Evaluating performance of a fault-tolerant system that implements replication and load balancing

En utvärdering av prestandan hos ett feltolerant system som implementerar replikering och lastbalansering

Rickard Hellenberg
Oskar Gustafsson

Supervisor: Simin Nadjm-Tehrani
Examiner: Nahid Shahmehri
Students in the 5 year Information Technology program complete a semester-long software development project during their sixth semester (third year). The project is completed in mid-sized groups, and the students implement a mobile application intended to be used in a multi-actor setting, currently a search and rescue scenario. In parallel they study several topics relevant to the technical and ethical considerations in the project. The project culminates by demonstrating a working product and a written report documenting the results of the practical development process including requirements elicitation. During the final stage of the semester, students create small groups and specialise in one topic, resulting in a bachelor thesis. The current report represents the results obtained during this specialisation work. Hence, the thesis should be viewed as part of a larger body of work required to pass the semester, including the conditions and requirements for a bachelor thesis.
Companies and organizations increasingly depend on their computer systems to help them in their work. This means that the availability of these computer systems becomes even more important as organizations are increasingly dependent on it to function. Therefore, fault tolerance needs to be considered when designing a computer system. However, when implementing fault tolerance to increase the availability it may affect the performance of the system. This thesis describes an implementation of a system that provides fault tolerance against fail-stop faults and analyzes the performance. The system consist of a primary server and a backup server and each has a GO web server and a MySQL database installed. MySQL has a built-in functionality for replication that is used to replicate the data from the primary to the replica. Two different approaches for replication are used and compared in this thesis. The system also has a load balancing server with a program called HAProxy installed. The program is used to switch between servers in case of a failure and enables load balancing between the servers, although this setup only allows for read requests to be sent to the backup server. The measurements of the implemented system shows that enabling load balancing for read requests has little effect on lower the response time when the system is under low load. For 25 users the response time was just 5 ms faster when enabling load balancing. For 50 users however, the response time was 33 ms faster when enabling load balancing. The system was evaluated using measurements of the response time and the percentage of stale data under different network loads and different requests from the system. Two different methods of replication in MySQL: asynchronous and semisynchronous were tested to see how they affect the response time and the consistency of the system. The measurements show asynchronous replication has a lower response time, but semisynchronous replication has less stale data. This means that choosing between asynchronous or semisynchronous replication is a trade-off between choosing lower response time or choosing less stale data and less risk of losing data.
Acknowledgments

We would like to thank our supervisor Simin Nadjm-Tehrani for her support and feedback during the process of this thesis. We would also like to thank Mikael Asplund for his feedback during the initial state of the process. We would also like to thank Fredrik Håkansson and Dennis Dufbäck for their opposition on this thesis. Finally, we would like to give a special thanks to our course mates in the parallel project during the sixth semester of our program. Anton Silfver, Markus Johansson, Per Gustavsson, Sebastian Andersson and Martin Larsson were great to work with and as a group we constructed the original system which laid ground for this thesis.
## Contents

**Abstract** iii  
**Acknowledgments** iv  
**Contents** v  
**List of Figures** vi  
**List of Tables** vii  

1 **Introduction** 1  
1.1 Purpose 2  
1.2 Problem Formulation 2  
1.3 Method 2  
1.4 Limitations 3  
1.5 Outline 3  

2 **Fundamental concepts and related works** 4  
2.1 Availability 4  
2.2 Faults and fault tolerance 4  
2.3 ACID 4  
2.4 InnoDB Checkpointing 5  
2.5 Replication 5  
2.6 Server Load Balancing 6  
2.7 DNS failover and load balancing 8  
2.8 Related Works 8  

3 **System description and experimental setup** 10  
3.1 Architecture of the system 10  
3.2 Experimental setup 12  

4 **Results** 15  
4.1 Server performance for different loads 15  
4.2 The impact of load balancing 17  
4.3 Comparing different replication styles 18  
4.4 Failover time after a fail-stop-failure 19  

5 **Discussion, Conclusion and Future work** 21  
5.1 Discussion 21  
5.2 Conclusion 23  
5.3 Future Work 24  

**Bibliography** 25
List of Figures

1.1 Architecture of the original system ..................................... 1
2.1 Description of MySQL's master/slave replication .......................... 6
2.2 Asynchronous replication in MySQL ........................................ 6
2.3 Semisynchronous replication in MySQL ................................. 6
2.4 HAProxy health checks ....................................................... 7
3.1 Logical architecture of the fault-tolerant system .......................... 11
4.1 Response time of requests with 10 users continuously sending requests ...... 15
4.2 Response time of requests with 100 users continuously sending requests .... 16
4.3 Average response time for experiments with different number of users continuously sending write requests .......................... 16
4.4 Average response time for experiments with different number of users continuously sending read requests .......................... 17
4.5 Response time of requests using asynchronous replication .................. 18
4.6 Response time of requests using semisynchronous replication ................ 19
4.7 Response time of requests being made during a fail-stop scenario .............. 20
4.8 Response time of requests being made during a fail-stop with the original system 20
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Average response time and standard deviation for different loads of write requests</td>
<td>16</td>
</tr>
<tr>
<td>4.2</td>
<td>Average response time and standard deviation for different loads of read requests</td>
<td>17</td>
</tr>
<tr>
<td>4.3</td>
<td>Response time when sending 100% read requests not using load balancing</td>
<td>18</td>
</tr>
<tr>
<td>4.4</td>
<td>Response time when sending 100% read requests using load balancing</td>
<td>18</td>
</tr>
<tr>
<td>4.5</td>
<td>Response time when sending 50% (48% for tests with 25 users) read requests not using load balancing</td>
<td>18</td>
</tr>
<tr>
<td>4.6</td>
<td>Response time when sending 50% (48% for tests with 25 users) read requests using load balancing</td>
<td>18</td>
</tr>
<tr>
<td>4.7</td>
<td>Measurements regarding semisynchronous and asynchronous replication</td>
<td>19</td>
</tr>
</tbody>
</table>
1 Introduction

Having computer systems with high availability is crucial for companies and organizations. Every minute that a server is not available can cost huge amounts of money for a company in lost revenue and bad publicity. For systems used in the healthcare it can be worse, it can have a negative impact on people’s health. Therefore, availability is a significant aspect to consider when designing computer systems.

