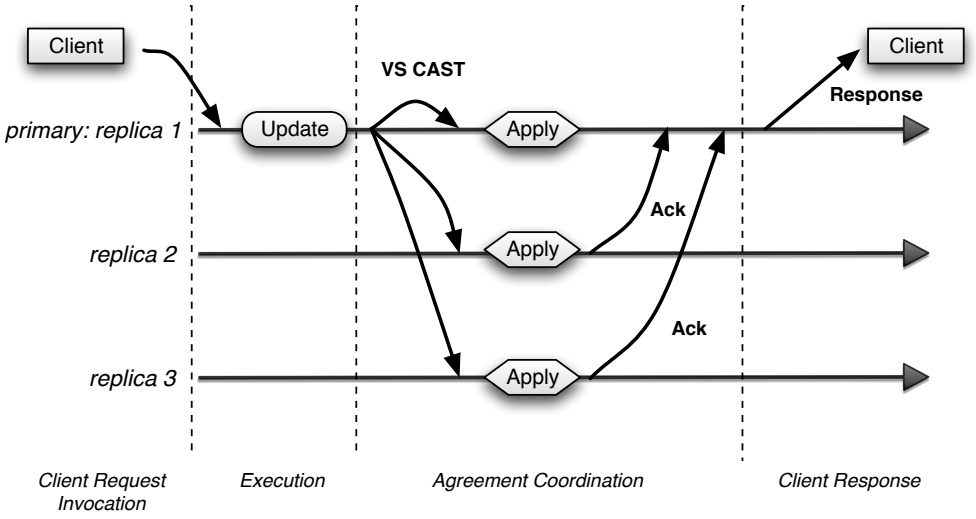

Figure 3.1: Primary-Backup Replication

3.2 Primary-Backup Replication

A classical approach for replication is to use a server as the primary and all the others as backups of this [8]. The client issue requests to the primary server only. This server has a main role to receive client invocations and to return to it the responses.

This technique states that the replicas do not execute the client invocation but apply the changes produced by the invocation executed on the primary, i.e., the updates [34]. The primary executes the client invocations and sends the updates to the replicas. However, updates need to be propagated in the same order according to the order in which the primary replica received the invocations. This way, linearizability is achieved because the order on the primary replica defines the total order of all servers [19].

As seen in (Figure 3.1) The client starts by sending the request invocation to the primary server. This server executes the request which will give rise to a response. It then updates its state and coordinates with the other replicas by sending them the update information. Finally the primary server sends the response to the client once it receives the acknowledgment from all the correct replicas.

However, linearizability is obtained if the primary does not crash since it states the total order on all invocations. If the primary crashes, three cases can be distinguished [18]:

- The primary crashes before sending the update message to the replicas;
- The primary crashes after or while sending the update message, but before the

client receives the response;

- The primary crashes after the client has received the response;

In all the three cases a new primary replica has to be selected. For the first case, when the crash happens before the primary sends the update to the replicas the client will not receive any response so it will issue the request again. The new primary will considers the invocation as a new one. In the second case, the client will also not receive any response, however since the crash happened after the update message was sent atomicity must be guaranteed, i.e. either the replicas receive the message or none. If none receives the update message then the process is similar to the fist case. Otherwise, if all the replicas receive the update then the state of each is updated as supposed but the client will not receive any response, issuing again the request invocation. The solution to this problem was to introduce information in order to know the invocation identification (invID) and respective response (red). Thus, avoiding to handle the same invocation twice. When the primary receives an invocation with the same identification (invID) it immediately send the response (res) back to the client.

The great advantage of the primary-backup technique is that it allows non-deterministic operations, i.e. it is possible for each replica to have multi-threading. Besides that factor, it has a lower cost in terms of processing power compared to other replication techniques. However, when the primary fails it has some costs for re-electing a new primary and handle the crash. Concerning fault transparency, in contrast to the state-machine replication the crash of the primary is not transparent to the client since it increases the latency between invocation and reception of the response. However, the replicas crash is completely transparent to the client.

3.2.1 Group communication and passive replication

At a first glance, the primary-backup technique does not need group communication to obtain primitives as *TOCAST* because the primary replica is which defines the update sending order. However, when the primary replica crashes there is a need to select a new primary and handle the crash event so group communication is needed. There is a need to use the dynamic groups property of group communication protocols. Group members must agree on a unique sequence of views [19]. When the primary replica crashes, a new view is installed and a new primary replica is chosen. However, in this example, the primary backup crashes while sending an update and only some of the replicas receive that update. Due to this, a simple multicast primitive is not enough so the view-synchronous multicast*(VSCAST)* is used.

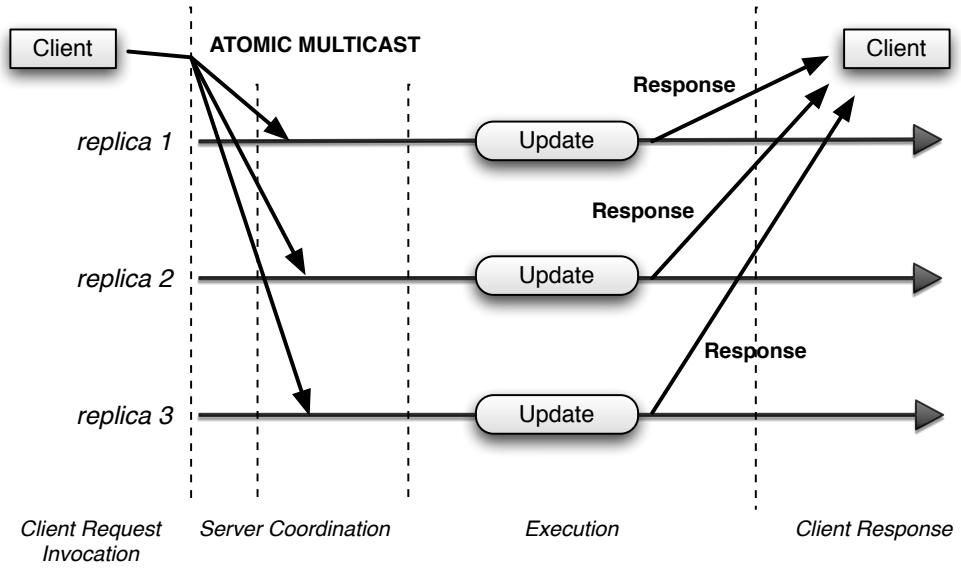

Figure 3.2: State-Machine Replication

3.3 State-Machine Replication

Since fault tolerance is commonly obtained with multiple servers with the same data, the state of each server must be distributed among all replicas. In this technique, the state update is received by all replicas in the same order [18].

Contrasting with the primary-backup model, in active replication there is not a centralized control by one of the servers. This way fault-tolerance can be achiever in a greater scale since the multiple servers can fail independently without compromising the whole replicated system. Each replica has the same role in processing and distributing the updates, and consistency is guaranteed by assuming that all replicas receive the invocations of client processes in the same order [19].

To obtain this level of consistency, the client requests must be propagated having the properties of order and atomicity, i.e., using the primitive Atomic Multicast or Total Order Multicast.

The great advantage of this technique is the transparency obtained. A crash of a single replica is transparent to the client process since it does not need to repeat the request. So, the client is never aware nor needs to take in concern a replica failure. All the replicas process the request even if one fails. However, active replication introduces more costs to the replication since each invocation is processed by all replicas.

As in (Figure 3.2), the client starts by sending a request to the servers. This is achieved using an Atomic Multicast that guarantees the total order property needed for coordina-

tion. Then each replica processes the request in the same order since replicas are deterministic producing the same result, and reply with the request result to the client. In this phase the client usually waits for receiving the first response, or to receive a majority of identical responses [18].

3.3.1 Group communication and active replication

The state-machine approach, as described above requires that the invocations sent to all servers are atomic and on the same order. As so, this technique requires the total-order multicast primitive*(TOCAST)*. A process sends a message with an invocation, which is received by a replica that coordinates with the other replicas to guarantee the properties of the total-order multicast primitive: order, atomicity and termination. After that the replica can deliver the message [19].

3.4 Spread Group Communication Toolkit

The Spread toolkit is a group communication system [1]. Spread provides reliability, ordering and stability guarantees for message delivery. Spread supports a rich fault model that includes process crashes and recoveries and network partitions and merges under the extended virtual synchrony semantics. The standard virtual synchrony semantics is also supported [3]. It provides besides group communication, an highly tuned application-level multicast and point to point support.

Spread provides high performance messaging across local and wide area networks. The big question that arises is how Spread handles wide area networks and how it provides these characteristics in those scenarios since they bring three main difficulties. One of the difficulties is related to the variety of loss rates, latency and bandwidth over the different parts of the network. Other difficult is related to the significantly higher rate of packet loss in comparison to LAN networks. And finally, it is more complex to implement efficient reliability and ordering on the wide area multicast mechanism besides its limitations.

The Spread group communication system addresses the above difficulties through three main structural design issues [4]. It allows the utilization of different low level protocols to disseminate messages depending on the configuration of the network. And in particular, Spread integrates two low-level protocols: Ring and Hop. Ring protocol is meant to be used on local area networks and the Hop protocol in wide area networks.

Spread is built following a daemon-client architecture. This brings several benefits,

[1] http://www.spread.org

mainly the fact that this way membership changes have less impact and cost on the global system. Simple joins and leaves of processes are translated into a single message.

Finally, spread decouples the message dissemination and reliability mechanisms from the global ordering and stability protocols. This allows messages to be forwarded to the network immediately as also supports the Extended Virtual Synchrony model [2] where data messages are only sent to the minimal necessary set of the network components, without compromising the strong semantic guarantees.

Spread is highly configurable, allowing the user to configure it to their needs. It allows the user to control the type of communication mechanisms used and the layout of the virtual network. Spread can use a single daemon over the whole network or to use one daemon in every node running group communication applications. Each Spread daemon keeps track of the computers's membership, keeping track of processes residing on each machine and participating on group communication. Since this information is shared between the daemon, it created the lightweight process group membership.

3.4.1 Message Types for Data and Membership Messages

Spread allows different types of messages satisfying the ordering and reliability properties described above. The following flags as described on [2] set the message type:

UNRELIABLE_MESS:
: The message is sent unreliably, however it is possible to be dropped or duplicated even that duplications are very rare.

RELIABLE_MESS:
: The message will arrive once at all members of its destination group, it may be arbitrarily, but finitely delayed before arriving, and may arrive out of order with regards to other reliable messages.

FIFO_MESS:
: The message has the reliable message properties, but it will be ordered with all other FIFO messages from the same source. However, nothing is guaranteed about the ordering of FIFO messages from different sources.

CAUSAL_MESS:
: This type of message has all the properties of FIFO messages and in addition are causally ordered with regards to all sources.

[2] http://www.spread.org/docs/spread_docs_4/docs/message_types.html

AGREED_MESS:
: These messages have all the properties of FIFO messages but will be delivered in a causal ordering which will be the same at all recipients, i.e. all the recipients will 'agree' on the order of delivery.

SAFE_MESS:
: These messages have all the properties of AGREED messages, but are not delivered until all daemons have received it and are ready to deliver it to the application. This guarantees that if any one application receives a SAFE message then all the applications in that group will also receive it unless the machine or program crashes.

Regarding data messages Spread allows to define a type of message that is used to identify a data/application message. This is defined by the flag: REGULAR_MESS.

Finally, a desired property in some use cases is the ability to not deliver a message to the application connection which sent it. However, one must be aware that if the application has multiple connections open which have joined the same group then other connections will receive it. This is defined by the flag: SELF_DISCARD.

3.5 Summary

This chapter describes group communication primitives, introducing the theoretical basis of message passing primitives, groups and group membership and motivating the work on defining a replication protocol based on group communication by demonstrating the properties and guarantees of reliability, order, and message stability, as also message delivery guarantees as for example reliable messaging or fully ordered messages.

Detailing these guarantees one can conclude the practical advantages of group-based replication. As so, we have described two main approaches for replication: primary-backup and state-machine, and how does group communication fits the needs of each.

Taking into account the limitations of MySQL's replication discussed on the previous chapter, one can induce a possible solution for this problem using group communication. However the main concern when using MySQL asynchronous replication mechanism is the data freshness. But one questions how big is this delay. Several efforts were made in order to measure the delay and to assess the impact on replication topologies. These efforts and concluding results are presented on the following chapter.

Chapter 4

Measuring Propagation Delay

MySQL allows almost any configuration of master and slaves, as long as each server has at most one master. As described in Chapter 2, this usually leads to a variety of hierarchical replication topologies, but includes also a ring which allows updates to be performed at any replica, as long as conflicts are avoided.

It is thus interesting to assess the impact of replication topology in MySQL, towards maximizing scalability and data freshness. This is not however easy to accomplish. First, it requires comparing samples obtained at different replicas and thus on different time referentials, or, when using a centralized probe, network round-trip has to be accounted for. Second, the number of samples that can be obtained has to be small in order not to introduce a probing overhead. Finally, the evaluation should be performed while the system is running a realistic workload, which makes it harder to assess the point-in-time at each replica with a simple operation.

In this chapter we address these challenges by presenting the several efforts made in order to measure the asynchronous replication delay of the MySQL's Database Management System.

4.1 Background

MySQL replication is commonly known as being very fast, as it depends strictly on the the speed that the engine copies and replays events, the network, the seize of the binary log, and time between logging and execution of a query [30]. However, there have not been many systematic efforts to precisely characterize the impact on data freshness.

One approach is based on the use of a User Defined Function returning the system time with microsecond precision [30]. Inserting this function's return value on the tables we want to measure and comparing it to the value on the respective slave's table we

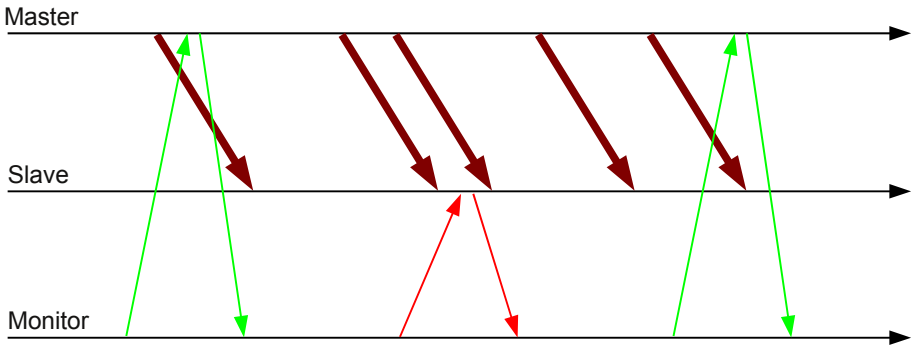

Figure 4.1: Impossibility to probe simultaneously master and slaves.

can obtain the time delay between them. But this measurements can only be achieved on MySQL instances running on the same server due to clock inaccuracies between different machines.

A more practical approach uses a Perl script and the Time::HiRes module to get the system time with seconds and microseconds precision.[1] The first step is to insert that time in a table on the master, including the time for the insertion. After this, the slave is queried to get the same record and immediately after the attainment of it the subtraction between system's date and time got from the slave's table is made, obtaining the replication time. As with the method described above this one lacks of accuracy due to the same clock inaccuracies.

4.2 Approach

Our approach is based on using a centralized probe to periodically query each of the replicas, thus discovering what has been the last update applied. By comparing such positions, it should be possible to discover the propagation delay. There are however several challenges that have to be tackled to obtain correct results, as follows.

Measuring updates. The first challenge is to determine by how much two replicas differ and thus when two replicas have applied exactly the same amount of updates. Instead of trying to compare database content, which would introduce a large overhead, or using a simple database schema and workload that makes it easy, we use the size of the transactional log itself. Although this does not allow us to measure logical divergence, we can determine when two replicas are exactly with the same state.

[1] http://datacharmer.blogspot.com/2006/04/measuring-replication-speed.html

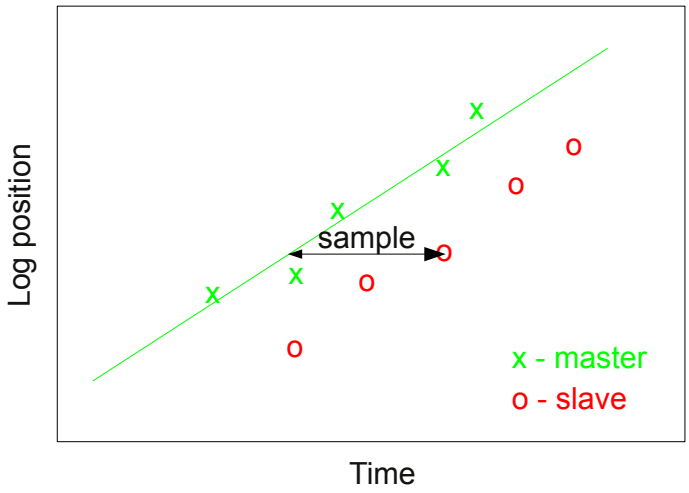

Figure 4.2: Log position over the time

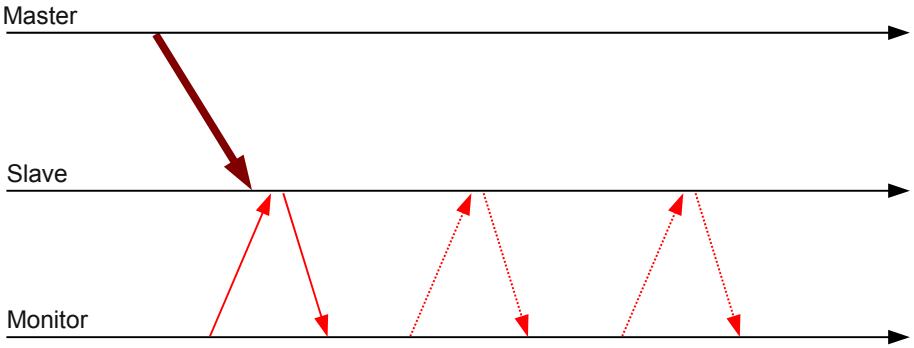

Figure 4.3: Sampling twice without updates erroneously biases the estimate.

Non-simultaneous probing. The second challenge is that, by using a single centralized probe one cannot be certain that several replicas are probed at exactly the same time. Actually, as shown in (Figure 4.1), if the same monitor periodically monitors several replicas it is unlikely that this happens at all. This makes it impossible to compare different samples directly.

Instead, as shown in (Figure 4.2) we consider time–log position pairs obtained by the monitor and fit a line to them (using the least-squares method). We can then compute the distance of each point obtained from other replicas to this line along the time axis. This measures how much time such replica was stale.

Eliminating quiet periods. Moreover, as replication traffic tends to be bursty. If one uses repeated samples of a replica that stands still at the same log position, the estimate is progressively biased towards a (falsely) higher propagation delay, as shown in (Figure 4.3). This was solved by selecting periods where line segments obtained from both

replicas have a positive slope, indicating activity.

Dealing with variability. Finally, one has to deal with variability of replication itself and over the network used for probing. This is done by considering a sufficient amount of samples and assuming that each probe happens after half of the observed round-trip. Moreover, a small percentage of the highest round-trips observed is discarded, to remove outliers.

4.2.1 Implementation

An application to interrogate the master instance and several replicas of the distributed database scheme was developed. This tool stores the results in a file for each instance. To obtain the log position it uses the MySQL API in order to obtain the replication log position. The temporal series of observed log positions are then stored in separate files, one for each node of the distributed database.

Results are then evaluated off-line using the Python programming language and R statistics package. This script filters data as described and then adjusts a line to the values of the log files and compares them. This includes looking for periods of heavy activity and fitting line segments to those periods. With these line segments, the script compares each slave points with the corresponding segment on the master, if the segment does not exist for the selected point, the point is ignored. In the end, average is calculated based on the difference of values between slave points and corresponding segments on the master. A confidence interval can also be computed, using the variance computed from the same data.

4.2.2 Workload

In order to assess the distributed database used in the case study, we have chosen the workload model defined by TPC-C benchmark [1], a standard on-line transaction processing (OLTP) benchmark which mimics a wholesale supplier with a number of geographically distributed sales districts and associated warehouses. Specifically, we used the Open-Source Development Labs Database Test Suit 2 (DBT-2), a fair usage implementation of the specification.