This thesis aims to improve the availability of an emergency communication system developed by a group of students at Linköping University that could be used by ambulance personnel in case of catastrophic events. Its low-tech architecture can be seen in Figure 1.1 and the server has a web server running Go and a MySQL database. Since the mock-up system could be used in various life threatening situations it is therefore crucial that the system is available at all times. To achieve high availability it is essential to have some form of redundancy in case of failures, which in turn may increase the complexity of the system and may cause new problems in the system. One way to achieve higher availability is to have fault tolerance, which means that the system can continue operating properly despite a fault. This can be achieved by having a backup server that can take over if the primary server should crash. It is essential that data that has been acknowledged to clients is not lost when the backup server takes over. Another important aspect to consider is the required performance of the system.

A problem is that when designing systems that can tolerate faults without failing, these two aspects may get into conflict. However, by allowing clients to access data from more than a single server, redundancy can improve performance as well but adds a risk of clients fetching old data from the backup server that has been updated in the primary server but yet not reached the backup server. This is referred to as stale data.

![Figure 1.1: Architecture of the original system](image-url)
To increase the availability of the system this thesis aims to implement a replicated backup server that can take over if the original server crashes. This is called a fail-stop failure and means that the system stops operating.

Different techniques and approaches are presented that is considered when designing the system. For some design choices experiments are needed to settle upon the best alternative. One design choice is whether or not to allow read requests from the backup server. This is investigated by measuring its advantages and disadvantages during different network loads. Another design choice that is settled by using experiments is the choice between using an approach called semisynchronous replication or asynchronous replication.

1.1 Purpose

The main purpose of this thesis is to present a way of implementing replication in order to improve the tolerance against fail-stop-faults at a server. It also aims to analyze the performance of different approaches for implementing fault tolerance.

1.2 Problem Formulation

This thesis focuses on implementing a system that can tolerate fail-stop-failures of the primary server and to investigate how the implementation affects the performance. To achieve this goal the problem has been broken down into the following subproblems.

- Analyze the response time of the original system under different loads.
- Investigate the impact load balancing has on the average response time for different proportions of read and write requests.
- Investigate the effect on the response time and the amount of stale data when using semisynchronous replication instead of asynchronous replication.
- Evaluate the failover time of the implemented fault-tolerant system.

1.3 Method

Different network loads are simulated and sent to the original system to see how the response time is affected, where the response time is defined as the time between sending a request and when the response is received. These measurements are needed to compare the original system to the more fault-tolerant alternative. Further on two of these network loads are chosen to be used in the remaining experiments.

Different techniques and approaches for implementing fault tolerance is investigated and compared before deciding upon an option. In some cases the alternatives are compared using experiments. One case is the choice between semisynchronous and asynchronous replication that is compared through experiments in regards of response time and the amount of stale data. To analyze the amount of stale data in the database write requests are sent to the primary database and is thereafter read from the same field in the backup database. If the read-value matches the write-value it is considered as data being up to date, otherwise it is considered as stale data. The occurrences of stale data are added together and the percentage of stale data is calculated.

To analyze the impact of allowing read requests to be sent to the backup server the response time is first measured when read requests to the backup server are allowed and then when they are not allowed. The measurements will be made under different proportions between read- and write requests and the data from the measurements analyzed to see how the different proportions affect the response time.
To show that the final system provides higher availability than the original system a crash of the primary server is simulated and the recovery time is measured. The failover time is defined as the time between a failed packet and the time it takes for the next packet to be successfully transmitted to the backup server.

All experiments are made using a program called Apache JMeter which can simulate several users by creating requests in several different threads.

1.4 Limitations

To implement a fault-tolerant system and analyze it in a Bachelors thesis the subject has been narrowed down and been focused on specific aspects of the system. Therefore these limitations were made:

- The implemented fault-tolerant server is limited to handle a single fail-stop failure of the primary server.
- After a fail-stop at a server the system has to be manually reset and no ways to restore the system to its original state is analyzed.

1.5 Outline

Chapter 2 presents fundamental concepts and background information about the techniques later referred to in the thesis. Chapter 3 describes the system that was implemented as well as the method used to measure the performance. Further on in chapter 4 the results of the measurements are presented. Finally, in chapter 5 the results are discussed and a conclusion is made. Chapter 5 also presents future work that can improve or build upon the work of this thesis.
This chapter describes concepts and techniques referred to and used in the rest of the thesis. It also presents selected previous work related to the work of this thesis.

2.1 Availability

Availability is a term describing a system’s readiness for correct service. It is measured by comparing the time the service is ready to be used and the time that the correct service is not ready to be used. The system’s uptime during a longer period of measurement is referred to as MTTF (Mean Time To Failure) and the time it takes for the system to be available again as MTTR (Mean Time To Recover). The availability can out of these terms be described by equation 2.1 which gives us an percentage value for systems being ready for use.

\[
\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}
\]  

2.2 Faults and fault tolerance

A fault is the cause of an error that may lead to a failure, which means that the specified service cannot be fulfilled. There may be several different kinds of faults that can cause a failure in a system. Faults in a system can affect the dependability and especially availability negatively. To be able to improve the dependability of a system there are three general means: fault prevention, fault tolerance and fault removal. This thesis focuses on fault tolerance.

Fault tolerance aims to avoid failure after an error has occurred in a system. The system can be able to handle a fault without reaching a state where there is a failure. There are several ways of implementing fault tolerance which include error detection, error handling and fault handling. To achieve a fault-tolerant system redundancy is necessary.

2.3 ACID

ACID stands for (atomicity, consistency, isolation and durability), and is a set of properties regarding database transactions, where each property describes:
• Atomicity implies that an operation must be performed entirely or not performed at all. Transactions can have several operations that acts as one atomic operation.

• Consistency is preserved if a transaction that is completed from beginning to end without interference from other transactions, takes the database from one consistent state to another.

• Isolation hides information within transactions from other concurrent transactions. This makes sure that the end result for interleaving transactions are the same as if they were made sequentially.

• The durability property makes sure that when changes in a database have been committed it will be saved and persist even if there is a failure.

2.4 InnoDB Checkpointing

InnoDB is the database storage engine used by default in MySQL. It uses checkpointing to create a known point in a log from where it can start applying changes after a crash. Instead of writing directly to disk after every change InnoDB saves modifications to a memory, when these modifications are flushed to disk a checkpoint is created. This is used to increase performance since writing to disk takes longer time than saving to a local memory.

Fuzzy checkpointing is a mechanism to decrease the time a checkpoint reduces the performance of the databases main thread. Fuzzy checkpointing flushes smaller batches of modifications over time instead of flushing everything at once, meaning that the decrease in throughput will be spread out more evenly.[3]

2.5 Replication

Replication means that information or computation ability is distributed at several locations and can be used to improve reliability, fault tolerance and availability. A fault tolerant system based upon replication should respond despite failures and clients should not be able to tell the difference between the services they receive from different replicas. Implementing this behaviour is not trivial and there are several issues that need to be considered. There are therefore many different ways of implementing replication, each with different advantages and disadvantages.

2.5.1 Active & Passive replication

Two common replication techniques are active replication and passive replication. In active replication client operations are directly sent to all replicas. A problem with this approach is that if a message does not reach all replicas the data could be inconsistent. In Passive replication, also known as primary-backup replication, one of the replicas is designated as primary. The primary replica then executes the clients’ operation and passes updates to the other replicas. Passive replication can be described by performing five steps. First there is a request which is sent from a client to the primary replica. Next step is coordination which means that the primary replica handles requests in the order they are received. Third, the primary replica executes the request. Forth, if there was an update request the primary replica sends the update to all other replicas which reply with and acknowledgement. Last step is a response from the primary replica to the client.[4]

2.5.2 Replication in MySQL

MySQL uses primary-backup replication and refers to it as master-slave replication. In Figure 2.1 we see that the master database and slave database uses different logs. The master
database saves all its actions in a transaction log and also saves update transactions to a log called binary log or binlog. The slave has an I/O thread that fetches data from the binary log and puts it into its relay log. Then the slave can read from the relay log and execute all the updates which has been made by the master. In this way the slave database can stay up to date with the master except from the delay from the time it takes to transfer the binary log to the relay log and execute the transaction. During this delay clients may receive stale data when reading from the backup server.

The usual passive replication has synchronous replication, meaning that before acknowledging the request the server needs to make sure that the replicas are updated. MySQL instead uses two different approaches, asynchronous and semisynchronous replication, which are illustrated in Figure 2.2 and 2.3. In asynchronous replication the primary database commits the updates which makes the changes permanent without knowing that the update is received at the replicas. In semisynchronous replication the primary waits for acknowledgement from the replicas before committing and replying the client. By waiting for an acknowledgement that the slave has received the update one gets a guarantee that there has been no data loss when sending the updates to the slave. This does not however mean that semisynchronous replication can provide a guarantee for no data being lost at the replicas, since the replica only acknowledges that the update has been received and not that the update has been committed. This means that data can be lost if there is a failure before the update is committed.

Server load balancing allows client requests to be spread out on several different servers. By having a pool of available servers a load balancing layer can distribute traffic between the
servers to improve overall performance. Since there are several servers to choose from, faulty servers can be removed from the pool when a fault is detected, thus improving availability as well.

2.6.1 HAProxy

A proxy can be used to make a primary-backup fault tolerance mechanisms transparent to the client. By forwarding the traffic to the primary server the client need not be aware of which server is the primary.

HAProxy stands for High Availability Proxy and is an open source solution for load balancing used by a lot of big companies such as Twitter, Alibaba and Instagram. It works with TCP and HTTP-based applications. It has a fast I/O layer combined with a priority-based scheduler. and has a lot of different features, such as the following:

- Proxying: The action of transferring data between a client and a server.
- Monitoring: The status of the servers are regularly monitored and can be done in various ways.
- Providing high availability: Only servers with a valid status are available in the pool of servers, thus being ready for use by the client.

HAProxy uses health checks (also called a ping) to check the status of the servers. This means that the proxy sends a request to the server and can classify the server as up or down depending on whether or not it receives a response. How these health checks are used can be seen in Figure 2.4a)

Figure 2.4: HAProxy health checks

(a) HAProxy health checks scenario with the primary server running

(b) HAProxy health checks scenario where the primary server is unavailable

Figure 2.4b) describes the concept of a health check made by the proxy. In this scenario the server is up and running and the figure has two variables $T_{RTT}$ and $T_{ping}$. $T_{RTT}$ is the time from where a health check is sent to the server until it receives a response and $T_{ping}$ is the interval between the health checks. In Figure 2.4b) another variable $T_{Timeout}$ is used, this describes the amount of time HAProxy waits until it suspects there is a failure at the server. So if there is no answer at all or if $T_{RTT} > T_{Timeout}$ our proxy will assume the server is not working. When reached one $T_{Timeout}$ the proxy will send another health check and if there still...
is no response it will no longer forward traffic to this server until the server starts responding again after a fixed number of health checks.