Although TPC-C includes a small amount of read-only transactions, it is composed mostly by update intensive transactions. This choice makes the master server be almost entirely dedicated to update transactions even in a small scale experimental setting, mimicking what would happen in a very large scale MySQL setup in which all conflicting

updates have to be directed at the master while read-only queries can be load-balanced across all remaining replicas.

It simulated the activities found in complex OLTP environment by exercising a breadth of system components associated with such environments, which are characterized by:

- The simultaneous execution of multiple transaction types that span a breadth of complexity;
- On-line and deferred transaction execution modes;
- Multiple on-line terminal sessions;
- Moderate system and application execution time;
- Significant disk input/output;
- Transaction integrity (ACID properties);
- Non-uniform distribution of data access through primary and secondary keys;
- Databases consisting of many tables with a wide variety of sizes, attributes, and relationships;
- Contention on data access and update.

In detail, the database is constituted by the following relations: *warehous*, *district*, *customer*, *stock*, *orders*, *order line*, *history*, *new order*, and *item*. Each simulated client can request five different transaction types that mimic the following operations:

New Order: adding a new order to the system (with 44% probability of occurrence);

Payment: updating customer's balance, district and warehouse statistics (with 44% probability of occurrence);

Orderstatus: returning a given costumer latest order (with 4% probability of occurrence);

Delivery: recording the delivery of products (with 4% probability of occurrence);

Stocklevel: determining the number of recently sold items that have a stock level below a specified threshold (with 4% probability of occurrence);

Each client is attached to a database server and produces a stream of transaction requests. When a client issues a request it blocks until the server replies, thus modeling a single threaded client process. After receiving a reply, the client is then paused for

some amount of time (think-time) before issuing the next transaction request. The TPC-C model scales the database according to the number of clients. An additional warehouse should be configured for each additional ten clients. The initial sizes of tables are also dependent on the number of configured clients.

During a simulation run, clients log the time at which a transaction is submitted, the time at which it terminates, the outcome (either abort or commit) and a transaction identifier. The latency, throughput and abort rate of the server can then be computed for one or multiple users, and for all or just a subclass of the transactions. The results of each DBT-2 run include also CPU utilization, I/O activity, and memory utilization.

4.2.3 Setting

Two replication schemes were installed and configured. A six machines topology of master and multiple slaves, and a six machine topology in daisy chain.

The hardware used included six HP Intel(R) Core(TM)2 CPU 6400 - 2.13GHz processor machines, each one with one GByte of RAM and SATA disk drive. The operating system used is Linux, kernel 2.6.31-14, from Ubuntu Server with ext4 filesystem, and the database engine used is MySQL 5.1.54. All machines are connected through a LAN, and are named PD00 to PD07. Being PD00 the master instance, PD04 the remote machine in which the interrogation client executes, and the others the slave instances.

The following benchmarks were done using the workload TPC-C with the scale factor (warehouses) of two, number of database connections (clients) one hundred and the duration of twenty minutes.

MySQL Replication Setup

Two replication schemes were installed and configured. A five machines topology of master and multiple slaves was configured using the MySQL's asynchronous replication scheme.

In (Figure 4.4), each computer represents a node in the topology.

The other replication scheme used was the chain topology, in other words, the open ring topology.

In (Figure 4.5), each computer represents a node in the topology.

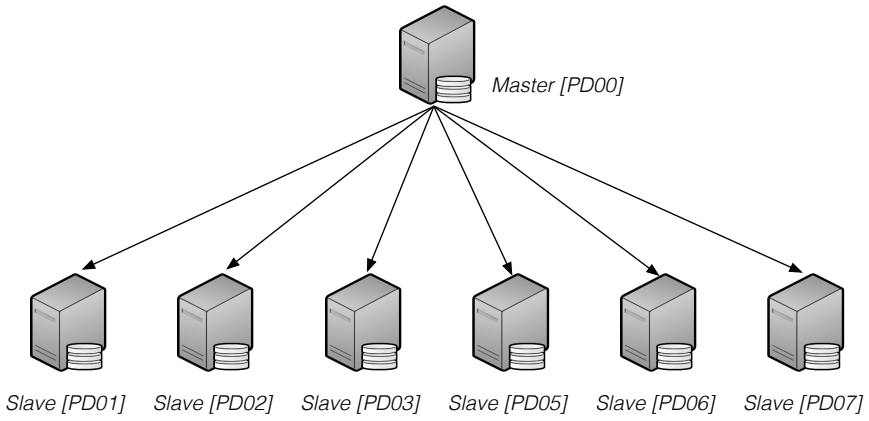

Figure 4.4: Master and Multiple Slaves topology

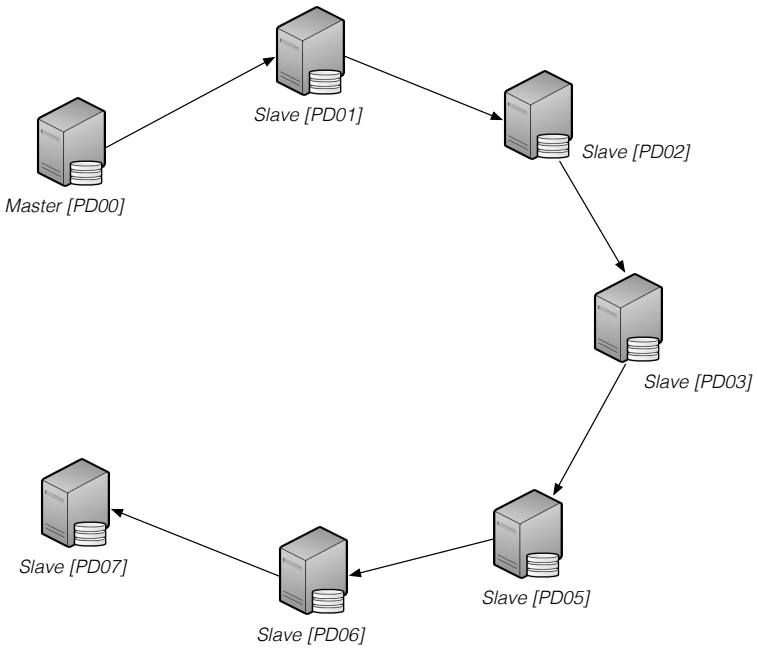

Figure 4.5: Chain topology

Replica	PD01	PD02	PD03	PD05	PD06	PD07
Number of samples	43947	43923	43797	43729	43962	44001
Average delay (μs)	3670	3419	3661	4121	3334	3565
99% confidence interval ($\pm$)	88	38	81	195	32	65

Table 4.1: Results for master and multiple slaves topology with 100 clients.

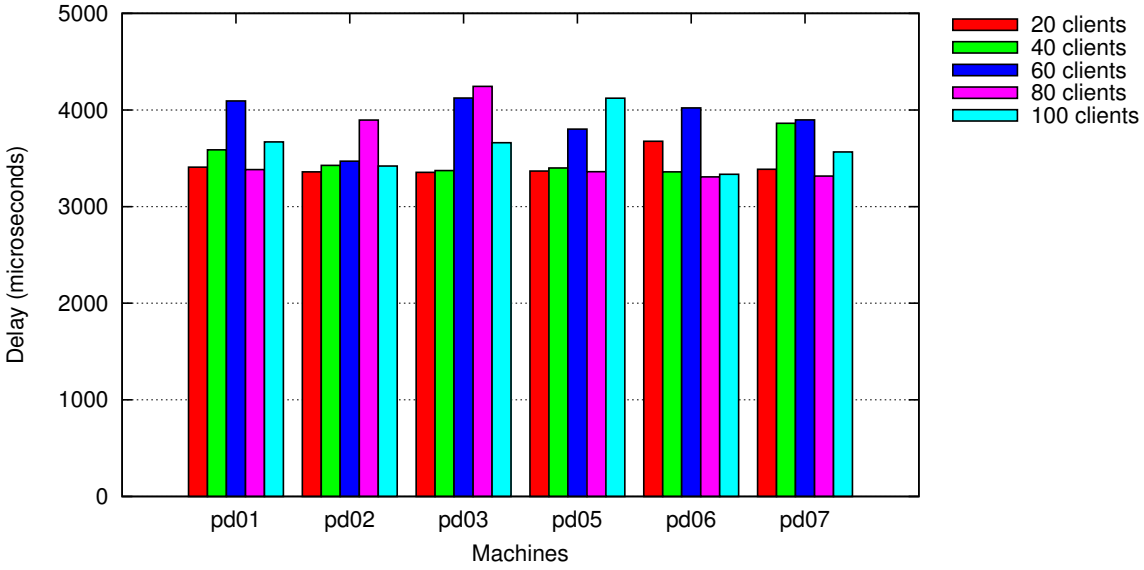

Figure 4.6: Scalability of master and multiple slaves topology.

4.3 Results

Results obtained with 100 TPC-C clients and the master and multiple slaves topology are presented in (Table 4.1). It can be observed that all replicas get similar results and that propagation delay is consistently measured close to 10 ms with a small variability.

Results obtained with an different number of TPC-C clients are show in (Figure 4.6). They show that propagation delay is similar between replicas and has little variation with the load imposed on the master. We can conclude that propagation delay is similar between replicas in a master and multiple slaves topology. Previously experiments with the same configuration but with ext3 filesystem showed that propagation delay grown substantially with the load imposed on the master. At the same time, as idle periods get less and less frequent due to the higher amount of information to transfer, the probability of a client being able to read stale data grown accordingly. However, with ext4 filesystem, propagation delay is similar between replicas and the setup behaves in the same way.

Replica	PD01	PD02	PD03	PD05	PD06	PD07
Number of samples	40597	40110	39372	38822	38161	39057
Average delay (μs)	3701	6505	9839	12409	15575	22341
99% confidence interval ($\pm$)	124	249	397	485	590	821

Table 4.2: Results for chain topology with 100 clients.

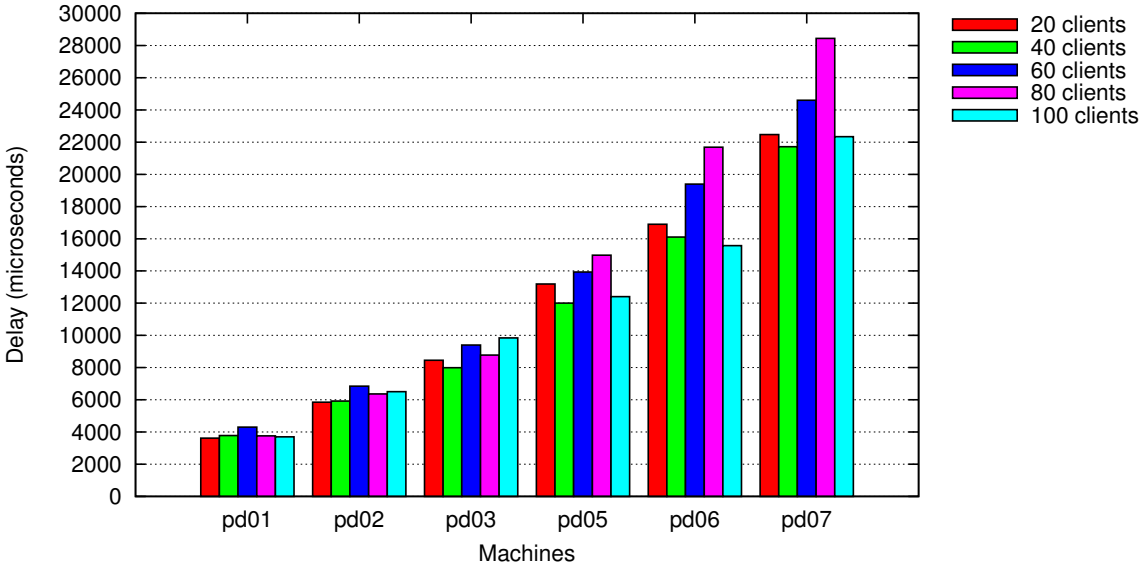

Figure 4.7: Scalability of the chain topology.

Results obtained with 100 TPC-C clients and the chain topology are presented in (Table 4.2). In contrast to master and multiple slaves, the delay now grows as the replica is farther away for the master. This configuration also gives an indication of how the ring topology would perform: As any replica would be, on average, half way to other masters, one should expect the same delay as observed here on replicas PD03 and PD05.

Results with an increasing number of TPC-C clients can also be found in (Figure 4.7), showing that propagation delay still grow substantially with the load imposed on the master. This means that using the ring configuration for write scalability with suffer the same problem, thus limiting its usefulness.

4.4 Summary

We have committed to evaluate the consequences on data freshness of the choice of replication topologies and of a growing workload. Our approach measures freshness in terms of time required for updates performed at the master replica to reach each slave while us-

ing a realistic update-intensive workload, as the proposed tool can infer freshness from a small number of samples taken at different points in time at different replicas. Experimental results obtained with this tool show that, in both tested replication topologies, the delay grows with the workload which limits the amount of updates that can be handled by a single replica. Moreover, we can also conclude that in circular replication the delay grows as the number of replicas increases, which means that spreading updates across several replicas does not improve update scalability. Finally, the delay grows also with the number of slaves attached to each master, which means that read scalability can also be achieved only at the expense of data freshness.

The conclusion is that the apparently unlimited scalability of MySQL using a combination of different replication topologies can only be achieved at the expense of an increasing impact in data freshness. The application has thus to explicitly deal with stale data in order to minimize or prevent the user from observing inconsistent results.

It is thus interesting to propose and implement other replication mechanisms to overcome the limitation presented and discussed above. However, one must take into account that in order to achieve this goal it is mandatory to intercept the requests and/or the updates to obtain primary-copy or state-machine replication.

Chapter 5

MySQL Proxy and Plugins

This chapter introduces the software tool MySQL Proxy. MySQL Proxy is a simple program which sits between a MySQL client and a MySQL server and can inspect, transform and act on the data sent through it.

Documentation on the internals and detailed operation for this software is scarce. Since version 0.8.1 documentation was introduced on the trunk branch of MySQL Proxy. However, it has been introduced gradually and so it still is incomplete and buggy. On this work, we complement and provide a foremost documentation by describing, analyzing and discussing the architecture and operation of this software tool.

5.1 Architecture

MySQL Proxy is a software application that provides communication between MySQL servers and one or several MySQL clients. It communicates over the network using the MySQL network protocol. A Proxy instance on his most basic configuration operates as Man in the Middle and pass the unchanged network packets from the client to the MySQL Server. It stands between servers and clients passing queries from the clients to the MySQL servers and returning the corresponding responses from the servers to the appropriate clients. So, this opens the possibility to change the packets when needed. This flexibility opens several other possibilities and it can be used for multiple purposes, being the most remarkable query analysis, query filtering and modification. Other possibilities include load balancing, failover, working as a pseudo MySQL server and client, query injection, connection pool and caching.

MySQL Proxy communicates over the network using the standard MySQL protocol, so it can be used with any MySQL compatible client. This includes the MySQL command-line client, any clients that use the MySQL client libraries, and any connector that sup-

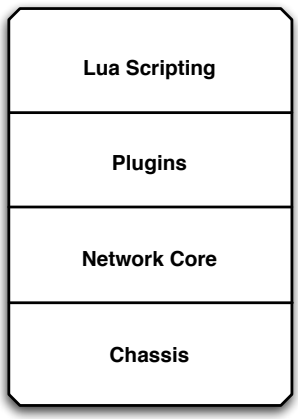

Figure 5.1: MySQL Proxy top-level architecture

ports the MySQL network protocol.

Being query interception the most basic feature of the Proxy it can be used for monitoring and altering the communication between the client and the corresponding server. With the Proxy it is possible to insert additional queries to send to the server and it can intercept and remove the results returned by this. A simple use case is to insert extra statements to each query sent to the server to obtain values of execution time, progress and then filter the results sent by the server to the client and separately log the monitoring results.

This feature of monitoring filtering and manipulation of queries does not require the user to make any modifications to the client or even implies that the client is aware that the Proxy is not a true MySQL server. The client communicates with the Proxy as with a MySQL server.

MySQL Proxy is also able to do load balancing by distributing the load across several servers. The default method used is the Shortest Queue First. It sends new connections to the server with the least number of open connections. Another useful application of the Proxy is Failover. The application can be used to detect dead hosts and use custom load balancers to decide how to handle a dead host.

MySQL Proxy embeds the Lua Scripting Language.[1] This programming language is simple and efficient. It can do object oriented programming, and it has scalars, tables, metatables and anonymous functions. It is also a language designed to be embedded into applications and widely used. The Proxy allows the use of Lua scripts and the basic query interception/changing is done using Lua scripts.

(Figure 5.1) illustrates the top-level architecture of MySQL Proxy. It was designed in

[1] http://www.lua.org/

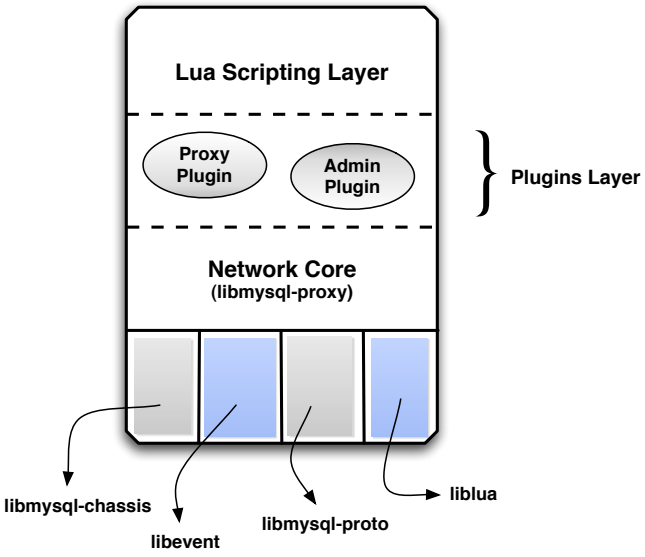

Figure 5.2: MySQL Proxy detailed architecture

a four layer application, being this layers the chassis, the network core, the plugins layer and finally the Lua scripting layer.

(Figure 5.2) illustrates a more detailed level of the architecture of MySQL Proxy. One can see that the Plugins layer is an abstract layer constituted by the loaded plugins. On the figure example, two plugins were loaded, Proxy and Admin. Under the Network Core layer there are several sub-layers, namely the chassis, libevent, mysql-protocol and liblua. These sub-layers constitute the top-level layer called chassis, being the main core of MySQL Proxy.

5.2 Chassis

The chassis is the main core of MySQL Proxy, providing the fundamental features that common applications need. Through it the application can load plugins and do whatever they implement. This means that the chassis itself can be used for any kind of application as it is not MySQL specific so the proxy features are carried out by the proxy plugin.

The chassis implements the following list of features and functionalities:

- Command-line option handling;
- Config-file handling;
- Logging;

- Plugin loading/unloading;
- Daemon (Unix) / service (Win32) support

The chassis also provides at a first glance several configuration file and command line options as described below.

5.2.1 Config-file and Command-line Options

MySQL Proxy leverages functionalities provided by Glib2. Glib is a low-level core library that is the basis of GTK+[2] and Gnome.[3] It eases development in C by providing data structures handling, portability wrappers and interfaces for functionalities as event loops, threads, dynamic loading and others.

MySQL Proxy uses Glib2 to provide parsing of configuration files and command-line options. Some of the functionalities of Glib2 that MySQL Proxy implements are the parsing with GOption and GKeyFile, the log facilities, and GModule.

For parsing of options the method starts with extracting the basic command-line options, then it processes the `defaults-file`, and finally processes other command-line options to override the `defaults-file`.

Basic options

The basic options provided by the application are:

- "–help", "-h". Shows the help menu;
- "–version", "-V". Show the version of MySQL Proxy;
- "–defaults-file=<file>". Configuration file;

Defaults File

The format of the defaults-file obeys to the key files syntax defined by freedesktop.[4] MySQL Proxy uses Glib to parse this config files.[5]

[2] http://www.gtk.org/
[3] http://www.gnome.org/
[4] http://freedesktop.org/wiki/Specifications/desktop-entry-spec?action=show&redirect=Standards%2Fdesktop-entry-spec
[5] http://library.gnome.org/devel/glib/stable/glib-Key-value-file-parser.html

Options

As most of front end applications, MySQL Proxy provides a set of command-line options. Command-line and defaults-file share the same set of options. Depending on the type of option they accept values in different forms. As showed on table 5.1.