When HAProxy distributes packets to different servers, it can use different kinds of scheduling algorithms. HAProxy supports an algorithm called round-robin which means that each server is used in turns. This is considered to be the smoothest and fairest algorithm when the processing time of the servers are evenly distributed. Another algorithm that can be used is called leastconn, which means that the packets are forwarded to the server with the least amount of connections open. It is recommended to be used when connections are expected to be long but is not recommended to be used when dealing with protocols like HTTP that uses short sessions\[7\].

2.6.2 Routing different requests to different servers

To be able to redirect traffic differently depending on what type of request it is two different approaches are presented. One approach is to choose where to redirect traffic depending on the HTTP-header of the packet. A problem arises when the HTTP-packets are encrypted using SSL or TLS. Then you need to decrypt the packet before being able to look at the HTTP-header. This technique is called SSL- or TLS-termination, since you terminate the SSL- or TLS-connection. However, if there is no trusted network after the proxy there may also be a need for re-encrypting traffic before forwarding it. Having to decrypt and re-encrypt traffic clearly leads to longer processing-time and lower performance.

Another approach is instead to use SSL- or TLS-passthrough. This means that the load balancer works in the transport layer and lets the TLS-connection pass through. As mentioned before, this means that there is no way to look at the HTTP-header before forwarding traffic. To be able to redirect traffic differently we need another approach. One way is to send packets to different ports depending on what type of request it is. This however, has the negative impact that the client needs to be aware of which ports are used for different requests. Also, if there are many different rules for forwarding it is not recommended to have an open port for each separate case.

2.7 DNS failover and load balancing

The Domain Name System is used as a “phonebook” that translates human friendly web adresses to IP-addresses. A DNS-lookup for a single web-address can give a list of several IP-addresses in an order that often varies in a round-robin fashion but can use other algorithms as well. The client will then try the first one and in case it does not respond, try again with the next one. This means that the DNS-server can be used to improve the availability in case of failures and to balance the load of the servers.

The DNS-servers uses no checks to see if the servers are in a correct state, which means that a client can be directed to a server that is not available. To address this issue a DNS-change can be triggered from another server which may keep the DNS-server from giving addresses to servers that are not available. A drawback with this solution is that the updates can be delayed due to DNS-servers using caches of the addresses. This may be reduced by setting a lower value of the “Time to live” property lower which makes the DNS-servers cache it for a shorter while. This however, has the disadvantage that it may decrease performance. Also, different internet service providers have different rules for caching data which can cause longer delays for some clients. One advantage with using DNS to handle failovers is that it is easy to implement without changing much of the architecture of the system\[8\].

2.8 Related Works

Difficulties in optimizing the performance when introducing redundancy into distributed systems is a well known problem. For example Ladin et al.\[9\] consider the cost in perfor-
2.8. Related Works

mance to keep high consistency in a distributed system. They describe a new replication method which supports different ways to order updates. This is used in order to give a developer more choices when designing a highly available system with a requirement of high consistency as well.

Several other papers investigate different methods of replication to keep the cost of performance as low as possible such as \cite{10,11}. This thesis only focus on analyzing the performance of using MySQL’s replication instead of comparing between different approaches.

An alternative to using replication for fault tolerance is to use a container based approach, which is described in \cite{12}. The container based approach allows for having different servers providing the same service, which gives the service fault tolerance.

In large distributed computer systems there is a need for load balancing between servers in order to not put all pressure on a single server, which may cause it to overload. Cardellini et al. \cite{13} introduces and compares several ways to implement a load balancing mechanism. In contrast to our thesis it uses simulations to compare the performance of the different techniques.

In this thesis only a certain algorithm is used for load balancing due to limitations. Although, the choice of algorithm to use when load balancing could make a significant difference in performance of the system. Sharma et al. \cite{14} analyzes the performance of several different load balancing algorithms and compares them to each other in different situations.
This chapter presents the architecture of the implemented system in section 3.1. The design choices are presented with a motivation why the choice was made. In section 3.2, the software that was used for the experiments is presented along with the setup for each experiment.

3.1 Architecture of the system

To design a fault-tolerant architecture it is essential to have some form of redundancy. The redundancy in this design is a replica of the original server. This means installing the same web server and synchronizing the data between the databases. This leads to having to decide how to implement the replication and how to switch between the servers.

The technique chosen for switching between servers is to use a proxy. The biggest reason is that the proxy can recognize if there is a problem with a server in seconds and then start the failover process immediately afterwards. Compared to DNS-changes where the IP-addresses are cached this is a major advantage. Another reason is that it allows for read requests to be sent to the backup server, which may improve performance. Despite using a proxy, there is still a single point of failure in the system. This could be resolved by having another load balancer, and in that case we could use a DNS-change to failover in case the load balancer crashes. Even though the proxy is a single point of failure, one could argue that a proxy is more stable than a web server, thus improving the availability.