Type	Command-line	Defaults-file
no value	`--daemon`	`daemon=1`
single value	`--user=foo ; --basedir=<dir>`	`user=foo ; basedir=dir`
multi value	`--plugins=proxy --plugins=admin`	`plugins=proxy, admin`

Table 5.1: Command-line and Defaults-file options examples

5.2.2 Front end

MySQL Proxy front end, parses the options and provides the `main()` functions the necessary values to start the chassis and defaults of the application. The command: `$ mysql--proxy` loads, by default, the plugins: plugin-proxy and plugin-admin.

5.2.3 Plugin Interface

The chassis provides the bridge needed for the correct work of the plugin interface. It resolves the path for the plugins and load them in a portable way. It checks versions, exposes the configuration options to the plugins and finally calls init and shutdown functions.

5.3 Network Core

The network core, is responsible for the sockets handling as also the database protocol. It is the layer that enables the interaction between the client and the server.

5.3.1 MySQL Protocol

The communication between the MySQL clients and the Servers is done via the MySQL protocol. This is implemented by the Connectors (Connector/C), MySQL Proxy and MySQL Server itself for the slave instances. And it supports:

- transparent encryption via "SSL"
- transparent compression via "Compressed Packet"

- a challenge-response "Auth Phase" to never send the clients password in cleartext
- and a "Command Phase" which supports needs of "Prepared Statements" and "Stored Procedures"

Full documentation of the Protocol can be found on the MySQL Forge Website. [6]

5.3.2 Connection Life Cycle

MySQL protocol has four basic states on the connection life cycle. This basic phases/states are:

- *connect*
  connection to the server
- *authentication*
  authentication phase including the sending and receiving of authentication request
- *query*
  execution of the query transactions on the server
- *disconnect*
  disconnection of server

MySQL Proxy has the ability to change the default behaviour of the network core. A plugin can implement the listening side of a connection, the connection side of a connection or both. For example, the admin-plugin implements the listening side of a connection as it only "reads" what a supposed client is sending to a server. The client-plugin as implementing the connection side it connects to an server, authenticates and "reads" the server outputs.

State machine

MySQL Proxy uses a state-machine approach to map the basic states of the MySQL protocol. As shown in (Figure 5.3), MySQL Proxy handles the basic states of MySQL protocol according to the basic hooks it implements. The basic procedure starts when the client connects. After receiving it, the server replies with the handshake packet. The client proceeds by sending the authentication packet with the necessary data. The server replies then with the result of the authentication processing. If the authentication is accepted, the client can send queries to the server in which the server replies with the result for

[6] http://forge.mysql.com/wiki/MySQL_Internals_ClientServer_Protocol

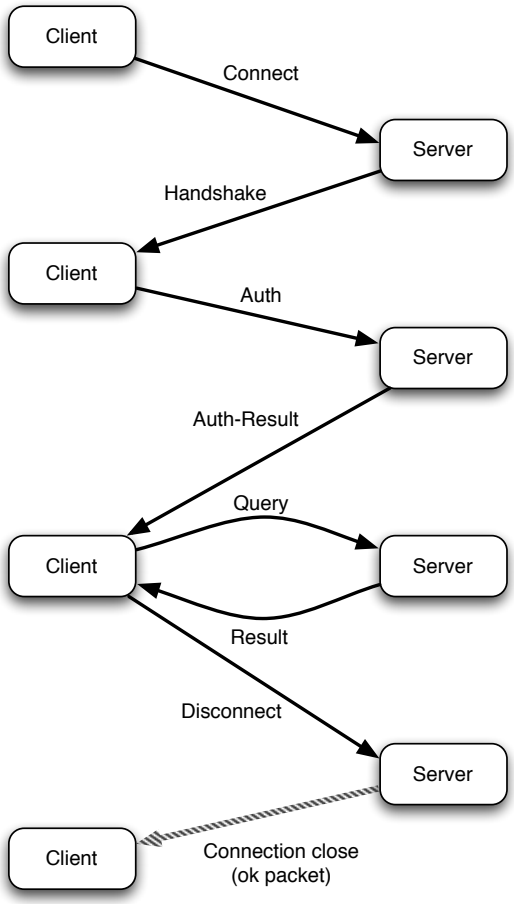

Figure 5.3: MySQL Protocol state-machine

each one. If the client wants to end the connection it sends a disconnect packet to the server.

A typical workflow of proxy plugin taking into account the proxy state machine, starts with the modification of the connection state (`con->state`) to the state: `CON_STATE_CONNECT_SERVER`. This happens due to the event handler that detects a client connection and starts the plugin state-machine. With this state alteration, the state-machine calls the `plugin_call` handler function with the actual state which in turn calls the corresponding plugin function to connect to the server, and change the state in order to receive the handshake from the server. This state would be `CON_STATE_READ_HANDSHAKE`. The connection state-machine will now call the *plugin_call* so that the necessary plugin function to read the handshake from the server is called. After the reading of the handshake the function can create the necessary authentication packet and change the state to `CON_STATE_SEND_AUTH`. This way the corresponding plugin function to sent the authentication packet to the server will be called. And so, after the execution of this function the

state is changed in order to read the result of the authentication sent to the server. This state is `CON_STATE_READ_AUTH_RESULT`. This alteration will call the `plugin_call` function in order to execute the plugin function to read the authentication result sent by the server, evaluate it and finally change the state. Being that the authentication is complete, the client can start to send queries to the server. And so, the state is changed to `CON_STATE_READY_QUERY`. In this stage, when the client performs a query, the proxy will pass it to the server and change the stat to `CON_STATE_READ_QUERY_RESULT` in order to read the result of the query appliance sent by the server. This connection stage will loop, until the client decides to finish the connection by sending the disconnect request.

5.3.3 Concurrency

The network engine is meant to handle several thousand simultaneous connections. As some of the features of MySQL Proxy include load-balancing and fail-over, it is supposed to handle a large group of connections to the MySQL server backends.

To achieve such scalability the Proxy was designed using an pure event driven asynchronous network approach. It exploits the socket ability to be set as nonblocking and use the the notifications (*poll()*, *select()*) to know when is the socket available to start the next I/O operation. This way, the implementation takes advantage of knowing when the sockets are available improving the I/O throughput. An event-driven design has a very small foot-print for idling connections. It just stores the connection state and let it wait for a event.

Up to version 0.7, MySQL Proxy uses a pure event-driven, non-blocking networking approach[7] using libevent 1.4.x.[8] Since version 0.8 of MySQL Proxy the chassis was improved with the implementation of a threaded network I/O in order to allow scaling with the number of CPUs and network cards available.

To enable network-threading MySQL Proxy must be started with the following option: `-event-threads=2 * no-of-cores (default: 0)`.

Without this option enabled the proxy executes the core functions in a single-threaded way. On a network or time event set the thread will execute the functions assigned to it. With multi-threading event-threads are created, being each one a simple small thread around the libevent `event_base_dispatch()` basic function. These event-threads have two states, idle and active. If they are executing the core functions they are active, when they are idle they wait for new evens to read and they can add them to their wait-list.

[7] http://kegel.com/c10k.html#nb
[8] http://monkey.org/ provos/libevent/

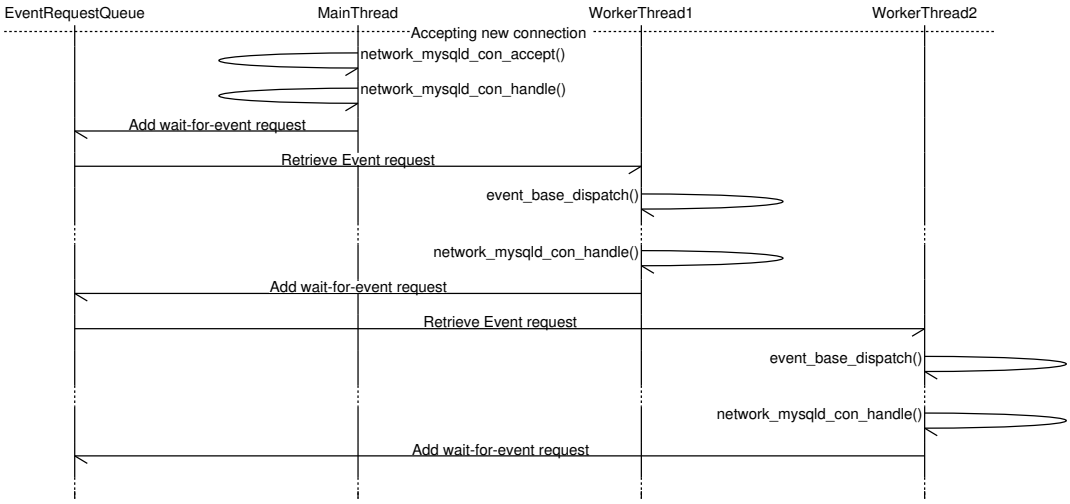

Figure 5.4: Thread I/O control flow

So the main advantage of the threaded-network is that a connection can jump between event-threads. If an idle thread is taking the wait-for-event request it will eventually execute the code, and so when the connection has to wait for a event again it is unregistered from the thread and sends its wait-for-event request to the global event-queue so that another thread can catch it up.

Up to version 0.8 the scripting layer is single-threaded. A global mutex protects the plugin interface, meaning that only one thread can be executing functions from the Lua layer. Even with the networking-threading enabled a connection is either sending packets or calling plugin functions on the Lua layer, meaning that the network events will be handled in parallel and will only wait if several connections call a plugin function.

Usually the scripts are small and only make simple decisions leaving most of the work to the network layer. As so, the next version of MySQL Proxy (version 0.9) will implement a multi-threaded approach to the scripting layer. Allowing this way several scripting threads are the same time. This allows the scripting layer to call blocking or slow functions without interfering with the execution of other connections, i.e. the network layer lifecycle.

MySQL Proxy implements network threading on `chassis-event-thread.c"`. The `chassis_event_thread_loop()` is the event-thread itself. A typical control flow is depicted in (Figure 5.4). In this example there are two event threads ("–event-threads=2"), each of which has its own `event_base`. The `network_mysqld_con_accept()` could be for example the proxy-plugin network event handler. It opens a socket to listen on and sets the accept handler which should get called whenever a new connection is made.

The accept handler is registered on the main thread's `event_base` (which is the same as the global chassis level `event_base`). After setting up the `network_mysqld_con` structure it then calls the state machine handler `network_mysqld_con_handle()`, still on the main thread. The state machine enters the initial state `CON_STATE_INIT` which currently will always execute on the main thread.

When MySQL Proxy needs to interact with either the client or the server, either waiting for the socket to be readable or needing to establish a connection to a backend, `network_mysqld_con_handle()` will schedule an "event wait" request (a `chassis_event_op_t`). It does so by adding the event structure into a asynchronous queue and generating a file descriptor event by writing a single byte into the write file descriptor of the `wakeup-pipe()`.

5.4 Plugins

As stated before, MySQL Proxy is in fact the "proxy-plugin". While the chassis and core make up an important part, it is really the plugins that make MySQL Proxy so flexible. The MySQL Proxy stable package contains the plugins: "proxy-plugin"; "admin-plugin"; "debug-plugin"; "master-plugin"; "replicant-plugin";

5.4.1 Proxy plugin

The "plugin-proxy" accepts connections on its `-proxy-address` and forwards the data to one of the `-proxy-backend-addresses`. Its default behaviour can be overwritten with scripting by providing a `-proxy-lua-script`.

Options

The Proxy Plugin options are presented in (Table 5.2).

5.4.2 Admin plugin

The admin plugin implements the listening side of a connection, i.e. it reads what the client is sending to the respective backend server.

Options

The Admin Plugin options are presented in (Table 5.3).

Option	Description
–proxy-lua-script=<file>, -s	Lua script to load at starting
–proxy-address=<host:port | file>, -P <host:port | file>	listening socket. Can be a unix-domain-socket or a IPv4 address. Default :4040
–proxy-backend-addresses=<host:port | file>, -b <host:port | file>	:default: 127.0.0.1:3306
–proxy-read-only-backend-addresses=<host:port>, -r <host:port | file>	only used if the scripting layer makes use of it
–proxy-skip-profiling	unused option. deprecated:: 0.9.0
–proxy-fix-bug-25371	unused option. deprecated:: 0.9.0
–no-proxy	unused option. deprecated:: 0.9.0
–proxy-pool-no-change-user	don't use "com-change-user" to reset the connection before giving a connection from the connection pool to another client

Table 5.2: Proxy Plugin options

Option	Description
–admin-username=<username>	admin username
–admin-password=<password>	admin password
–admin-address=<host:port>	admin address and port. default: :4041
–admin-lua-script=<file>	admin lua script. default: ../lib/mysql-proxy/admin.lua

Table 5.3: Admin Plugin options

5.4.3 Debug plugin

The debug plugin accepts a connection from the mysql client and executes the queries as Lua commands.

Options

Option	Description
–debug-address=<host:port>	debug address and port. default: :4043

Table 5.4: Debug Plugin options

The Debug Plugin options are presented in (Table 5.3).

5.4.4 Client plugin

The client plugin connects to the backend server, authenticates and reads the server outputs.

Options

Option	Description
–address=<host:port>	admin address and port. default: :4041
–username=<username>	admin username
–password=<password>	admin password

Table 5.5: Client Plugin options

The Client Plugin options are presented in (Table 5.3).

5.4.5 Master plugin

The master plugin acts like a MySQL master instance in a replication scenario. It reads the events from a file, Lua script or something else applied to the backend server and exposes them as a binlog-streams. In order to expose them and act like a master server it listens on the default port and handles the `COM_BINLOG_DUMP` command sent by the slave instances. It allows a MySQL server to connect to it using the `CHANGE MASTER TO ...` and `START SLAVE` commands in order to fetch and apply the binlog-streams.

Options

Option	Description
–master-address=<host:port>	master listening address and port. Default: :4041
–master-username=<username>	username to allow log in. Default: root
–master-password=<password>	password to allow log in. Default:
–master-lua-script	Lua script to execute by the master plugin.

Table 5.6: Master Plugin options

The master plugin options are presented in (Table 5.6).

5.4.6 Replicant plugin

The replicant plugin acts like a MySQL slave instance. It connects to a master, sends the `COM_BINLOG_DUMP` and the corresponding arguments for the binlog file, position, username and password. After the connection is established, it parses the binlog-streams sent by the server in order to create the relay binlog or to apply them directly to the backend server, taking in account what is defined on the Lua script.

Option	Description
–replicant-master-address=<host:port>	master instance listening address and port. Default: :4040
–replicant-username=<username>	slave instance username.
–replicant-password=<password>	slave instance password.
–replicant-lua-script	Lua script to execute by the replicant plugin.
–replicant-binlog-file	master binlog file from which the replicant plugin should read.
–replicant-binlog-pos	master binlog file position from which the replicant plugin should start reading.
–replicant-use-semisync	use semi-synchronous replication.

Table 5.7: Replicant Plugin options

Options

The replicant plugin options are presented in (Table 5.7).

5.5 Scripting

Each MySQL Proxy plugin can be written in order to expose hooks to the scripting layer that get called where needed.

The proxy-plugin exposes some hooks that are called on different stages of the communication between the client and the backend. A typical control flow is show in (Figure 5.5). On proxy-plugin the hooks allow changing the normal connection lifecycle allowing the features of never connecting to a backend, replace or inject commands and even replace responses.

Proxy plugin Hooks

connect_server Intercepts the connect() call to the backend server. If it returns nothing the proxy connects to the backend using the standard backend selection algorithm. On the other side if it is set to return `proxy.PROXY_SEND_RESULT` is doesn't connect to the backend, but returns the content of `proxy.response` to the client.

read_auth Reads the authentication packet sent by the backend as a string. If it return nothing it forwards the authentication packet to the client. On the other hand it can be set to replace the backends packet with the content of `proxy.response` if set to return `proxy.PROXY_SEND_RESULT`.

read_auth_result

Similar to the read_auth function, except it is for the next stage of the authentication

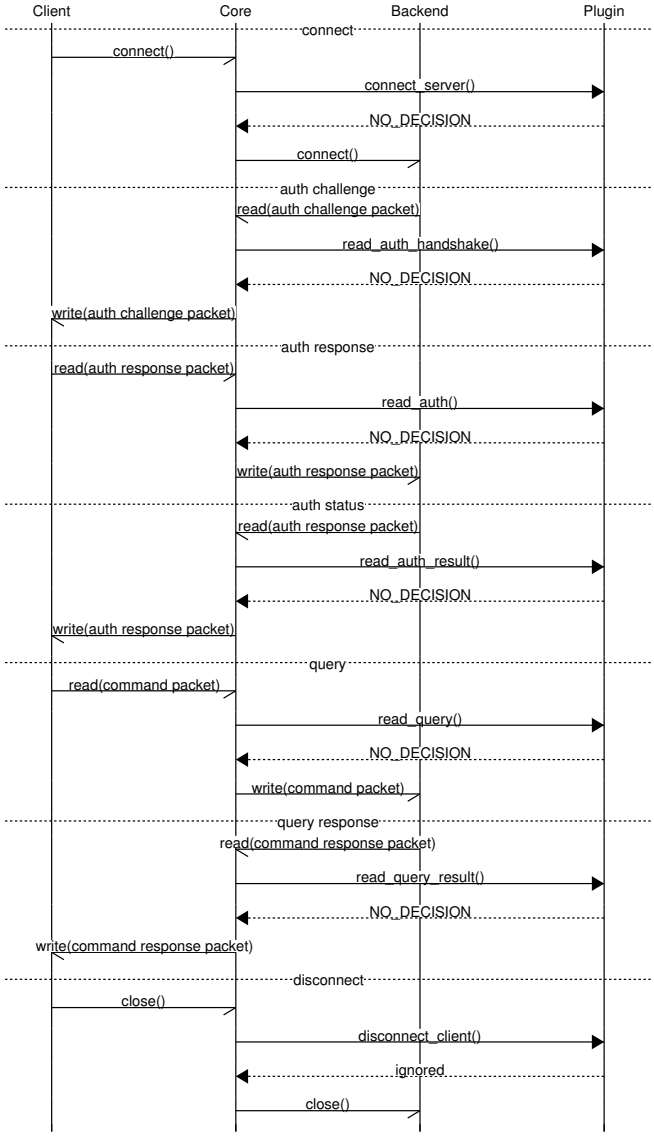

Figure 5.5: Proxy Plugin hooks control flow

cycle. It can replace the clients packet with the content of `proxy.response` if set to return `proxy.PROXY_SEND_RESULT`.

read_query

Reads the command/query packet as a string. If nothing is set it forwards the command packet do the backend. It can send the first packet of `proxy.queries` to the backend if set to return `proxy.PROXY_SEND_QUERY`. This allows to add queries to the proxy.queries structure and inject them on the backend. It can also send to the client the content of `proxy.response` and do not send nothing to the client as the normal connection cycle does.

read_query_result

Intercepts the response sent to a command packet. If nothing set it forwards the result-set to the client. If return set to `proxy.PROXY_SEND_RESULT` it sends to the client the content of `proxy.responde`, and with `proxy.PROXY_IGNORE_RESULT` it doesn't send the result-set to the client.

disconnect_client

Intercepts the `close()` function of the client connection.

5.6 Summary

In this chapter we have presented the software tool MySQL Proxy. We have introduced and documented the characteristics of this tool underlining the fact that it operates similarly as Man in the Middle standing between servers and clients, and also between servers and servers namely master and slave instances. This characteristic makes it feasible to implement either active and passive replication transparently using MySQL Proxy to intercept requests.

Inferring this property and taking advantage of the group communication characteristics we are headed to the main motivation of this work; proposing a mechanism of updated distribution that allows thousands of hundreds of replicas. Bringing all the pieces together we can propose a mechanism based on group communication, using the tool MySQL Proxy in order to accomplish our goal.

Chapter 6

Replication Plugins Using Group Communication

MySQL Proxy provides us with the possibility to stand between server and clients, master and replicas and to infer it operation with the use of plugins. To accomplish our goal we propose and implement solutions based on the development of plugins to this software tool incorporating group communication. In this chapter we present and discuss the problems and corresponding resolutions and the proposed implementations of group based replication using MySQL Proxy.

6.1 General Approach

As previously presented and discussed, MySQL Proxy is a software tool that can sit between a MySQL client and a MySQL server inspecting transforming and acting on data sent through it. Taking into account these functionalities the Proxy can be described as man in the middle. Also, is has the very interesting and useful ability to handle the MySQL protocol. Our proposal is to exploit these properties in order to develop plugins to allow group based replication through it.

As stated on Chapter 2, MySQL replication mechanism relies on a binary log that keeps track of all changes made to a database. The master instance records the changes to its data and records it on a binary log being each of the records called a binary log event. The slaves copy those binary log events to itself in order to apply them on its own data. As so, in order to implement a mechanism of update distribution using a Group Communication protocol and the binary logging MySQL mechanism, we needed to intercept the whole communication that goes through the replication stream on the MySQL DBMS and distribute it through the Spread toolkit. In the context of having an

application "man in the middle" that puts itself between master and slaves on the normal replication workflow, appears the tool MySQL Proxy, as described on the previous chapter. This software tool besides being open-source it has a plugin layer, permitting the development of plugins and in our specific case plugins to intercept replication and allow group communication using the Spread Toolkit.

In order to distribute and receive the updates from the group communication system, a mechanism to allow the use of the Spread Toolkit in MySQL Proxy was developed. This mechanism, using the C API provided, enables sending and receiving messages using the group communication protocol. With this mechanism the Proxy can send and receive messages through the Spread toolkit containing the updates or binary log events.

With this mechanism fully functional and depending on where do we place MySQL Proxy and what it intercepts, we make state-machine or primary-copy replication through plugins of MySQL Proxy.

6.2 Active Replication

Having completed the challenge of having a mechanism fully integrated on MySQL Proxy to use group communication our following step and first approach was to overcome the single master limitation of MySQL. On the scope of the data freshness issues presented on this work, overcoming the limitation on MySQL of having a single master for each replica has a strong outcome on update distribution delay. On circular and alike topologies the impact of the updates not requiring to pass through a series of hops is significant. As so, taking into account these assumptions and setting aside the possibility of distributing the binlog event containing the updates we base our first approach on the Active replication method.

With the Spread mechanism the Proxy can send and receive messages through the Spread toolkit containing the updates. This way an update can be propagated to all the replicas obeying the properties of group communication. However in order to detect and propagate an update on the master side and to receive and apply an update on the replica side, a mechanism to accomplish this task is needed.

A naive and simple approach to solve this need is to use a Lua script to work with the proxy plugin leveraging the capabilities of the Lua hooks and using the `read_query_result` function. Using a script that never returns from the `read_query_result` and since this function polls on a blocking queue the script can wait until it gets a query form it and so passing it to the injection queue and finally returning from the callback when the query is done. This way, using the spread mechanism it can fill the queue with the received

queries from the group communication system using Lua or C global function.

However, since each plugin lives on a separate process each one will have its own connection handler. Thus, the spread plugin will not have any connection to use for packet injection as this approach states. A possible solution for this issue is to add the spread mechanism to the proxy plugin, thus allowing it to send and receive messages. However, on the replica side of the approach there will be no client connection since the query source is the spread group communication system. As so, it is impossible to inject the queries on the injection queue since there is no connection and the state machine was not initialized. A solution to this problem is to fake a connection and make it connect to the respective backend in order to inject the queries received from Spread.

Even though MySQL Proxy has the ability to handle completely the MySQL network protocol, faking the connection is not trivial since we need to send the correct MySQL packets so that the state machine enters the correct states until it reaches the `CON_STATE_SEND_QUERY` state. To accomplish such task a viable solution is to create a `socketpair()` and use that socket as a client and on the other side of it the spread handling code. That way, a network can be triggered to wake up the network handler thread. This can be done by for example writing a single byte to the socket so that the `network_mysqld_con_accept(int G_GNUC_UNUSED event_fd, short events, void *user_data)` creates a connection and starts the state machine. However, several issues need to be tackled to make this approach viable. The possibility that the connection is dropped can be an issue, and the creating and handling of the MySQL packets is not straightforward.

6.2.1 Lua bindings

Spread can be used by Lua scripts by means of an existing package, a pack of bindings for the Spread Group Communication System for several programming languages: C++, Lua, Perl, Python and Ruby.[1] These bindings allow us to use the Spread API in a very simple way through Lua scripts. On the other hand, we need to handle events on the Lua scripting layer. This was achieve with a package of libevent bindings for the Lua programming language.[2]. Using this tools, we can set a new base event with the result of the `event_init()` function on the chassis main loop, and add the necessary callbacks:

```
1 require("luaevent")
2 base = luaevent.core.new()
3 base:addevent(mbox:descriptor(), luaevent.core.EV_READ,
```

[1] http://www.savarese.com/software/libssrcspread/
[2] http://code.matthewwild.co.uk/luaevent-prosody

```
4  readmsg, nil)
```

And with the Spread bindings we can initialize, send and receive messages:

```
1  require("ssrc.spread")
2
3  mbox = ssrc.spread.Mailbox()
4  mbox:join("repl")
5
6  --send message:
7
8  message = ssrc.spread.Message()
9
10 message:write(q)
11    mbox:send(message, "repl")
12
13
14 --receive message:
15
16 function readmsg(ev)
17   mbox:receive()
18 end
```

So, we need a way to get the luaevent defined callbacks to be called on the proxy plugin. Using the global Lua varible `proxy.global` we can set any type of variable. This works as a Lua table, i.e. an hashtable, so we can retain the result of the `event_init` of the chassiss main loop and pass it to the luaevent bindings. However, this is not straightforward and error prone.

6.2.2 Challenges

The basis of all the approaches is to call the `query_injection()` function to add queries on the injection queue so that they are injected on the MySQL server backend. But calling this function is not so straightforward. Analyzing the proxy plugin workflow we can infer that:

The plugin starts with `network_mysqld_proxy_plugin_apply_config()`. It starts an connection for the listening socket and calls the `network_mysqld_proxy_connection_init` hook to it. This function register several callbacks like for example `con->plugins.con-_read_query = proxy_read_query;`. Following this callbacks registration it calls `network-_mysqld_lua_setup_global()`. This function, defined on `network-mysqld-lua.c` pushes

to the Lua stack the global name proxy. If this one does not exist, it calls network_mysqld_lua_init_global_fenv() and creates the global object proxy with all the necessary properties, functions and variables including the returns definitions like for example the PROXY_SEND_QUERY. If it exists it registers on the proxy.global the backends array. Finally it is made the callback register for the listening socket, with the function: event_set(&(listen_sock->event),listen_sock->fd, EV_READ|EV_PERSIST, network_mysqld_con_accept, con);. The handler network_mysqld_con_accept accepts the connection and opens a client connection. Finally it is added another event handler for this connection: network_mysqld_con_handle(). This function implements the core state machine that does all the states handling for the connection. Including the CON_STATE_SEND_QUERY that writes an query to the socket server.

Using an example we can follow the workflow of the state machine. So, the read_query function it is called when the connection state machine is on the CON_STATE_READ_QUERY state.

What this function does is an switch of an state (ret), that comes from ret = proxy_lua_read_query(con). proxy_lua_read_query() resets the injection queries queue and calls network_mysqld_con_lua_register_callback(con, con->config->lua_script) that setups the local script environment before the hook function is called. After it, it does the loading of the Lua script network_mysqld_lua_load_script(sc, lua_script) and setups the global Lua tables with network_mysqld_lua_setup_global(sc->L, g). If everything goes without problems until this point it then call the read_query callback lua_getfield_literal(L, -1, C("read_query")). Depending on the return value of this function it passes it to the proxy_lua_read_query function.

In other words, we can report that for each state machine state it is called an callback. As we can see on network-mysqld.c:

```
1 case CON_STATE_INIT:
2   switch (plugin_call(srv, con, con−>state))
```

The plugin_call function it is an macro that will execute the registered callback con->plugins.con_init = proxy_init defined on the proxy plugin with NETWORK_MYSQLD_PLUGIN_PROTO(proxy_init). On the end this callback will switch the state to CON_STATE_CONNECT_SERVER and so on. Thus, on the read_query() example when the state is in CON_STATE_READ_QUERY the Lua script is loaded and then the callback is made.

Other challenge is to load the Lua script in order to call the quer_injection() function. Loading the script with lua_pcall(L, 1, 1, 0) destroys the global Lua state. So after studying these problems, some new solutions have emerged. Being the first one to create a new state for the state machine. This state is returned after read_query so that af-

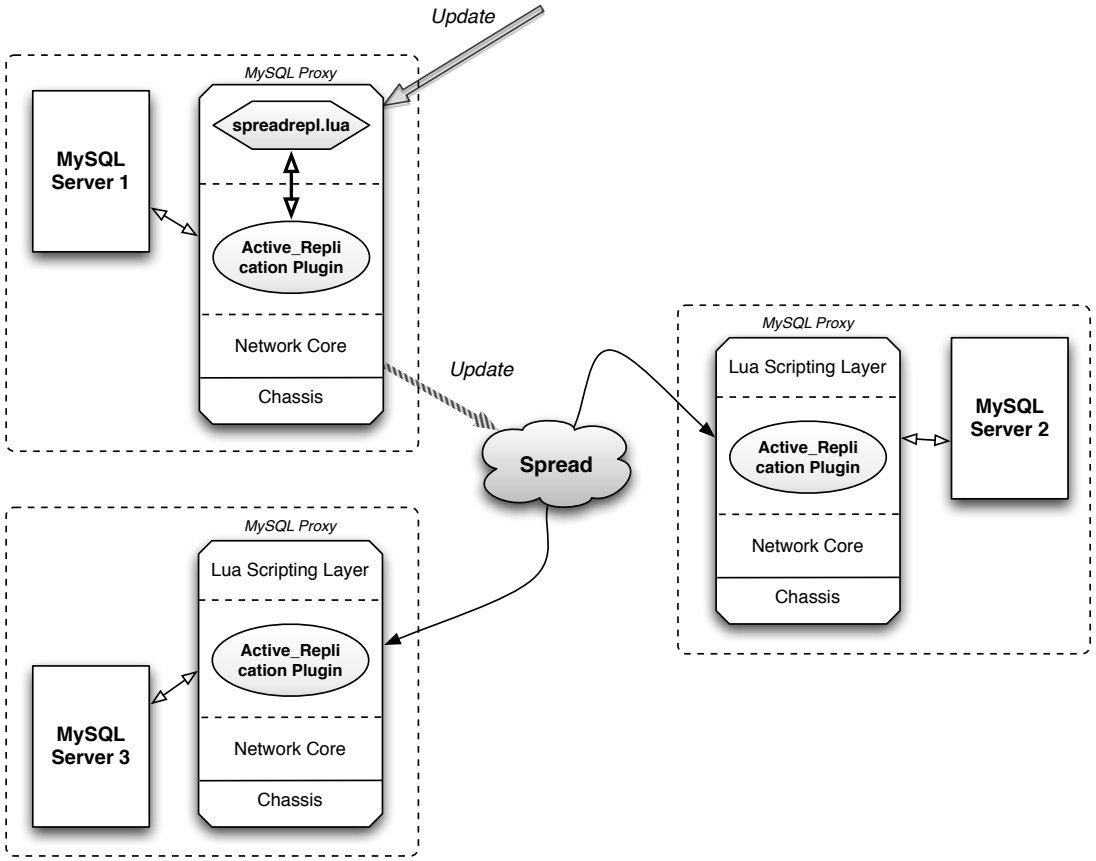

Figure 6.1: Active replication plugin architecture

ter receiving an query sends it through Spread, and after receiving an query from Spread calls the `query_injection` function. This way a new callback on the `network_mysqld_proxy_connection_init` can be added. However, besides being an complex solution it brings changes on the network core that are not desired since the purpose is to develop plugins to work with MySQL Proxy and not to change it's core components.

6.2.3 Solution

The approach developed to overcome all the difficulties presented above is to use the proxy plugin features together with the spread plugin. However, it does not inject the queries on the query injection queue as described above. (Figure 6.1) illustrates our approach based on the state-machine replication model.

On the master side it uses the proxy plugin features to detect new query events on the connection and using the Spread API and the ability to call C functions from the

Lua scripting Layer it broadcasts the query events as a Spread message. In contrast, on the replicant side it works by receiving the queries from the Spread Group Communication System and applying them on the respective backend. Unfortunately, this is not as straightforward we thought it was taking advantage of the backend connection and just inject the queries.

MySQL does not accept anything that is written to the listening connection, i.e. it is necessary to start with the handshake and authentication. So we needed to simulate an MySQL client to handshake and authenticate with the MySQL server and deal correctly with the state machine. After that it can build and send the respective packets with the query content as normal `COM_QUERY` packets.

To handle new upcoming queries, an event is registered to watch for events on the Spread Mailbox. To avoid having a new connection for each received query and the overhead induced by that, the connection is made when the plugin starts and remains on the `CON_STATE_READ_QUERY` state awaiting for new events.

Summarizing, this model is designed for the active replication scenario, i.e., state machine replication model. In this model the proxy plugin is used with the spread plugin as described above. Thus, in the master side, a query is made directly to the server directly on the proxy and this, through a Lua script executes a function that sends the query content to the Spread toolkit, using the spread mechanism. So the message is sent to all nodes in the system and each one, using the spread plugin, detects a new message and through the proxy plugin injects the query directly on the respective backend.

6.3 Passive Replication

With the active replication approach we overcome the single master limitation of MySQL replication mechanism. In comparison with chain and ring topologies where updates are passed from each replica to the following, distribution delay is decreased since updates are directly distributed and applied on each replica. However, it discards the binary log mechanism properties and advantages.

We define another approach that brings together the group communication and binlog properties. We base this approach on the primary-copy replication model. On this model, replicas apply the changes produced by the primary copy therefore a replica crash is completely transparent to the client. Besides this advantage, it allows nondeterministic operations since it is possible for each replica to have multithreading.

(Figure 6.2) illustrates our approach based on passive replication, i.e., master/slave replication model. This approach takes advantage of the plugins in development by the MySQL Proxy development team: master plugin and replicant plugin.

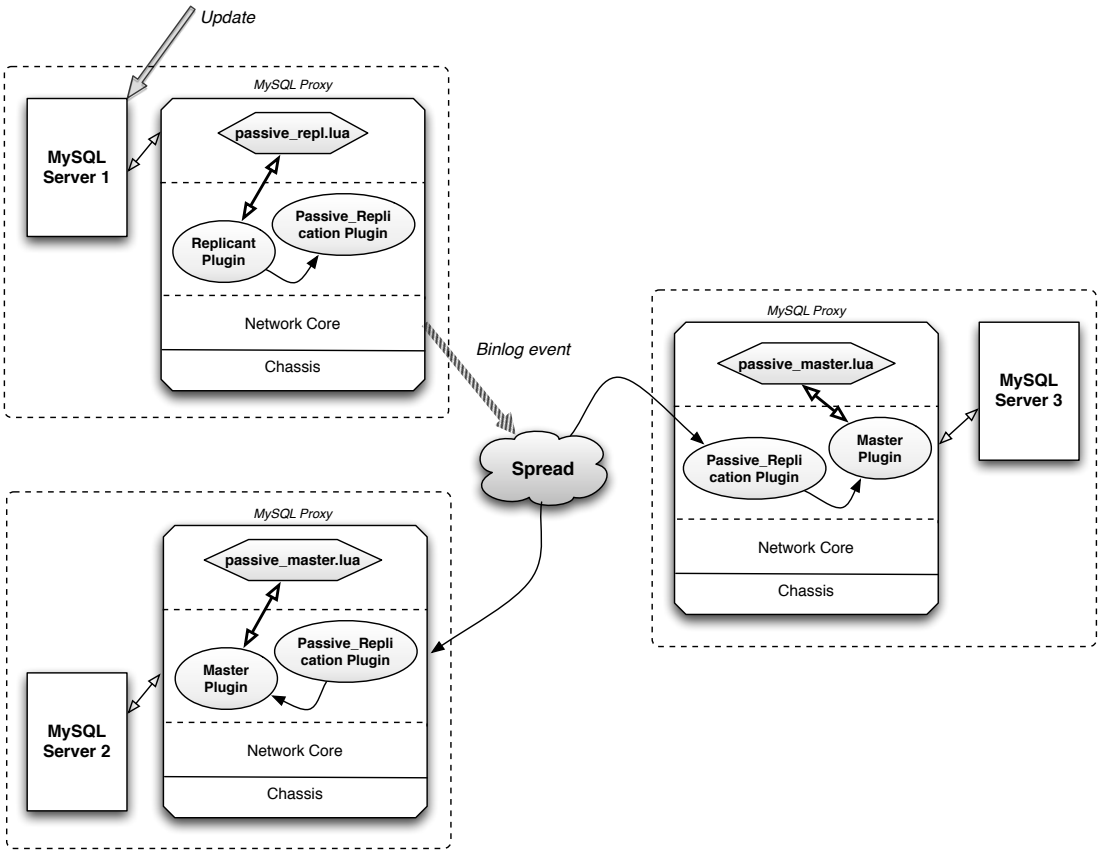

Figure 6.2: Passive replication plugin architecture

Master and replicant plugins simulate a master and a slave, respectively, i.e., the master plugin works as a MySQL server in master mode on the normal MySQL asynchronous replication feature. A replica connects to it, and it does all the necessary connection handling, thus creating the necessary binary log. After that the binary log events are sent through the Lua scripting layer. On the other side of replication, the plugin replicant works as a MySQL slave. A master connects to it, does all the connection handling necessary and it reads a binary log to execute well on the respective backend.

Therefore, in this approach, the master-side proxy uses the plugin replicant to intercept the binary log events that are to be written to the binary log and passes this information to the spread mechanism in order to broadcast these event to the Spread Group Communication System.

On the replicas side, the proxy works with the master plugin in order to expose these binary log events to the MySQL backend, working this way as a master on the normal asynchronous replication model. The proxy plugin is then used to receive these events

from the group to which is connection and forward them to master plugin so that these events are exposed to the MySQL backend.

Limitations of master and replication plugins of MySQL Proxy delayed and even interrupted for the moment the implementation of the passive replication plugin. More precisely, on the replicant plugin, several aspects are incomplete. Namely, the mechanism for reading binary log events from a file, Lua script or some buffer and expose them as binlog-streams is not yet implemented and fully usable. Regarding the replicant plugin, it is also incomplete taking into account that the mechanism to parse the binlog-streams sent by the server and to buffer or append them to the corresponding relay binary log is also not yet implemented and fully usable.

However, it is important to note that the communication protocol of MySQL between master and slave is the same as client and server. As so, with the replicant and master plugins fully functional the challenge is to inject data obtained from the group communication system. This issue was already exceeded with active replication plugin.

6.4 Recovery

In order to maintain throughput and fault-tolerance the system must replace any failed replica without stopping the services provided. When a replica fails, the reconfiguration of the cluster is necessary in order to restore the resiliency of the system [33].

Our approach to accomplish these goal starts when a new view is delivered on the Group Communication System meaning that a new replica joined the group. Two similar methods were discussed and proposed to accomplish our goal on both approaches of passive and active replication.

Both methods have the binary logs option enabled on each server, even though on the active method it is not necessary it became essential for the recovery mechanism. For the active replication approach, the recovery mechanism runs as follows. When a new replica joins the system, it firstly obtains its own binlog coordinates and it elects a server from the group based on proximity. Afterwards it asks to start replication from the log coordinates obtained. In the meanwhile, since we are dealing with a state-machine protocol in which updates are multicasted to each node, and taking into account that stoping replication is not desirable, the new replica starts a buffering mechanism to store the incoming updates while the recovering process is not finished. Upon the end of the process of applying the binlog asked by the new replica, it starts reading and applying the events stored on the buffer log. The recovery process is said to be finished when the buffer log is empty, and from then on the new replica switches to the replication protocol by receiving the updates directly form the Group Communication System.