As seen in Figure 3.1, there is now a proxy that primarily sends traffic to the primary server, using the primary channel. This proxy server is located physically in London, since it could be provided for free from a company with servers placed there and Linköping University only provided two servers. This has the consequence that when sending a ping to the proxy it has a response time around 60 ms higher than for the other servers. The backup channel is used in case of a failure of the primary server. It may also be used for read requests that are allowed to read from the backup server. The users are simulated using a software called JMeter which is described under section 3.2.1 that is able to send HTTP requests. The internal channel for replication is used for replication between the servers.
3.1. Architecture of the system

3.1.1 Replication of the server

Instead of developing a functionality for replication it was decided to use the built-in function in MySQL. Whether to choose the asynchronous or semisynchronous approach remains to be investigated in chapter 4 and 5.

In this system there are the two servers with their own databases, one working as a primary and one working as a backup. In Figure 3.1 we see the internal channel between the two servers. This channel will be used for the backup to be able to update the database equal to the primary. This channel is directed one way only, meaning that when using master slave replication all database updates must go through the primary server to keep the data consistent in the databases. The reason for choosing master-slave replication was that we wanted to fulfill the ACID properties. A different approach called master-master replication could have been used to be able to load balance write requests as well. This means that two different operations could update the same data item simultaneously on different databases which would potentially violate the consistency of the databases.

3.1.2 Server functionality

The web servers handle two different HTTP requests named read and write. Given in the HTTP body of a write request there is an id and a value. These are used to update the database at a row with the given id and inserts the value. If the update was made successfully then a OK statement will be returned and in case of an error it will return an error message. The read request will instead take an id as input and select a value from a table in the database with that id and return the value as output. As in the write header this will return an error message if there occurred any error. With these two headers we can distribute the requests in any way we want with the help of the proxy.

3.1.3 Configuration of HAProxy

Instead of developing new software for load balancing it was decided to use an already developed software since there are several free open source alternatives available. By using a trusted load balancing software the risk of software-failures at the proxy is limited. Ultimately it was decided to use a software called HAProxy on the proxy server. The reason for choosing HAProxy is that it is considered to be reliable and fast and is the de-facto standard free open source load balancer. Another contributing factor is because of the widespread information about how to configure the software.

To be able to route read- and write requests differently we chose to use TLS-passthrough and look at what type of port the request was sent to. The reason for that technique was...
chosen was that the load balancer redirects traffic through an insecure network and therefore would need to re-encrypt the traffic if using TLS-termination and look at the HTTP-headers, which would lead to a drop in performance. In this configuration the port 8080 is used when requests are allowed to go to both servers and port 80 is used for requests that only a forwarded to the primary server.

The read requests are forwarded to the primary and backup server using the algorithm round-robin. One of the reasons for choosing the algorithm is that it is easy to understand and by using this algorithm it is easier to analyze the results when measuring the response time for different scenarios. Another reason is that round-robin is preferred over leastconn for shorter sessions which is the case when using HTTP. The configuration uses the default value for the time interval between health checks which is 2 seconds and the default value of 3 unsuccessful health checks before confirming that the server is down.

3.2 Experimental setup

To evaluate the implemented system and to compare the different approaches several experiments are performed. To lower the margin of errors, each measurement has been executed several times. For each measurement there are 1000 requests which allows calculating the value of the result for average, minimum and maximum response time. When executing every measurement several times we can calculate the standard deviation for each experiment. The standard deviation shows how much variation there is between each measurement and can indicate how likely it is that the experiment would give the same results when recreated.

3.2.1 Apache JMeter

Apache JMeter is a software that can be used to load test functional behavior and measure performance. The fact that it is open source and free to use was a contributing factor for choosing this software. Another factor was that it was easy to access information about how to use the software. The software can simulate many simultaneous users and use conditional operators to make complex tests for various situations. A project in JMeter is called a test plan and is used in this thesis to describe the setup used in JMeter. A test plan has a component called thread group that controls how many threads that are being run and how many times. The number of threads controls how many users that is simulated and the value of the loop count controls how many times the thread will send a request. This thesis uses the component HTTP request to send requests and the component “View Result Tree” to collect data from the HTTP requests. It also uses the logical component “If Controller” to send a new requests if the previous request was successful.

In the result tree shown after a test plan is executed there are several fields which are interesting. In particular, we are looking at the load time of the HTTP requests which includes the total time from when the request was sent until the response arrives. This includes another time variable called connect time. The connect time is the time it takes for the simulated user to make a TCP handshake with the server. Using this tool, a simulated user does not always establish a new TCP handshake for each request. Since not all request have a connect time this amount of time is subtracted from the load time. This time is what we refer to as the response time and now only includes round trip time to and back from the server as well as the time it takes for the server to process the request.

Using thread groups with a loop count greater than one implies that a simulated user sends a request and another one as soon as a response for the first one is received. This means that a measurement which simulates 25 users constantly will have 25 ongoing requests, but it does not say anything about how often they send another request. Therefore after each experiment a request per second value is calculated to give the reader a feeling of how many requests are being handled over the period of the experiment. This value together with the number of users tells what amount load the server handles.
3.2. Experimental setup

3.2.2 Setup for creating different loads

To simulate different loads a test plan with one thread group, one HTTP request sampler and a result tree was created. The thread group has two parameters of interest for this experiment, the number of threads and the loop count. The two variables has been changed in such a way that there is always a total of 1000 requests for each experiment. To see how the response time is affected by different loads, 7 different values of the load is measured, with each load measured 5 times. The two other components used in JMeter, the HTTP request and result tree had the same configurations for all loads. The HTTP request was set to go directly to our primary server since we wanted to test the original system. This HTTP request was a write request with a randomized id between 1 and 10000 to insert into the database. This randomized id was important since inserting at the same row into the database would make the database write-lock that row and create a queue for the users trying to access it. By randomizing the id it is possible to handle multiple users simultaneously. Finally, the result tree was there to save the interesting fields of the requests header and response into an XML-file which we could use to analyze the result.