On the passive replication approach, the recovery mechanism runs similarly as above described. When a new replica joins the system it also obtains its own binlog coordinates and starts replication from that position. In the meanwhile it buffers the binlog updates received from the replication mechanism. A naive approach can be to use the master plugin in order to create a binlog and after the recovery process is finished the new replica can change its connection to the master plugin. However, this is not possible since the master plugin does not have the ability to create binary logs being only able to expose the binlog event as a stream simulating a master instance as on the MySQL's replication mechanism. As so, the recovery protocol uses a buffering mechanism to store the binlog events while the recovering process runs. Again, similarly as the behaviour of the recovery mechanism for the active replication method described above, upon the end of the process of applying the binlog the new replica master plugin starts reading and applying the binlog events stored on the buffer log file. The recovery process ends by the moment the buffer log is empty, and from then on the new replica master plugin changes it input to the Group Communication System.

The convergence phase of reading the buffer file in contrast with the exponential growing of it, since the replicated database is not stopped, can leave to discredit. However, as shown by [33], if the system is well configured and normally functioning the buffering will converge attaining the empty state. Upon reaching that state, switching to the replication protocol developed is straightforward.

Regarding failures, for both cases if the new replica fails during the recovery it aborts. On the other side, if the replica on which the new replica connects and asks for the binlog dump to start on the last binlog coordinate fails, the recovering replica elects a new server and obtains its own last binlog coordinated in order to start the recovering process again.

6.5 Summary

This chapter describes the major contributions of this thesis. The active and passive replication approaches implemented using MySQL Proxy and the Spread Group Communication Toolkit.

We start by presenting the general approach to achieve this goal, naming the conclusions and inferences made from the previous chapters, and stating the steps and mechanisms needed in order to implement the passive and active replication plugins for MySQL Proxy.

Having the general approach well defined, we present both implementations of active and passive replication. Starting with active replication, we describe and discuss the several challenges and problems we had to tackle in order to achieve the correct and

working protocol. Afterwards, we present the passive replication approach describing its behaviour.

Finally, and of great importance, we describe the recovery protocol. Taking into account possible failures of replicas, we tackle them without stopping the services provided by the system. We describe the behaviour of the recovery protocol for both active and passive replication replication, focusing on the re-joining of a previous failed replica to the system.

Chapter 7

Results and Performance Analysis

The purpose of this chapter is to present and evaluate the proposed group communication database replication model and its costs using a workload widely used to measure the performance of commercial database servers. Special attention is payed to the performance evaluation of the developed system in contrast to the replication model provided by the MySQL DBMS.

7.1 Motivation

In Chapter 6 we describe the proposed and implemented solution based on MySQL Proxy. To evaluate the impact, performance and trade-offs of the solution we have committed to answer and provide real proofs for several questions related to the behaviour of the solution. Questions such as:

- How does the solution behave in comparison with the MySQL's replication using the topologies of Chain and Master and Multiple Slaves.
- What is the impact of performance with different number of TPC-C clients, and various values of think-times.
- What is the impact of performance with different types of Spread Messages: FIFO and AGREED.
- How does the solution behave with a different number of replicas.

7.2 Workload and Setting

In order to assess the distributed database used in the case study, we have chosen the workload model defined by the TPC-C benchmark [1]. We have used the same implementation used on the benchmarks described on chapter 4.

To obtain the delay values of the MySQL's replication method, two replication schemes were installed and configured. A six machines topology of master and multiple slaves, and a six machine topology in daisy chain.

The hardware used included six HP Intel(R) Core(TM)2 CPU 6400 - 2.13GHz processor machines, each one with one GByte of RAM and SATA disk drive. The operating system used is Linux, kernel 2.6.31-14, from Ubuntu Server with ext4 filesystem, and the database engine used is MySQL 5.1.54. All machines are connected through a LAN, and are named PD00 to PD07. Being PD00 the master instance, PD04 the remote machine in which the interrogation client executes, and the others the slave instances.

The following benchmarks were done using the workload TPC-C with the scale factor (warehouses) of two, number of database connections (clients) twenty, forty, sixty, eighty and one hundred and the duration of 10 minutes.

7.3 Experimental Results

Having the workload and setting defined, we have committed to evaluate the impact, performance and trade-offs on several different scenarios in order to answer the questions stated above.

Database connections (clients)	20	40	60	80	100
NOTPM (default think time)	26.29	24.34	25.28	25.48	25.27
NOTPM (half think time)	50.60	50.98	51.10	48.40	53.09

Table 7.1: TPC-C new-order transactions per minute for master and multiple slaves topology

Replica	PD01	PD02	PD03	PD05	PD06	PD07
Number of samples	408	394	402	408	390	399
Average delay (μs)	323	279	288	292	276	287
99% confidence interval ($\pm$)	150	145	139	135	141	140

Table 7.2: Results for master and multiple slaves topology with 100 clients.

7.3.1 MySQL Replication

Master and Multiples Slaves

Results obtained with 100 TPC-C clients, default think time of the DBT-2 TPC-C and the Master and Multiple Slaves topology are presented in (Table 7.2). It can be observed that all replicas get similar results and that the propagation delay is consistently measured close to 10 μs with a small variability.

Results obtained for the same replication scheme and DBT-2 think time but with varying number of TPC-C clients, are presented in (Figure 7.1). They show that propagation delay is similar between replicas and has little variation with the load imposed on the master re-affirming the propagation delay is similar between replicas and the master in a master and multiple slaves topology.

Same set of results but for half of the think time of DBT-2 TPC-C are presented in (Figure 7.2).

It can be observed that all replicas get similar results as in the previous set. We can state the same previous conclusions, being the only difference that the delay is slightly superior, in the overall. This can be justified with the largest value of new order transactions per minute as shown on (Table 7.1) that presents the values obtained from the DBT-2 TPC-C benchmark regarding the new-order transactions per minute for the varying number of clients.

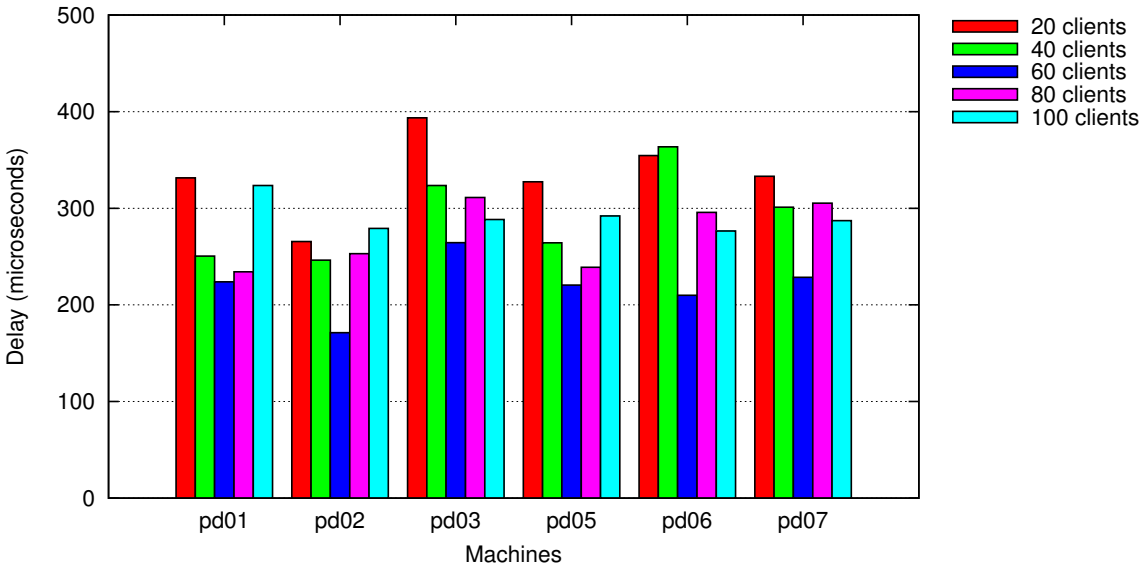

Figure 7.1: Replication delay values for Master and Multiple Slaves topology (default think-time)

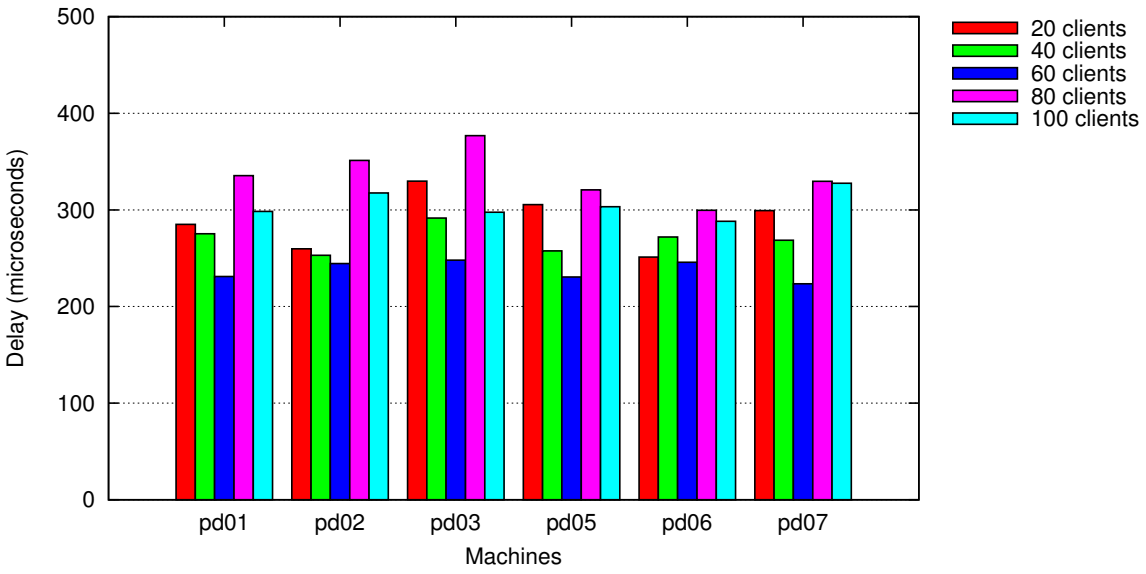

Figure 7.2: Replication delay values for Master and Multiple Slaves topology (half think-time)

Database connections (clients)	20	40	60	80	100
NOTPM (default think time)	23.94	24.60	26.30	27.05	25.79
NOTPM (half think time)	54.19	50.90	52.20	53.10	50.88

Table 7.3: TPC-C new-order transactions per minute for chain topology

Replica	PD01	PD02	PD03	PD05	PD06	PD07
Number of samples	370	362	365	367	368	365
Average delay (μs)	376	369	457	596	655	747
99% confidence interval ($\pm$)	188	275	333	428	489	585

Table 7.4: Results for chain topology with 100 clients.

Chain

Results obtained with 100 TPC-C clients and the chain topology are presented in (Table 7.4). In contrast to master and multiple slaves, the delay now grows as the replica is farther away for the master. This configuration also gives an indication of how the ring topology would perform: As any replica would be, on average, half way to other masters, one should expect the same delay as observed here on replicas PD03 and PD05.

Results obtained with varying number of TPC-C clients, default think time of the DBT-2 TPC-C implementation and the Chain scheme are presented in (Figure 7.3). Results confirm that propagation delay grows as the replica is farther away from the master. In contrast to the previous experiments with ext3 filesystem, we can not conclude the propagation delay grows substantially with the load imposed on the master. However, results show that using the chain configuration for write scalability with suffer the same problem, thus limiting its usefulness.

Same set of results but for half of the think time of DBT-2 TPC-C are presented in (Figure 7.4). It can be observed that all replicas get similar results as in the previous set allowing us to state the same previous conclusions

We also present in (Table 7.3) the values obtained from the DBT-2 TPC-C benchmark regarding the new-order transactions per minute for the varying number of clients.

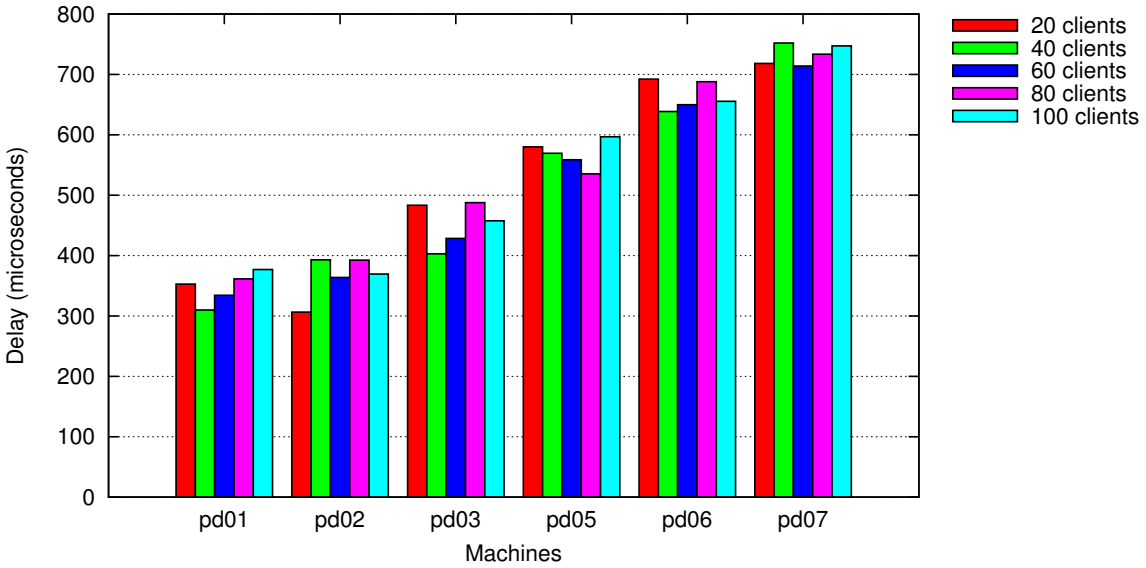

Figure 7.3: Replication delay values for Chain topology (default think-time)

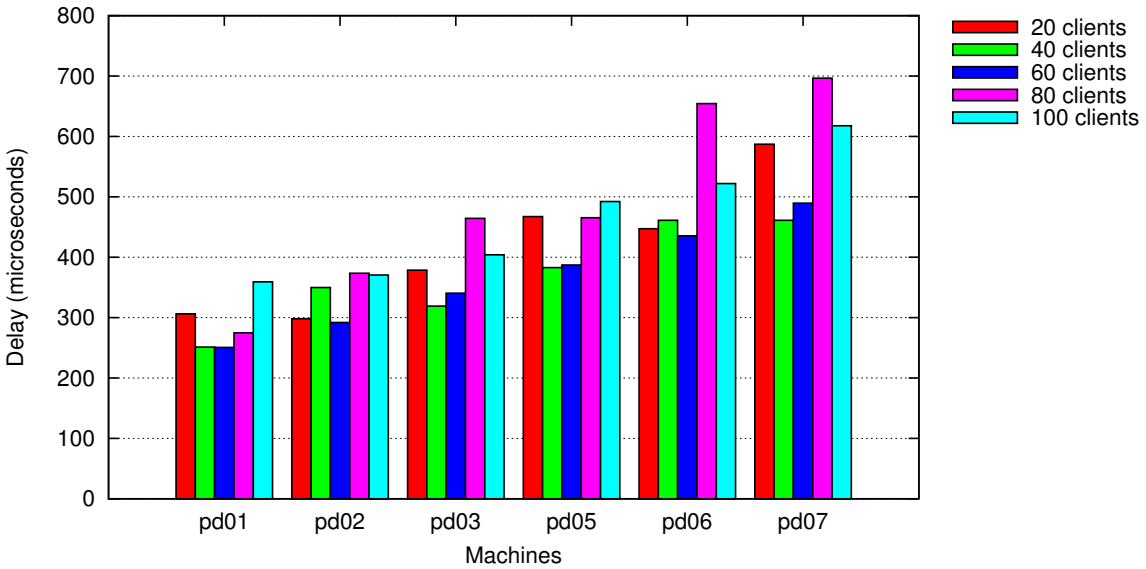

Figure 7.4: Replication delay values for Chain topology (half think-time)

Database connections (clients)	20	40	60	80	100
NOTPM (default think time)	24.64	25.28	25.14	25.64	24.28
NOTPM (half think time)	52.99	50.58	50.48	51.99	51.40

Table 7.5: TPC-C new-order transactions per minute for active replication with Proxy Spread Plugins with FIFO messages

Replica	PD01	PD02	PD03	PD05	PD06	PD07
Number of samples	446	440	445	435	437	450
Average delay (μs)	438	375	435	419	405	445
99% confidence interval ($\pm$)	126	104	130	121	120	126

Table 7.6: Replication delay values for active replication with Proxy Spread plugins with FIFO messages (default think-time) with 100 clients.

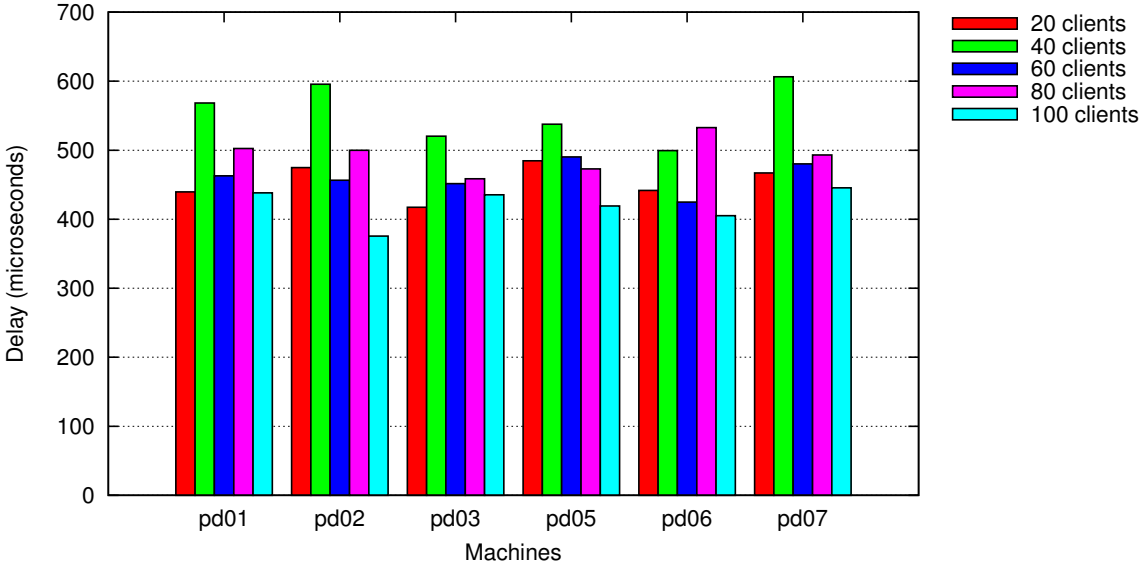

Figure 7.5: Replication delay values for active replication with Proxy Spread plugins with FIFO messages (default think-time)

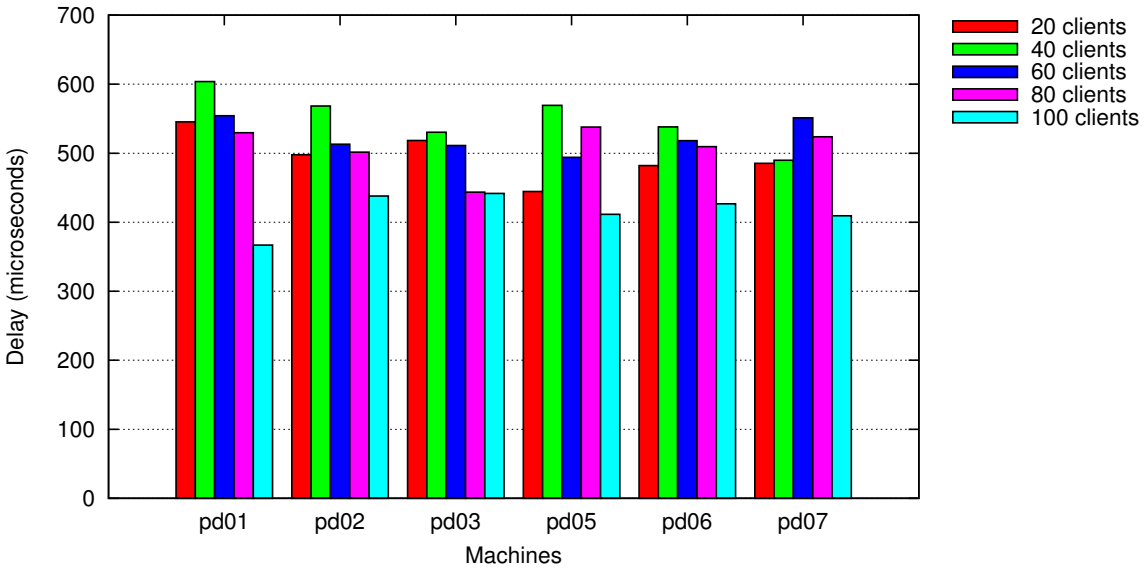

Figure 7.6: Replication delay values for active replication with Proxy Spread plugins with FIFO messages (half think-time)

7.3.2 Proxy Spread Plugins - Active Replication

FIFO Messages

Results obtained with an increasing number of TPC-C clients, default think time of the DBT-2 TPC-C implementation and using the developed solution with active replication and FIFO type of messages are presented in (Figure 7.5). These experiments were using the FIFO messages on the Group Communication protocol. FIFO messages are reliably delivered once to all members of its destination groups, and ordered with all the other FIFO messages form the same source. However, there are no ordering guaranteed of FIFO messages from different sources.