3.2.3 Setup for creating different request mixes

As mentioned earlier the server can handle two different types of requests, read and write. To be able to adjust the proportions of read- and write requests another test plan was created. This test plan contains two different thread groups: one with a HTTP read request and one with a HTTP write request. By adjusting the number of threads in each thread group we can simulate different proportions of read and write requests. For example when sending 50 % reads with 50 users, the thread group for both reads and writes was set to 25 users. To be able to make a comparison without using load balancing the experiment was also performed when the proxy was configured to direct all requests to the primary server.

3.2.4 Setup for comparison of replication styles

To measure response time and stale data proportion two different test plans was used. For the response time the tesplan described in chapter 3.2.2 having 25 users and a loop count of 40 was used.

To check the amount of stale data another test plan was created. This test plan has one thread group with a single thread and a loop count for 1000 times. It has a HTTP request-component that sends a write request with an id of value 1 and a random number between 0 and 10000 to the primary server through the proxy. When the response is received an If-Controller checks if the status is “ok” and if so, it transmits a read request to the secondary server to see if the slave has data which is up to date. The reason for waiting for a status message before transmitting a read request was to confirm that the requested data was updated on the primary server. For each request the value that was written in the write request was compared to the value that was received from the read request. If the values do not match it is considered as a case of stale data. The test plan ran 3 times and the number of occurrences of stale data was added together and the percentage of stale data was calculated.

During these tests the database was configured to either use asynchronous or semisynchronous replication to compare the two. Switching between these two was made by a change in the configuration file in MySQL and then restarting the MySQL service.

3.2.5 Setup to measure failover time

To show that our fault-tolerant system has a higher availability than the original system another test plan was created. Similar to the previous test plans this one had a thread group with 25 users sending write request, but this time the loop count was set to forever and had to be manually stopped. In the middle of each measurement the primary server was shut
down, to see the effect of a failure of the primary server. It was expected that several requests would not be successful. After a while, the responses were expected to be received once again for the fault-tolerant system. In this experiment we were mostly interested in the time when the requests was sent. By looking at the time stamp of the first request which was unsuccessful and the first one which was successful we could calculate a time where there were no successful requests. By doing the measurement 10 times, the MTTR could be calculated.
4 Results

This chapter presents the results from the experiments described in section 3.2.

4.1 Server performance for different loads

The test plan described in section 3.2.2 was executed for seven different numbers of simultaneous users to investigate how our implemented server could handle different loads. Each measurement sent a total of 1000 requests to the server and was made 5 times each to lower the margin of error. Figure 4.1 and 4.2 shows only one of the five samples to be able to follow the behaviour of the system.

Figure 4.1 shows us the behaviour of 10 users making requests. In this figure each dot represents one request. The y-axis shows the response time of the request and the x-axis shows at what time into the test the request was sent. In the figure we can see that there is a stable behaviour except one increasing spike around 10 second into the measurement.

![Figure 4.1: Response time of 10 users continuously sending requests](image)

Figure 4.1: Response time of 10 users continuously sending requests
In Figure 4.2, it can be seen that for a load of 100 users the server no longer shows the same behaviour and seems to be overloaded. This is also noticed on the average response times since it increased a lot as shown in Figure 4.3 and Table 4.1.

Figure 4.2: Response time of 100 users continuously sending requests

Figure 4.3 shows us how the average response time increases by the number of users sending requests. The y-axis describes the average response time for all experiments and the x-axis shows the number of users. At 50 users and higher we can see that the response time increases at a higher rate, as well with the standard deviation shown in Table 4.1 where we can see a more unsteady test results from 75 users and forth.

Figure 4.3: Average response time for experiments with different number of users continuously sending write requests

<table>
<thead>
<tr>
<th>Number of users</th>
<th>5</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg. response time (ms)</td>
<td>43</td>
<td>44</td>
<td>55</td>
<td>102</td>
<td>180</td>
<td>526</td>
<td>1123</td>
</tr>
<tr>
<td>Standard deviation</td>
<td>0.71</td>
<td>1.41</td>
<td>2.13</td>
<td>2.82</td>
<td>10.68</td>
<td>61.91</td>
<td>58.01</td>
</tr>
</tbody>
</table>

Table 4.1: Average response time and standard deviation for different loads of write requests
Comparing the response times when only sending write requests to only sending read requests (see figures 4.3, 4.4 and tables 4.1, 4.2) shows us similar behaviour from the server. The only difference between the two are that the average response time for read requests are slightly lower.

![Figure 4.4: Average response time for experiments with different number of users continuously sending read requests](image)

<table>
<thead>
<tr>
<th>Number of users</th>
<th>5</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg. response time (ms)</td>
<td>10</td>
<td>11</td>
<td>16</td>
<td>49</td>
<td>137</td>
<td>362</td>
<td>897</td>
</tr>
<tr>
<td>Standard deviation</td>
<td>0.05</td>
<td>0.17</td>
<td>2.24</td>
<td>3.6</td>
<td>39.43</td>
<td>25.32</td>
<td>69.57</td>
</tr>
</tbody>
</table>

Table 4.2: Average response time and standard deviation for different loads of read requests

From the results shown in this section we find it most interesting to further study a load between 25 and 50 users since we want to have a scenario as realistic as possible without getting misleading results because of an overloaded server.

### 4.2 The impact of load balancing

When HAProxy was installed a choice regarding whether or not to load balance had to be made. Table 4.3-4.6 show us the average, min, max response times as well as the standard deviation and the average number of request sent per second. The reason for having the average number of request sent per second is because the average response time on its own, does not say enough about the performance. If the requests are spread out over a longer period of time the response time will decrease, but it will not show that the performance of the system is better. Therefore the average request per second together with the average response time tells us what load the server could handle. As suspected using load balancing decreases the response time. The higher load the server is under, the more effect the load balancing has. The standard deviation for these measurements are very low, meaning that recreating these test would probably give the same result.
4.3 Comparing different replication styles

In order to compare asynchronous to semisynchronous replication in terms of response time and stale data two different test plans were used. To compare the different replication methods the same test plan as the one used in section 4.1 was used. Both methods were tested with a load of 25 users.

Figure 4.5 shows us the familiar behaviour from our system when using a load which is not overwhelming. As shown in table 4.7 this experiment gave us an average of 234 ms when using asynchronous replication. Compared to the results described in table 4.1 there is an increase of the average response time. This increase comes from the longer traveling distance by the proxy instead of sending requests directly to the primary server as well as the processing time at the proxy.
When instead enabling semisynchronous replication the server now waits for the acknowledgment from the backup database. This implies a short period of time which the primary server waits which should slightly increase the response time. This is shown in Figure 4.6 where the average response time was 239ms. Compared to asynchronous replication there is an increase of the response time with 5ms.

![Figure 4.6: Response time of requests using semisynchronous replication](image)

In table 4.7 we can see that when comparing the responses for 1000 read requests using semisynchronous replication there are on average 3 cases of stale data. When using asynchronous replication there are on average 6 cases of stale data. This means that the percentage of stale data is 0.3% when using semisynchronous replication and 0.6% when using asynchronous replication. The standard deviation for these tests are low except for semisynchronous stale data. With an average of 3 and a standard deviation of 2.08 there is a higher risk that recreating these tests would not give the average value.

<table>
<thead>
<tr>
<th></th>
<th>Semisynchronous</th>
<th>Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg. nr of requests with stale data</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Percentage of stale data</td>
<td>0.3 %</td>
<td>0.6 %</td>
</tr>
<tr>
<td>Standard deviation of stale data</td>
<td>2.08</td>
<td>1</td>
</tr>
<tr>
<td>Average response time</td>
<td>239 ms</td>
<td>234 ms</td>
</tr>
<tr>
<td>Standard deviation response time</td>
<td>3.31</td>
<td>5.76</td>
</tr>
</tbody>
</table>

Table 4.7: Measurements regarding semisynchronous and asynchronous replication

4.4 Failover time after a fail-stop-failure

Running the test plan regarding the response time with 25 users gave a stable result. To measure the failover time and the response time for the system we simulated a crash by shutting down our primary server during the runtime of the test plan.

In Figure 4.7 there is an interruption in the frequency of completed requests. This is due to 20 unsuccessful requests, since the graph only shows requests which has been successful. After a short period of time the requests were successful again but were now redirected to the backup server. Worth mentioning is that after the crash the average response is slightly lower than before. This is because our slave database does not write to a binary log which
4.4. Failover time after a fail-stop-failure

decreases the work compared to the master. In these results the connect time for the requests was subtracted to minimize the travel time and error margins. In the figure there are 25 requests sent at second 15, these requests had a connect time at roughly 3.5 seconds since the proxy was in a state where it still not had confirmed that the primary was down. With this in mind the MTTR for this experiment is 4 seconds, which can be seen in figure 4.7 at the interrupt between 10 and 14 seconds.

![Figure 4.7: Response time of requests being made during a fail-stop scenario](image)

Comparing these results with the original systems behaviour at a fail-stop as shown in Figure 4.8 there is an obvious difference. The original system has no tolerance against a fail-stop and therefore as soon as the primary server stops working there are no more successful requests until manually started again giving us a higher MTTR.

![Figure 4.8: Response time of requests being made during a fail-stop with the original system](image)
5 Discussion, Conclusion and Future work

In this chapter we discuss the results from chapter 4 and argue for these results using the fundamental concepts presented in chapter 2. To answer our problem formulations a conclusion is drawn from the results and presented in section 5.2. Finally, in section 5.3 we present some interesting aspects of this thesis which could be analyzed further in future work.

5.1 Discussion

In this section we discuss the results and the method that we used. The work will also be discussed in a wider context.

5.1.1 Method

When simulating the load of the server a thread sends a new request as soon as it receives a response. By having a fixed amount of requests to send, the load of the system will be lower at the end of each measurement since no new requests are being made. Also, in the case of comparing the proportions of read- and write requests, all of the read requests are finished before the write requests because a read request has a lower response time.

The experiment measuring the occurrence of stale data is not likely to reflect how the system would act in reality. The method in this thesis can show the existence of stale data, not how the data would be perceived in real use of the system. To investigate the percentage of stale data found in a real scenario, there need to be experiments for the specific scenario.

The method used in this thesis should provide high replicability of the measurements, with the state of the network being the only varying condition. Although different network conditions would provide different values, the behaviour and conclusions should not be affected. This is also shown by the standard deviation for each result. All deviation values except the stale data for semisynchronous replication are low. This implies that there would most likely be no big differences between measurements if recreating the experiments.

When measuring how the load affects the response time there is a fixed number of simulated users sending at the same time and the traffic does not vary during an experiment, which does not reflect the behaviour of real traffic. If there was a better method of sending requests that better reflected variation of load the validity of the method would improve.
An alternative when designing the system would be to have the web servers and databases run on different servers. By doing so, there could be even more to gain by using load balancing. The traffic could then have been balanced between different web servers regardless of which type of requests it was. This would better utilize the capacity of the web servers, although the system would need another layer balancing the requests between the database servers. Another aspect of the architecture which has given a bit misleading results is the placement of the proxy server. The server is placed physically in London which makes the trip time for requests through the proxy about 60ms higher.