Similarly to the Master and Multiple Slaves scheme using MySQL replication behaviour, using the developed solution it can be observed, as expected according to active replication properties that all replicas get similar get similar results and that the propagation delay has a small variability as shown on (Table 7.6). Propagation delay does not grow substantially with the load imposed on the master.

This results show that propagation delay stands on the same range of values for all the replicas, thus defining the overhead imposed by the solution and the Group Communication protocol. We can observe that the maximum delay observe is about 500 μs. These results gives us the indication that propagation delay will not overcome that range of values, and so using this configuration for write scalability will have substantial gains

in comparison with the ring scheme with the standard MySQL replication. From the fifth replica and forward, performance gains are visible and remarkable.

The same set of results but for half of the think of DBT-2 TPC-C time are presented in (Figure 7.6). In this scenario results are virtually identical, only one can note a slightly increase of propagation delay. Being this in the order of plus 50 μs.

We also present in (Table 7.5) the values obtained from the DBT-2 TPC-C benchmark regarding the new-order transaction per minute for the varying number of clients.

FIFO Messages - Varying number of replicas

In order to assess the impact of a varying number of replicas on the solution results, experiments with the number of 2 and 4 replicas were made. These experiments were set using the same settings used on the previous tests: increasing number of TPC-C clients of the DBT-2 TPC-C implementation, using the developed solution with active replication and FIFO messages.

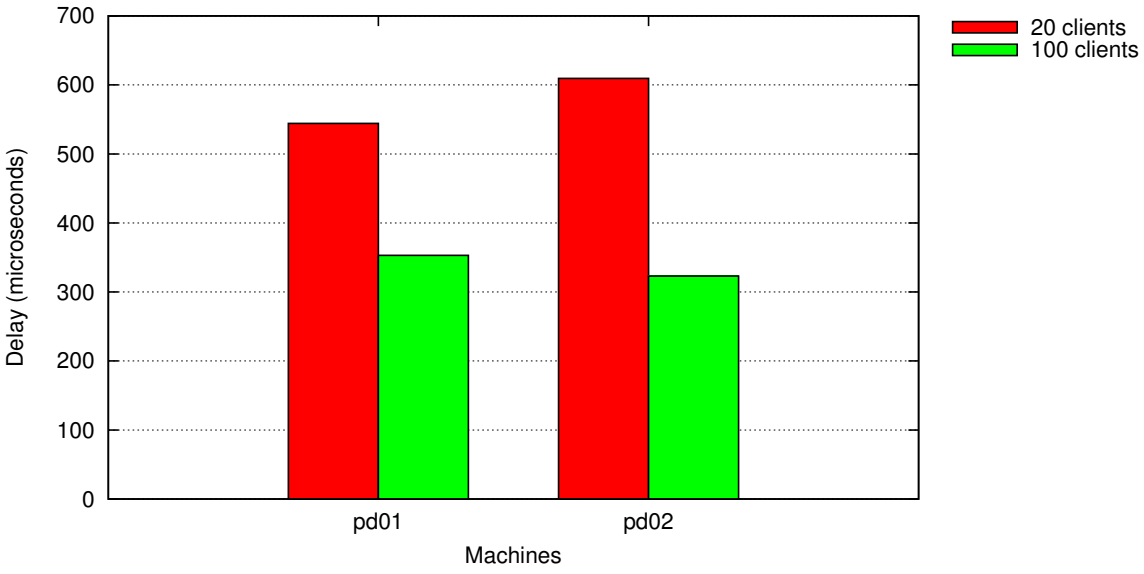

Figure 7.7: Replication delay values for active replication with Proxy Spread plugins with FIFO messages (default think-time, two replicas)

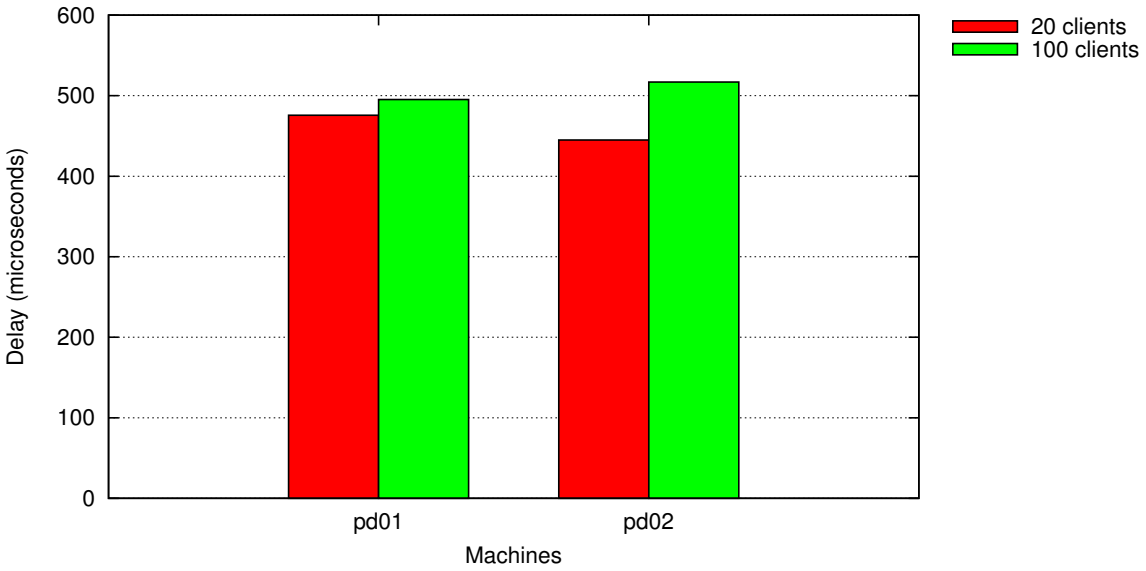

Figure 7.8: Replication delay values for active replication with Proxy Spread plugins with FIFO messages (half think-time, two replicas)

2 Replicas

Results obtained with 2 replicas, default and half think time are presented in (Figure 7.7) and (Figure 7.8). As desired, we can assess that the number of replicas in the configuration does not have impact on the overall performance, being the results similar to the ones obtained with the number of 6 replicas.

4 Replicas

Results obtained with 4 replicas, default and half think time are presented in (Figure 7.9) and (Figure 7.10). Like the previous results for 2 replicas, results obtained with 4 replicas show that the number of replicas in the configuration does not interfere with the overall performance, being the results also similar with the ones with 6 replicas scenario.

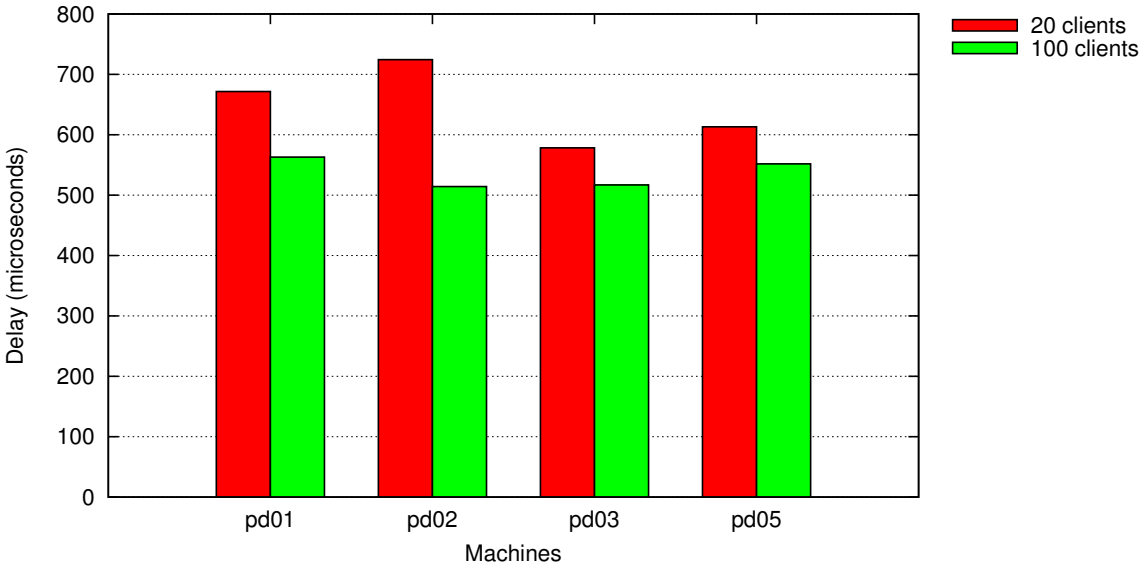

Figure 7.9: Replication delay values for active replication with Proxy Spread plugins with FIFO messages (default think-time, four replicas)

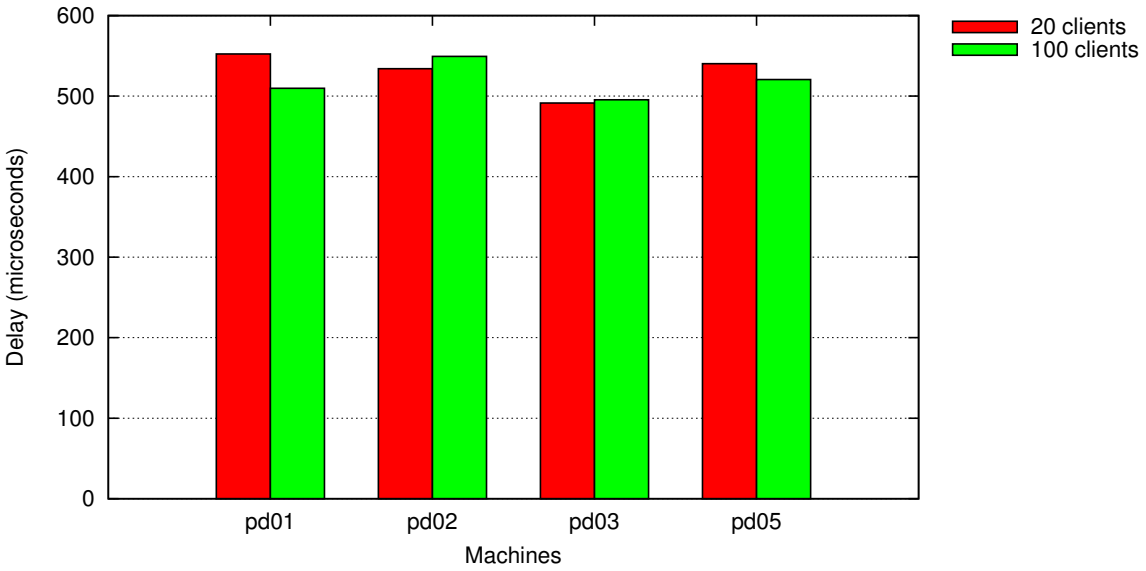

Figure 7.10: Replication delay values for active replication with Proxy Spread plugins with FIFO messages (half think-time, four replicas)

7.3.3 Agreed Messages

Database connections (clients)	20	40	60	80	100
NOTPM (default think time)	24.60	23.34	24.94	25.53	24.58
NOTPM (half think time)	52.69	52.30	52.40	51.60	53.00

Table 7.7: TPC-C new-order transactions per minute for active replication with Proxy Spread Plugins with AGREED messages

Replica	PD01	PD02	PD03	PD05	PD06	PD07
Number of samples	449	435	447	441	448	439
Average delay (μs)	574	474	547	566	540	476
99% confidence interval ($\pm$)	152	130	149	151	147	138

Table 7.8: Replication delay values for active replication with Proxy Spread plugins with AGREED messages (default think-time) with 100 clients.

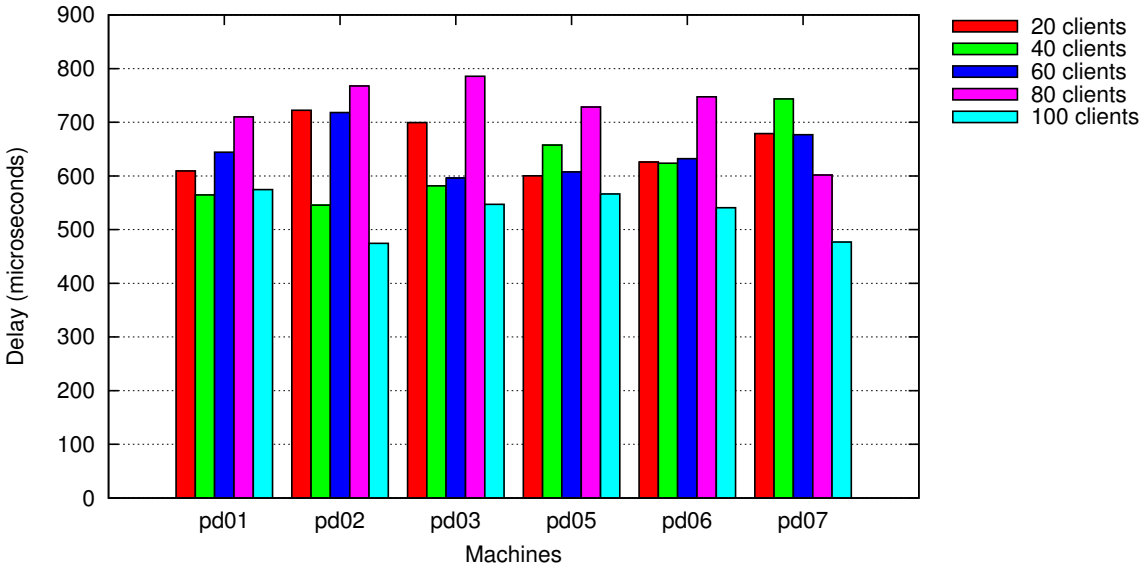

Figure 7.11: Replication delay values for active replication with Proxy Spread plugins with AGREED messages (default think-time)

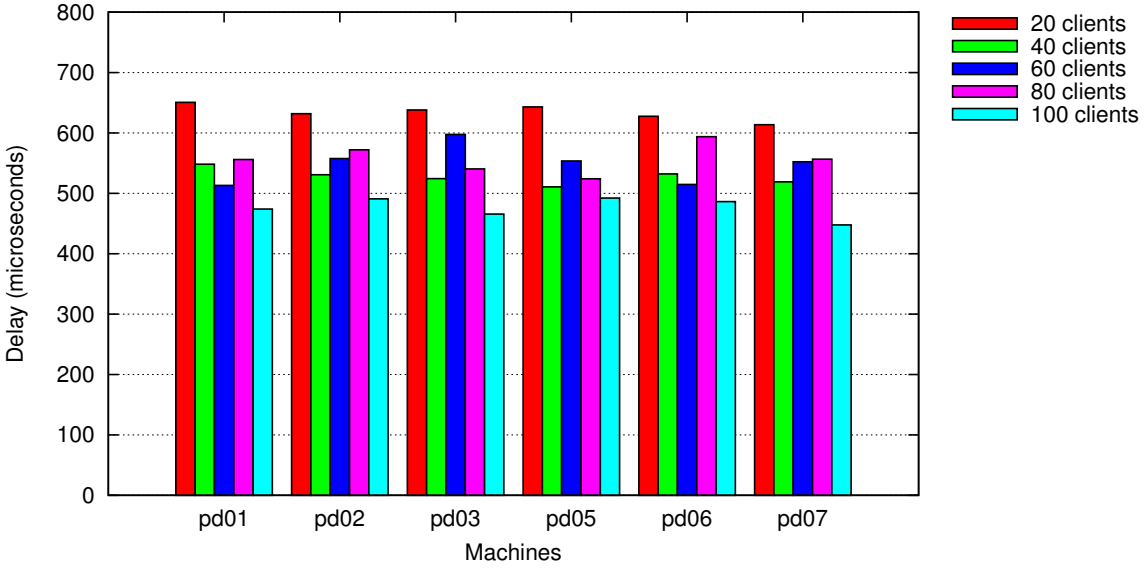

Figure 7.12: Replication delay values for active replication with Proxy Spread plugins with AGREED messages (half think-time)

Results obtained with an increasing number of TPC-C clients, default think time of the DBT-2 TPC-C implementation and using the developed solution with active replication and AGREED type of messages are presented in (Figure 7.11). The descriptive results obtained with 100 TPC-C clients are presented in (Table 7.8). This experiments were similar to the previous showed, however they were using the AGREED messages on the Group Communication protocol. AGREED messages have all the properties of FIFO messages but are delivered in a causal ordering which is the same to all recipients. All the recipients will "agree" on the order of delivery

This results show that the setup using AGREED messages introduces a small overhead. This reflects on a small increase on the propagation delay. However, this increase is of about 100 μs thus reducing the global impact on the performance taking into account the properties gained through the use of AGREED messages on the Group Communication protocol.

The same set of results but for half of the think time are presented in (Figure 7.12). In this scenario results are virtually identical, only one can note a slightly decrease of propagation delay not remarkable.

Like the previous setup with FIFO messages, this results show that propagation delay stands on the same range of values for all replicas. These results, even showing that the propagation delay is slightly larger, gives us the indication that delay will not overcome that range of values and so it has also substantial gains in comparison with standard MySQL replication on the ring schemes for write scalability.

We also present in (Table 7.7) the values obtained from the DBT-2 TPC-C benchmark regarding the new-order transaction per minute for the varying number of clients.