5.1.2 Results

During the experiments in section 4.1, it can be seen that the average response time increases as the number of users increases. According to Nielsen[15], a user will notice a user interface delay that is larger than 0.1 seconds. It will not feel instantaneous but the user’s flow of thought will stay uninterrupted for up to 1.0 second. This means that the system would not be able to handle 100 users satisfactorily without getting an unwanted behaviour.

In many of the measurements there are spikes of increasing response time, see Figure 4.1 or 4.5. Our initial guess was that these spikes occurs from MySQL’s checkpointing mechanism. MySQL uses fuzzy checkpointing which means that it flushes batches of dirty pages over time instead of flushing all at once. When exposing to a write-intensive load a lot of requests is written to the log. If this log gets filled up, MySQL executes a sharp checkpoint causing a temporary reduction in throughput ending up in an increase of response time. However, since measurements with 100% read requests showed similar behaviour the guess is most likely wrong. We were not able to identify the real cause of these spikes during our allowed time.

By comparing the average response time when using load balancing compared to not using load balancing there was a clear difference in performance for higher loads. For 25 users the effect of using load balancing only showed a small decrease in the response time and a small increase in average requests per second. For 50 users on the other hand, the response time was drastically lower and the average request per second was much higher when using load balancing, especially when the proportion of read-requests were higher. This means that the gain of using load balancing clearly drops when there is a bigger proportion of writes, which is reasonable since fewer requests can be sent to the backup server.

Comparing the two different ways of implementing database replication showed as expected a trade-off between increasing the average response time and decreasing the amount of stale data. As mentioned earlier a load of 25 users was used simultaneously to measure the average response time and another test to show existence the stale data. This ended up in the asynchronous replication having 5ms lower average response time for this load and about the double amount of stale data compared to the semisynchronous replication. One could argue that if it is important to not have a high risk of stale data, the better choice would be to use semisynchronous replication since it provides stale data less often. However, since there is no guarantee that there is no stale data the decrease in stale data might not be good enough. If you can accept stale data to some extent, the difference in the amount of stale data when using semisynchronous replication instead of asynchronous replication may not be worth sacrificing having a lower response time. Therefore, choosing semisynchronous solely to decrease stale data might not be worthwhile.

However, semisynchronous replication does provide lower risk of data loss and in those cases one should consider if reducing the risk is worth the higher response time. If it is essential that the data is up to date at all times neither semisynchronous- or asynchronous replication will do when allowing read-request from the backup server. You either need to only allow read-request to the master or need to implement a fully synchronous replication, and both would increase the response time.
As the main purpose is to implement a more fault-tolerant version of the original system the effect of a fail-stop needed to be investigated. The interruption clearly demonstrated a period of time where the system is unavailable. This period of time could be changed by configuring the proxy by varying the frequency of the health checks and how many failed checks there must be before switching servers. This however, has not been investigated in this report. In other words, this MTTR could be shortened but then other difficulties such as starting a failover process even though the server is running may occur. Looking at equation \[2.1\] there is the MTTF value as well. This value is harder to measure without doing a long time experiment, but the equation clearly shows that a decreased value of MTTR gives an overall higher availability.

The results from the experiment in Figure 4.7 shows that the system can provide fault tolerance against a fail-stop of the primary server. However, since the proxy has no backup and is the only way to access the system, the system has still got a single point of failure. Even though the system still has a single point of failure, it should be considered as more fault-tolerant and should provide higher availability than the original system. Compared to the web server the proxy requires less configurations that are prone to cause failures and the software that is used is widely considered as stable. This means that software failures of the proxy are less likely than those at the web servers.

5.1.3 The work in a wider context

The mock-up system is supposed to be used by the healthcare during emergency situations. This means that there may be huge consequences if the system would fail, since it could affect the health of patients. This raises ethical dilemmas regarding who is to blame if the system failure leads to harm to the health of humans. The engineer has a responsibility to develop a system that is reliable. If the cause of the failure could be prevented, or is expected to be prevented one could argue that the engineer can be blamed for the situation. No matter how hard one tries to prevent faults, the risk of a failure of the system exists. This means that the healthcare personnel have some responsibility to know how to manage situations where the system is not available.

5.2 Conclusion

In this thesis we have considered a system that is redesigned to improve the fault tolerance. We have investigated how implementing fault tolerance would affect the performance of the system.

From the experiments it is clear that the original system had lower response time for lower loads. When the load was increased there was a big difference in response time. By allowing load balancing this difference decreased dramatically when there was traffic with a high proportion of read requests. From the experiments we also saw that semisynchronous replication is slower than asynchronous replication but has less occurrences of stale data. It is clear that the system provides fault tolerance against fail-stops of the primary server and the experiments show that the system has a failover time of a few seconds.

We can see from the experiments that implementing fault-tolerance need not have a negative impact on the performance. How much it impacts performance depends on the load of the system, proportions of read requests and what type of guarantees against data-loss one needs. For traffic with high load and a high proportion of read request the experiments suggests that had the proxy been located closer to the servers, the implementation would not only improve fault tolerance, but may have also improved the performance of the system.
5.3 Future Work

Our work has covered a few aspects in the back-end system which affects the response time of
the system. During this work we realized that the load balancer can be configured in a lot of
different ways that impacts the results. There are a lot of parameters which could be changed
to get different trade-offs and response times. One interesting subject to look deeper into
would have been the algorithm used by the proxy to forward packets. This thesis covered
the use of round robin, but using another algorithm could possibly improve the average
response time.
Bibliography


Upphovsrätt
Detta dokument hålls tillgängligt på Internet – eller dess framtidss ersättare – under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

Copyright
The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Rickard Hellenberg, Oskar Gustafsson