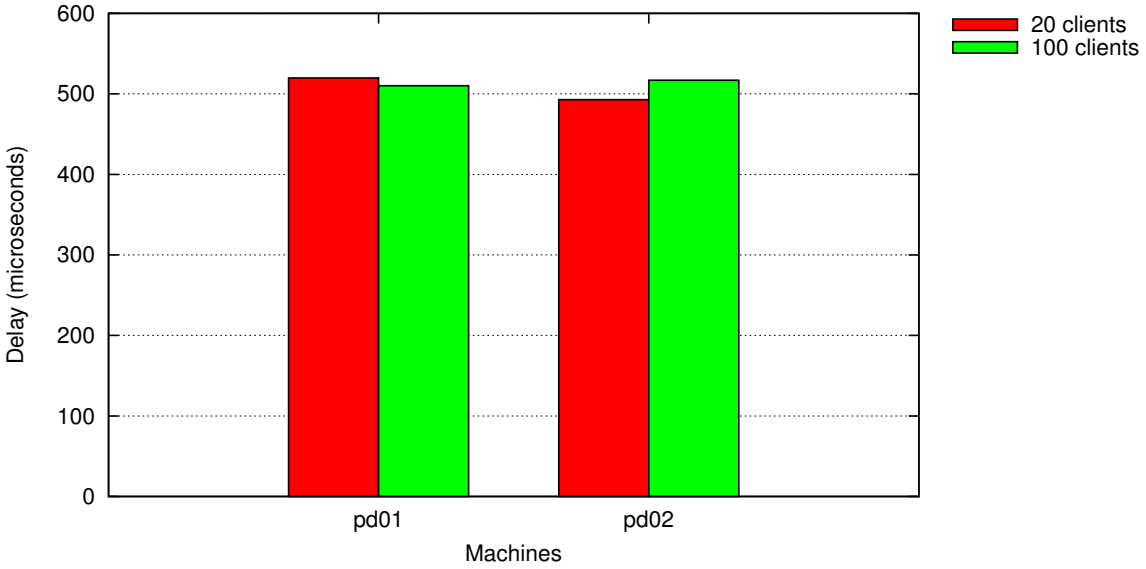

Figure 7.13: Replication delay values for active replication with Proxy Spread plugins with AGREED messages (default think-time, two replicas)

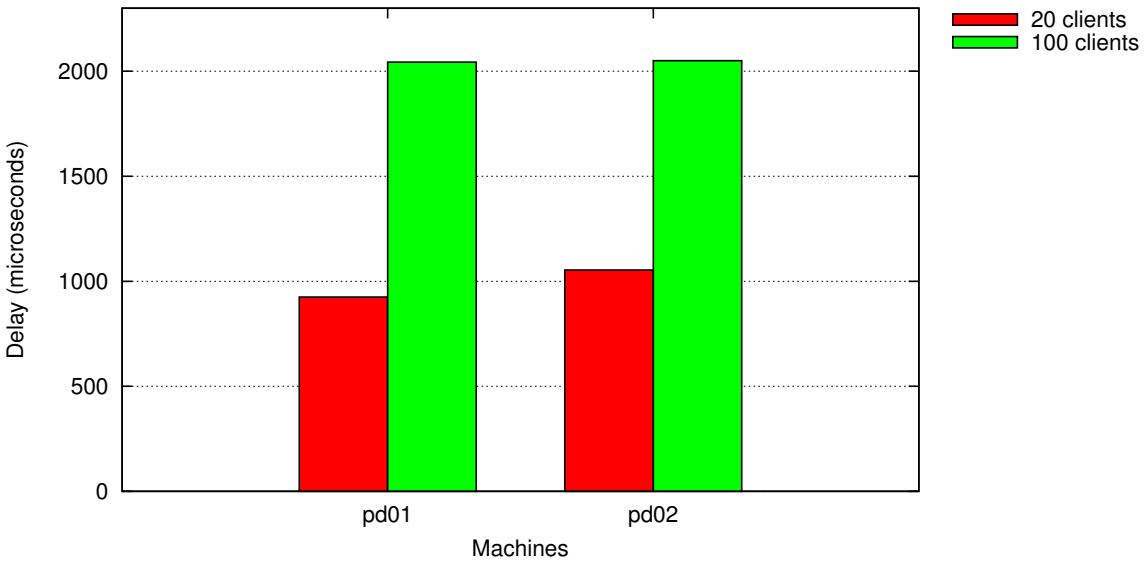

Figure 7.14: Replication delay values for active replication with Proxy Spread plugins with AGREED messages (half think-time, two replicas)

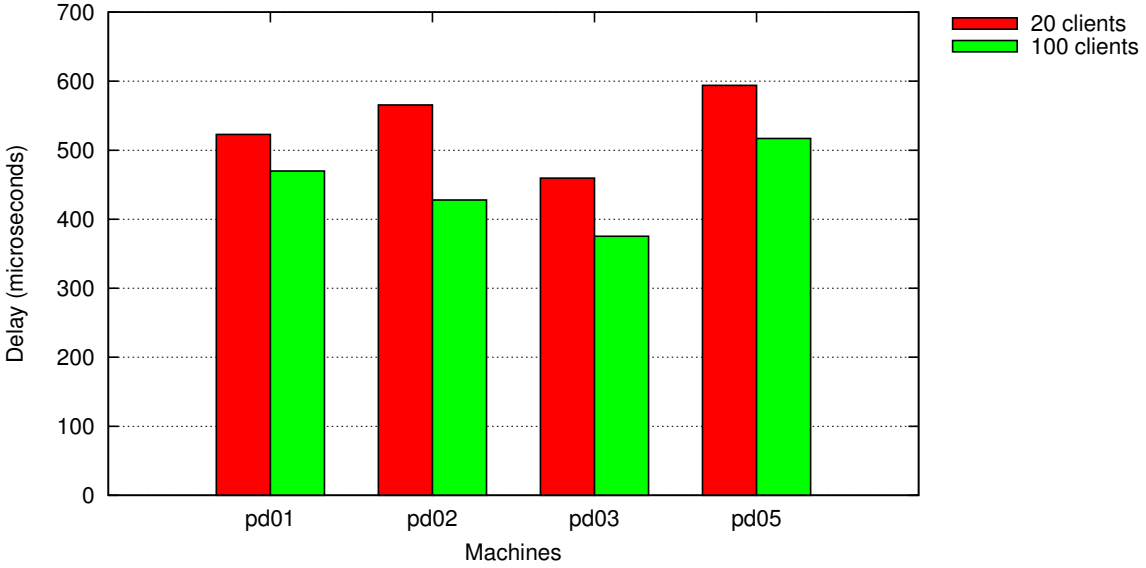

Figure 7.15: Replication delay values for active replication with Proxy Spread plugins with AGREED messages (default think-time, four replicas)

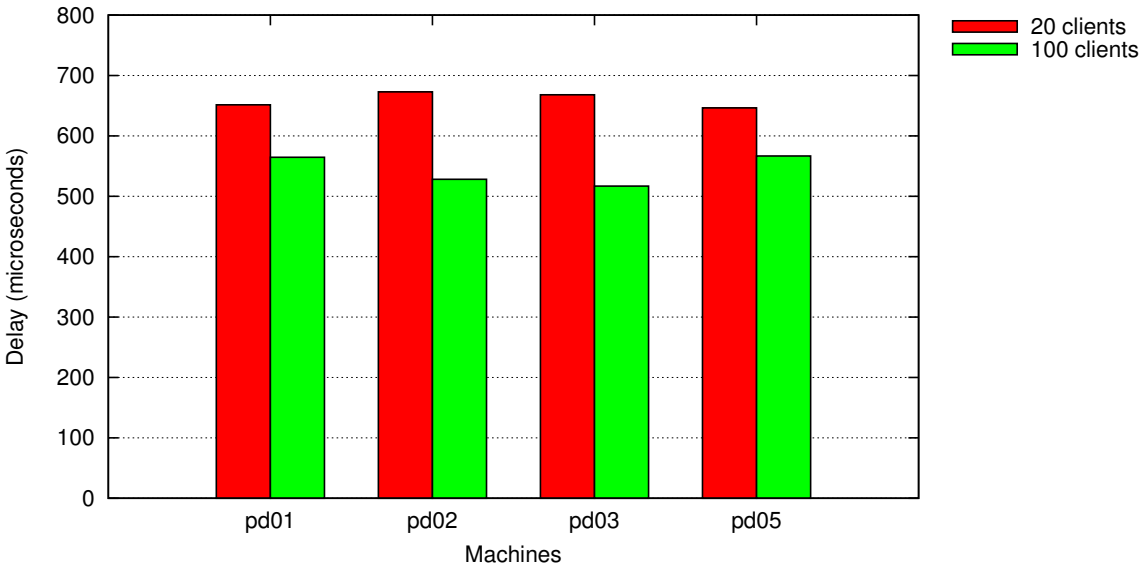

Figure 7.16: Replication delay values for active replication with Proxy Spread plugins with AGREED messages (half think-time, four replicas)

Agreed Messages - Varying number of replicas

2 Replicas

Likewise the FIFO messages setup, in order to assess the impact of a varying number of replicas on the solution results, experiments with the number of 2 and 4 replicas were made. These experiments were set using the same settings used on the previous tests: increasing number of TPC-C clients of the DBT-2 TPC-C implementation, using the developed solution with active replication and FIFO messages.

4 Replicas

Like the previous results for 2 replicas, results obtained with 4 replicas show that the number of replicas in the configuration does not interfere with the overall performance, being the results also similar with the ones with 6 replicas scenario.

7.4 Summary

This chapter presents the usage of the implemented solution in a realist environment, specifically the workload defined by TPC-C. Assumptions of a better performance in comparison to the traditional replication mechanism of MySQL are confirmed by performance results. Also, we could present and discuss the results of active replication in comparison with ring or chain topologies. Besides the visible gain in performance as the hops increase, a great advantage is the possibility of having more than one master.

Chapter 8

Conclusions

Large companies that provide online services guide them according to the information gathered from users and local data. These rapid growing services support a wide range of economic, social and public activities. Due to the need of having this information always available, data must be stored persistently. For that requirement databases are used.

To avoid data loss and unavailability, it is imperative to use replication. But one must take into account that since these large organizations are present in different places and data is stored in wide spread areas, strong consistency and availability are essential requirements. Also, updates to the database must be allowed at any replica. Unfortunately, when these properties are present, traditional replication protocols do not scale. A trade-off between consistency and performance arises.

Group communication primitives provide a framework that promises to reduce the complexity of these problems. Taking into account that group communication provides a reliable, ordered and atomic mechanism of message multicasting, with group-based replication approaches, transactions or updates are propagated to all replicas thus achieving the required fault-tolerance and availability requirements.

This is the context of the proposed approach which aims to solve the database replication scalability and consistency problems detailed previously by introducing a tool that implements and allows state-machine or primary-copy replication suited for large scale replicated databases. In order to do that, we have proposed and developed plugins for the MySQL Proxy software tool. Briefly, two distinct approaches were taken:

- Passive replication;
- Active replication;

The work presented in Chapter 6 demonstrates and specifies how this was achieved.

Particularly, to provide active replication for the MySQL Database Management System, one approach exploits group communication. In the other approach it exploits the traditional binary log replication mechanism of MySQL and group communication to provide passive replication.

Due to limitations of MySQL Proxy we could not obtain results for the passive replication solution, since it does not yet provide the complete mechanism for binary log events parsing and transformation. Therefore, the developed plugins are incomplete even if its architecture and behavior is fully thought and analyzed.

Active replication plugins were fully developed and tested. The implementation details of this approach is shown in Section 6.2, and the workload results are presented in Chapter 7.

The results show that this solution has a substantial performance gain compared with the traditional MySQL replication. The set of conducted experiments have shown that with active replication, the propagation delay stands on the same range of values for all replicas. When comparing with the ring scheme, from the fifth replica on the gains are visible and substantial.

Despite that MySQL Proxy and group communication introduces a visible overhead on replication, when comparing to the tradition replication mechanism of MySQL one notices that the gains are relevant.

We have shown that the solution can overcome the scalability problems of ring topologies without losing the ability to update the database at any replica, using a set of experiments with the realistic workload defined by TPC-C. Note that TPC-C, being write intensive, makes the master server be nearly entirely dedicated to update transactions. This is the case even at a small scale, as in the performed set of experiments. It mimics what would happen in a very large scale setup of database servers.

This work is open-source and available at launchpad. Recently, it was proposed for evaluation and merge on the MySQL Proxy project by the developers team.[1]

8.1 Future Work

This work presents important results for large scale replicated databases. Nevertheless, from the difficulties, knowledge and contributions we realize that there are some features and work that could be done in order to improve substantially not only the results but also the passive replication approach.

Regarding the passive replication plugins, it would be necessary to finish the work

[1] https://code.launchpad.net/~miguelaraujo/mysql-proxy-spread/trunk

started by the MySQL Proxy developers team in order to have the *replicant* and *master* plugins fully functional. Taking into account that having these plugins working, we could complete the symbiosis with the spread plugins in order to provide the passive replication mechanism to MySQL.

Although MySQL Proxy implements a threaded network I/O since version 0.8, further development on the spread plugins and on MySQL Proxy would be necessary, since transactions order could be affected by multithreading. Having several threads to handle the transactions could improve performance significantly, but the ordering trade-off would rise since it would allow transactions to be processed in a different orders that were applied. It would be interesting to evaluate the performance gains and trade-offs and propose a solution to allow the threaded network I/O option on MySQL Proxy.

Finally, taking into account that MySQL Proxy developer team announced that the next version will implement a multi-threaded approach to the scripting layer, we could exploit this to improve performance in the field of message multicasting. Having multithreading on the Lua script layer, performance gains could be visible regarding that in our solution, Lua is responsible for handling and calling the message multicast containing the transactions or updates.

Bibliography

[1] Transaction processing performance council. TPC Benchmark^{TM} C standard specification revision 5.11, February 2010.

[2] M. Amir, L. E. Moser, Y. Amir, P. M. Melliar-smith, and D. A. Agarwal. Extended virtual synchrony. In *in Proceedings of the IEEE 14th International Conference on Distributed Computing Systems*, pages 56–65. IEEE Computer Society Press, 1994.

[3] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and J. Stanton. The spread toolkit: Architecture and performance. Technical Report CNDS-2004-1, Johns Hopkins University Center for Networking and Distributed Systems, 2004.

[4] Y. Amir and J. Stanton. The spread wide area group communication system. Technical Report CNDS-98-4, Johns Hopkins University Center for Networking and Distributed Systems, 1998.

[5] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ansi sql isolation levels. In *SIGMOD Conference*, pages 1–10, 1995.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[7] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databates. *SIGMOD Rec.*, 28:97–108, June 1999.

[8] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. pages 199–216, New York, NY, USA, 1993. ACM Press/Addison-Wesley Publishing Co.

[9] D. Carney, S. Lee, and S. B. Zdonik. Scalable application-aware data freshening. In *Proceedings of the 19th International Conference on Data Engineering (ICDE)*, pages 481–492, 2003.

[10] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computer Surveys*, 33:427–469, December 2001.

[11] P. Chundi, D. J. Rosenkrantz, and S. S. Ravi. Deferred updates and data placement in distributed databases. In *ICDE*, pages 469–476, 1996.

[12] K. Daudjee and K. Salem. Lazy database replication with freshness guarantees. *Proceedings of ICDE*, 2004.

[13] K. Daudjee and K. Salem. Lazy database replication with ordering guarantees. *Proceedings of the 20th International Conference on Data Engineering (ICDE 2004)*, 30:424–435, 2004.

[14] M. P. Developers. Mysql proxy documentation in launchpad. `https://launchpad.net/mysql-proxy`.

[15] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36:2004, 2003.

[16] S. Elnikety, W. Zwaenepoel, and F. Pedone. Database replication using generalized snapshot isolation. In *SRDS*, pages 73–84. IEEE Computer Society, 2005.

[17] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, page 173, 1996.

[18] R. Guerraoui and A. Schiper. Fault-tolerance by replication in distributed systems. In *Proceedings of the 1996 Ada-Europe International Conference on Reliable Software Technologies*, pages 38–57, London, UK, 1996. Springer-Verlag.

[19] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *Computer*, 30:68–74, April 1997.

[20] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Ithaca, NY, USA, 1994.

[21] A. C. Junior, A. Sousa, E. Cecchet, F. Pedone, J. Pereira, L. Rodrigues, N. M. Carvalho, R. Vilaça, R. Oliveira, S. Bouchenak, and V. Zuikeviciute. State of the art in database replication. Technical report, Universidade do Minho, Jan 2007.

[22] B. Kemme. *Database replication for clusters of workstations*. PhD thesis, Dept. of Computer Science, Swiss Federal Institute of Technology Zurich, Jan 2000.

[23] A. Labrinidis and N. Roussopoulos. Balancing performance and data freshness in web database servers. In *VLDB*, pages 393–404, 2003.

[24] Y. Lin, B. Kemme, M. P. Martínez, and R. J. Peris. Middleware based data replication providing snapshot isolation. In *Proceedings of the 2005 ACM SIGMOD international*

conference on Management of data, SIGMOD '05, pages 419–430, New York, NY, USA, 2005. ACM.

[25] H. Liu, W. K. Ng, and E.-P. Lim. Scheduling queries to improve the freshness of a website. *World Wide Web*, 8:61–90, March 2005.

[26] C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 144–155, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[27] S. Pachev. *Understanding MySQL internals*. O'Reilly, Jan 2007.

[28] E. Pacitti, P. Minet, and E. Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 126–137, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[29] C. L. Pape, S. Gançarski, and P. Valduriez. Data quality management in a database cluster with lazy replication. *JDIM*, 3(2):82–87, 2005.

[30] B. Schwartz, J. D. Zawodny, D. J. Balling, V. Tkachenko, and P. Zaitsev. *High Performance MySQL: Optimization, Backups, Replication, and More; 2nd ed.* O'Reilly, 2008.

[31] D. Serrano, M. Patino-Martinez, R. J. Peris, and B. Kemme. Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation. In *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, pages 290–297, Washington, DC, USA, 2007. IEEE Computer Society.

[32] S. Shah, K. Ramamritham, and P. Shenoy. Resilient and coherence preserving dissemination of dynamic data using cooperating peers. *IEEE Transactions on Knowledge and Data Engineering*, 16:799–812, July 2004.

[33] R. M. Vilaça, J. Pereira, R. Oliveira, J. E. Iñigo, and J. R. de Mendívil. On the cost of database clusters reconfiguration. In *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems*, pages 259–267, Washington, DC, USA, 2009. IEEE Computer Society.

[34] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. *Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, page 464, 2000.

[35] M. Wiesmann, A. Schiper, F. Pedone, K. Bettina, and G. Alonso. Database replication techniques: A three parameter classification. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*, pages 206–, Washington, DC, USA, 2000. IEEE Computer Society.

Appendix A

Additional Results

A.1 MySQL Replication

A.1.1 Master and Multiples Slaves

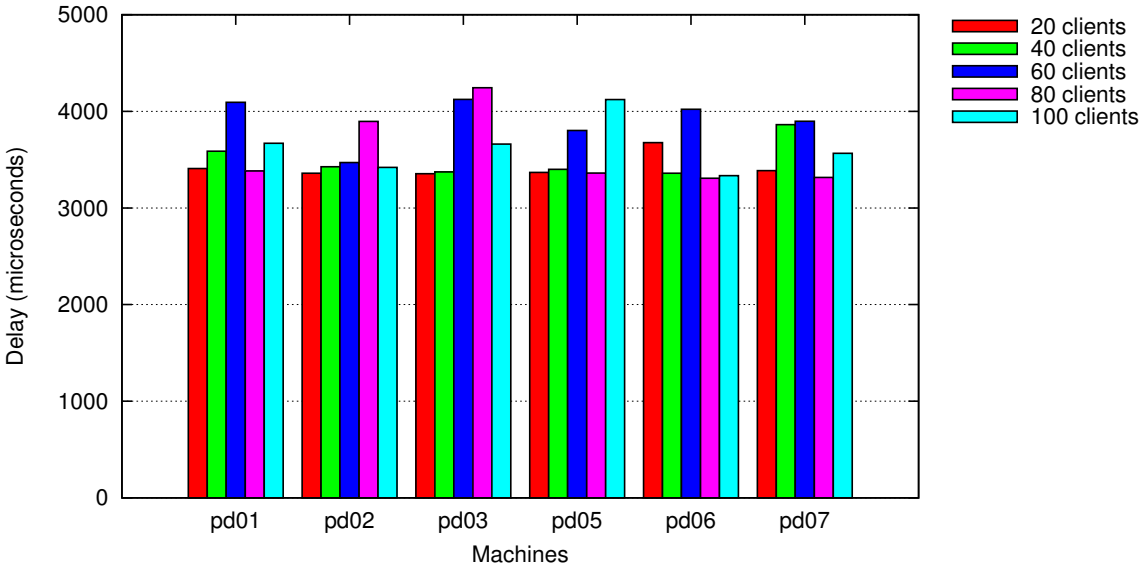

Figure A.1: Replication delay values for Master and Multiple Slaves topology (no think-time)

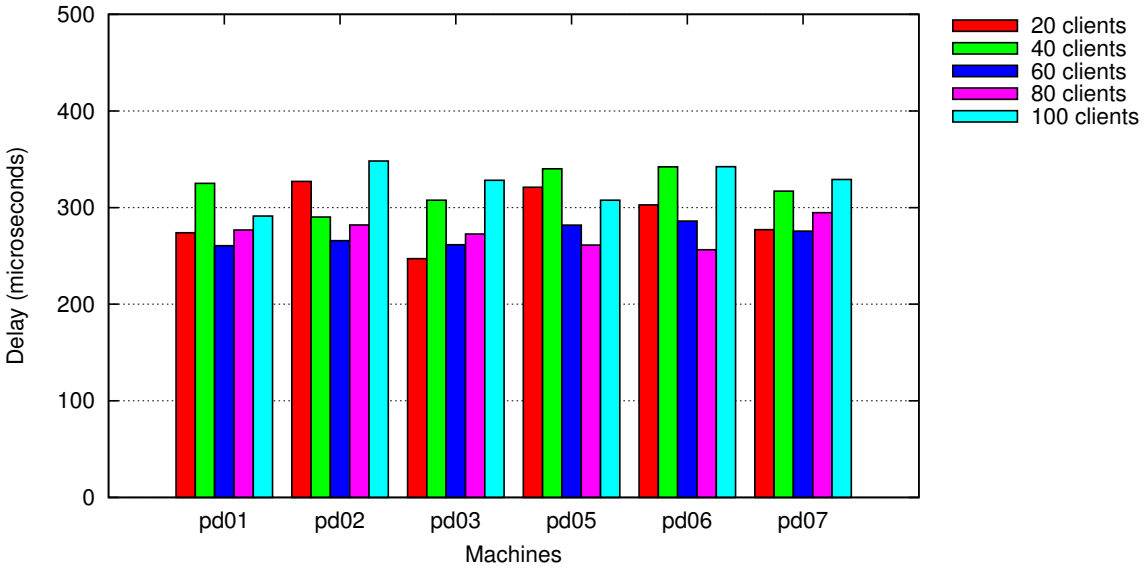

Figure A.2: Replication delay values for Master and Multiple Slaves topology (one-third of think-time)

A.1.2 Chain

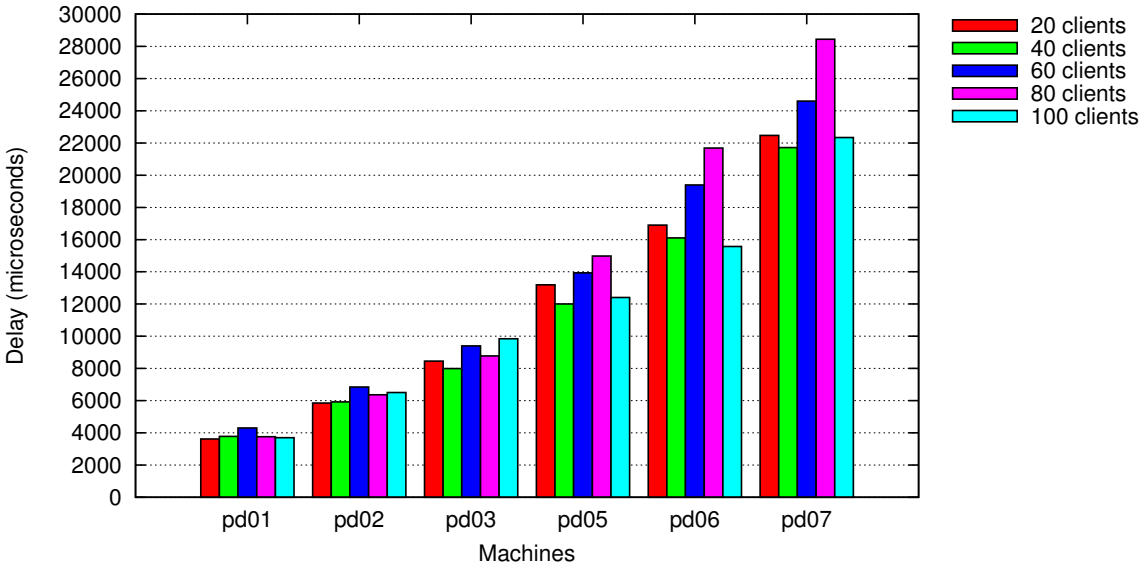

Figure A.3: Replication delay values for Chain topology (no think-time)

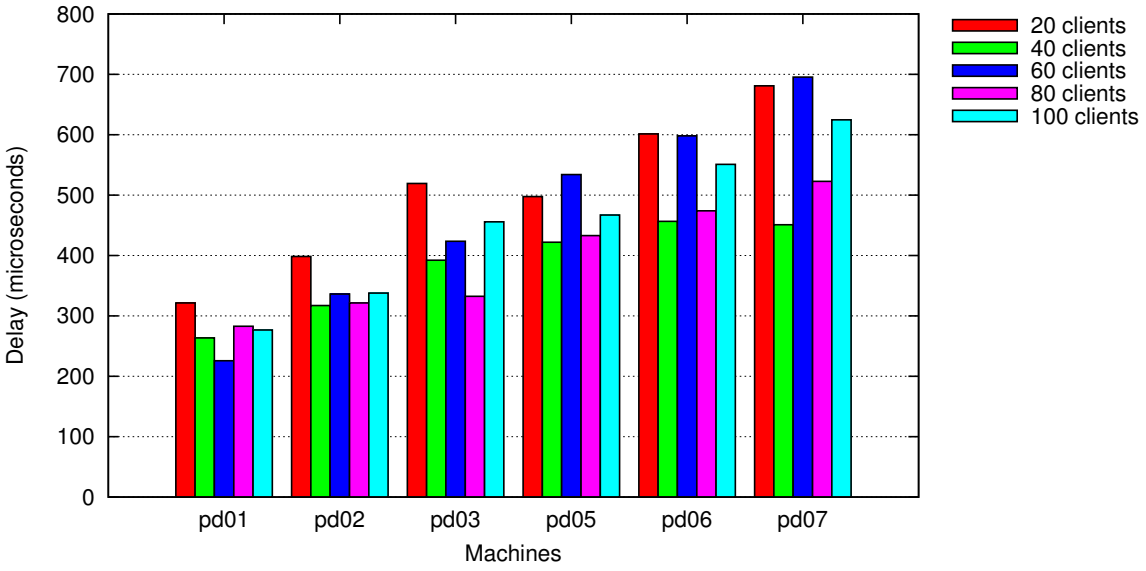

Figure A.4: Replication delay values for Chain topology (one-third of think-time)

A.2 Proxy Spread Plugins - Active Replication

A.2.1 FIFO Messages

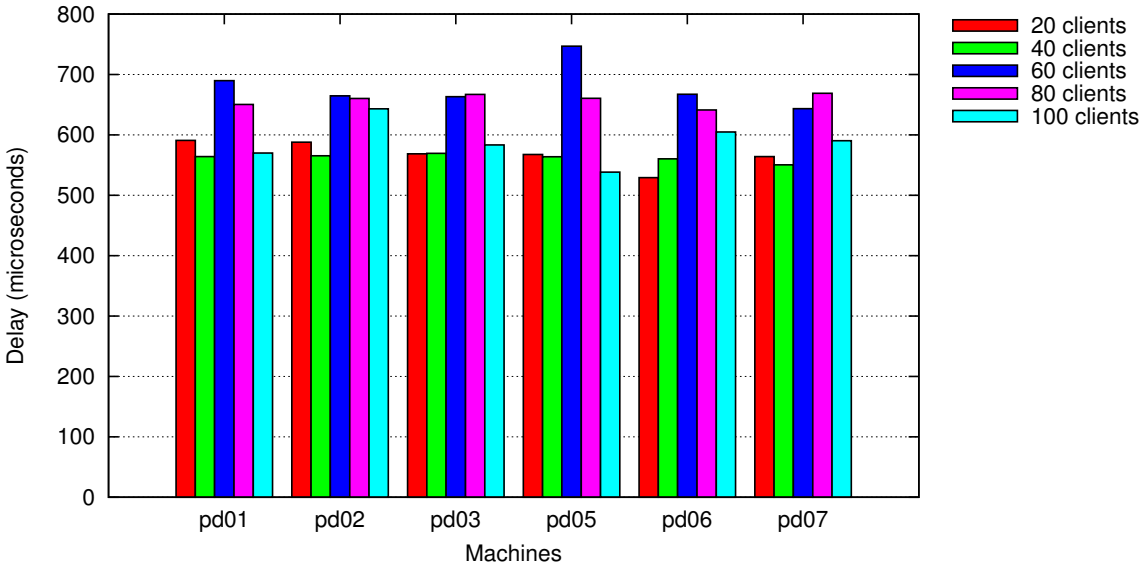

Figure A.5: Replication delay values for active replication with Proxy Spread plugins with FIFO messages (one-third of think-time)

Varying number of replicas

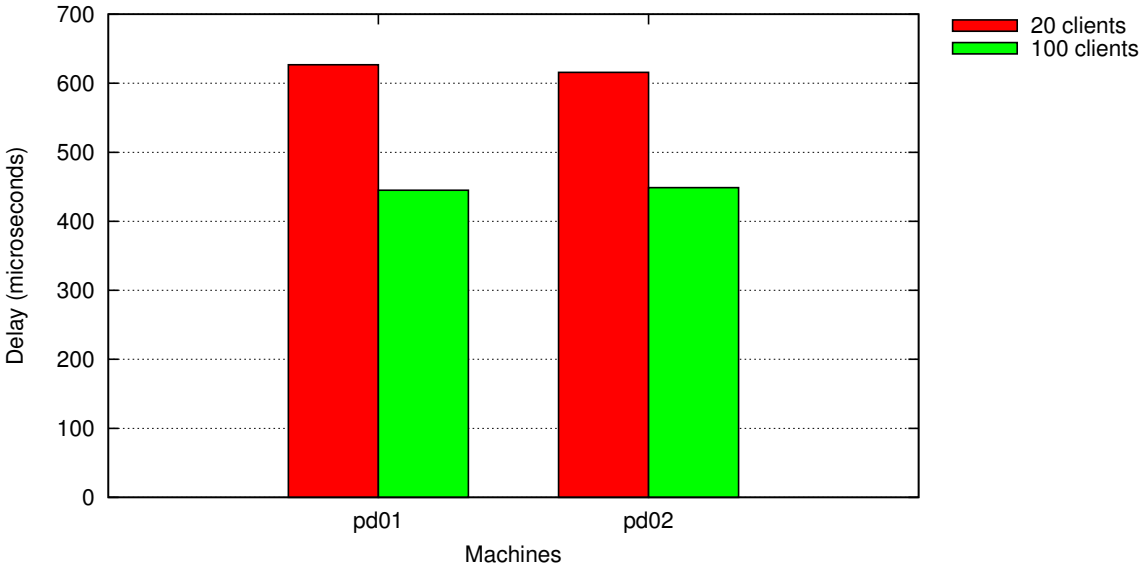

Figure A.6: Replication delay values for active replication with Proxy Spread plugins with FIFO messages (one-third of think-time, two replicas)

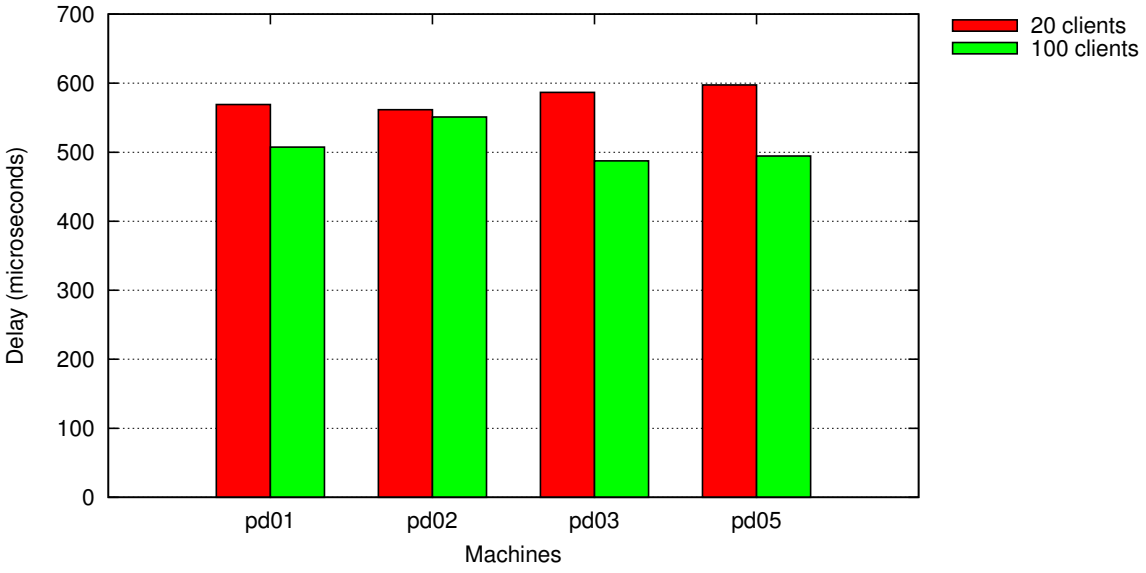

Figure A.7: Replication delay values for active replication with Proxy Spread plugins with FIFO messages (one-third of think-time, four replicas)

A.2.2 AGREED Messages

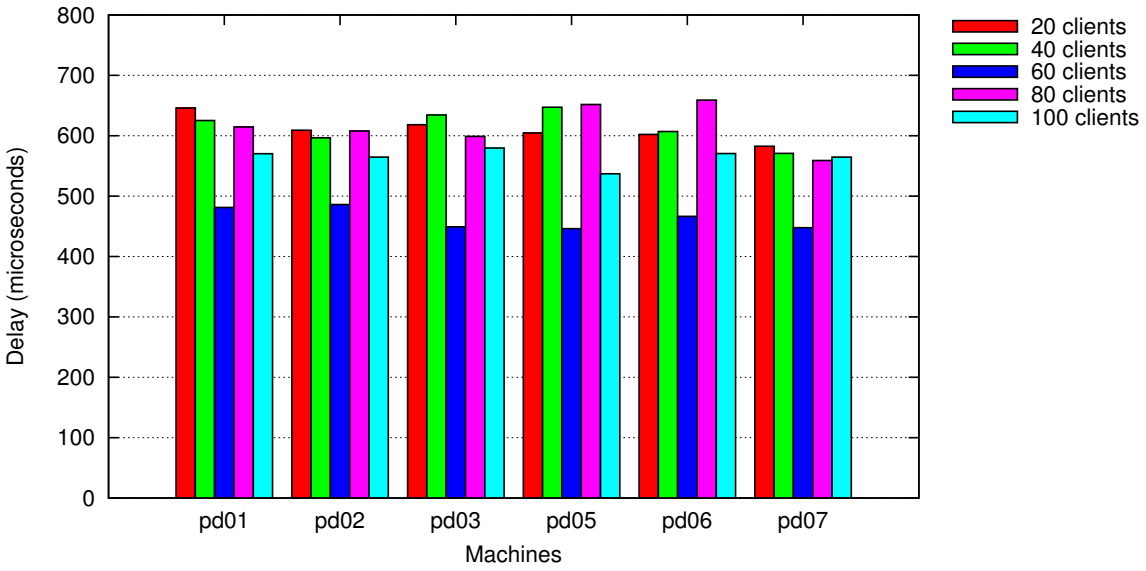

Figure A.8: Replication delay values for active replication with Proxy Spread plugins with AGREED messages (one-third of think-time)

Varying number of replicas

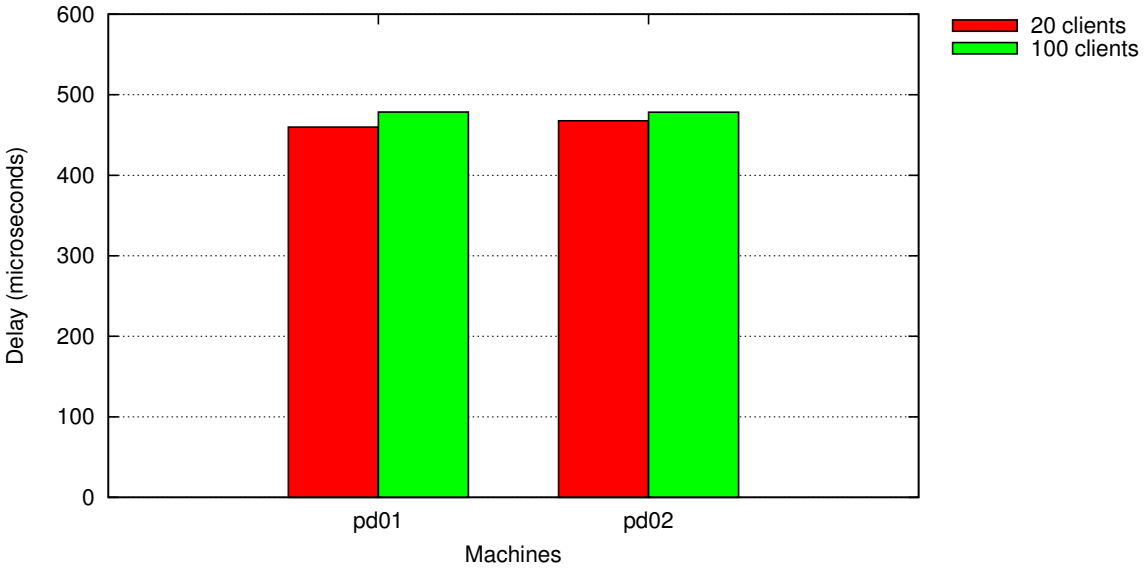

Figure A.9: Replication delay values for active replication with Proxy Spread plugins with AGREED messages (one-third of think-time, two replicas)

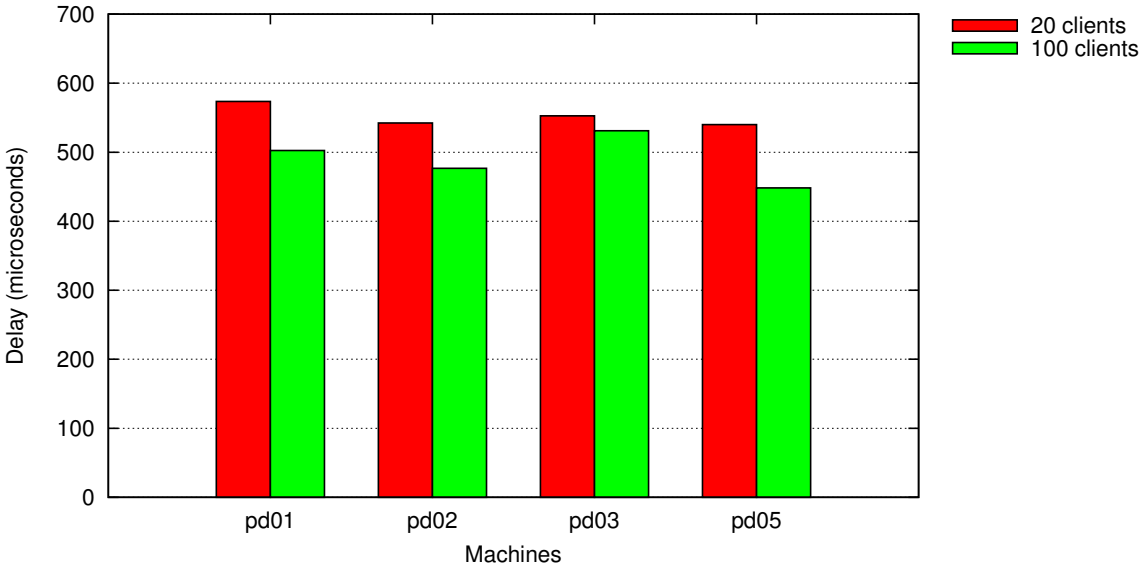

Figure A.10: Replication delay values for active replication with Proxy Spread plugins with AGREED messages (one-third of think-time, four replicas)

Appendix B

Code and Scripts

B.1 Lua Script to use on Proxy Spread Master Plugin

```
mysql-proxy --plugins=proxyspread_master --proxyspread-lua-script=spreadrepl.lua
```

spreadrepl.lua

```
1 function read_query(packet)
2
3   query = string.sub(packet, 2)
4
5   --print("Seen the query: " .. query)
6
7   spread_send_msg(query)
8
9 end
```