Master Slave Database Replication in IOT

By

George Rohan Mathew

(2047228)

Under the guidance of

Dr. Rajani Poonia

&

Mr. Anuj Saxena

A Project report submitted in partial fulfilment of the requirements for the award of the degree of Master of Computer Applications of CHRIST (Deemed to be University)

March - 2021

CERTIFICATE

This is to certify that the report titled ***Master Slave Database Replication in IOT*** *is a bonafide record of work done by* ***George Rohan Mathew (2047228)*** *of CHRIST (Deemed to be University), Bengaluru, in partial fulfilment of the requirements of 4th Semester MCA during the year 2021-22.*

Head of the Department **Project Guide**

Valued-by:

	Name	: George Rohan Mathew
1.	Register Number	: 2047228
	Examination Centre	: CHRIST (Deemed to be University)
2.	Date of Exam	:

Trivium eSolutions

PRIVATE & CONFIDENTIAL

April 11th, 2022

Name: GEORGE ROHAN MATHEW
Intern No. : I 023

Subject: Internship with Trivium eSolutions Pvt. Ltd.

This is to certify that GEORGE from Christ University is undergoing his internship program with Trivium eSolutions Pvt. Ltd., as an Intern between December 6th, 2022, and June 30th, 2022. His performance and competence during this internship are satisfactory.

Yours sincerely,
for **Trivium eSolutions Pvt. Ltd.**

[illegible]

Surbhi Anand
Manager-HR

★ Trivium eSolutions Pvt. Ltd. ★ Bangalore

ACKNOWLEDGEMENTS

First of all, I thank God almighty for his immense grace and blessings showered on me at every stage of this work.

I am greatly indebted to Dr Joy Paulose, Head, Department of Computer Science, CHRIST (Deemed to be University) for providing the opportunity to take up this project as part of my curriculum. I am greatly indebted to PG Coordinator, Dr Tulasi B, for providing the opportunity to take up this project and for helping with her valuable suggestions.
I am deeply indebted to our project guide Dr Rajani Poonia, for her assistance and valuable suggestions as a guide. He made this project a reality.

I am deeply indebted to my industrial supervisor Mr Anuj Saxena and my other company mentor Mr Ashwin Baby, CEO Mr Deepak Nakra for rendering support during the project, and all my colleagues for their valuable suggestions and contributions to make this project a reality.

I express my sincere thanks to all faculty members and staff of the Department of Computer Science, CHRIST (Deemed to be University), for their valuable suggestions during the course of this project. Their critical suggestions helped us to improve the project work.

Acknowledging the efforts of everyone, their chivalrous help in the course of the project preparation and their willingness to corroborate with the work, their magnanimity through lucid technical details lead to the successful completion of my project.

I would like to express my sincere thanks to all my friends, colleagues, parents and all those who have directly or indirectly assisted me during this work.

ABSTRACT

Replication basically means to have a copy or to replicate. So in the case of Databases, we have one database which has all the data that we want to save and we will have the exact copy of that data in another database or machine. We will call another database which has the copy of the first one as a replica. So the Database which has the main data or main source of writes and updates becomes the primary database, known as Master and the database which has the copy or replication from the primary database is known as secondary, known as Slave.

In my project "**Master Slave Database Replication in IOT**", for fault tolerance, scalability, and performance, database replication is commonly employed. The failure of one database replica does not prevent the system from functioning since other replicas can take over the failed replica's functions. The load can be distributed over all replicas for scalability, and more replicas can be added if the load grows. Finally, if data copies are stored close to clients, database replication can give rapid local access even if clients are geographically spread.

Despite its benefits, replication is a difficult process to use, with numerous obstacles to overcome. The most important aspect is replica control, which ensures that data copies remain consistent when updates are made. There are numerous options for where updates can take place and when changes are propagated to data copies, as well as how changes are implemented and where the replication tool is stored. Combining replica control with transaction management presents a unique difficulty since it requires numerous activities to be handled as a single logical unit while also ensuring atomicity, consistency, isolation, and durability across the replicated system. The book categorizes replica control mechanisms, goes through numerous replica and concurrency control mechanisms in depth, and covers many of the challenges that arise when such solutions are implemented within or on top of relational database systems.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

The project's key aims are discussed in the introduction, as well as the project's context. This aids in comprehending the organization's profile as well as the problem we've uncovered. This section contains a full organisation profile that includes information on the company, as well as its different goods and services. Along with the organisational profile, there is a background study that deals with the primary study conducted to understand the problems; thus, the background study deals with the problem description, existing system, disadvantages of the existing system, advantages of the proposed system, and project scope; all of these studies give a clear idea of the project's need and importance.

1.1 Organization Profile

Trivium provides consulting and software development services through our Munich and Bangalore locations, which operate on a dispersed delivery model. Trivium's technological expertise spans all major platforms, including Java,.NET, C++, and mobile platforms such as iOS, Android, Windows Phone, and Cross-Platform Technologies.
We now have over 150 employees and a number of long-term customers from a variety of industries. Large organisations, as well as a number of mid-sized businesses and start-ups, are among them. Trivium has operations in Munich and Bangalore, giving our customers the best of cultural understanding, competitiveness, and scalability when it comes to software development projects.
Enterprise software, IoT, analytics, cloud, and mobile apps are all areas of expertise for us. We have a strong foundation in IoT platforms, Cloud technologies, Analytics, and Machine Learning. Learning, Java / JEE platform, Microsoft technologies, C++, DevOps and mobile development tools.

Trivium has always prioritised social responsibility. Over the years, we've undertaken a number of programmes focused at addressing societal issues in a positive way. In 2017, we focused our CSR efforts on adopting and transforming a primary school in Gollahalli, a community roughly 70 kilometres from Bangalore. Trivium totally restored and outfitted the rural school, which has 90 students in grades 1 to 7, with new classrooms, modern bathrooms, desks, play equipment, a kitchen, and clean drinking water. The youngsters can now enjoy and benefit from the greatly improved facilities, which were launched in November 2017.

1.2 Background Study

Background research aids in comprehending the challenge and the significance of the endeavour. The research aids in the comprehension of the problem, the current system, and the proposed remedy. The service is designed to assist users in more effectively processing and analysing their data. The NGS analysis, which entails a great deal of computing and analysis, must be carefully planned and analysed. The background research also aids in comprehending the project's significance. The problem description, the existing system study, the shortcomings of the existing system, the advantages of the proposed system, and the scope and importance of the suggested solution are all covered in the background study.

1.3 Project Description

To improve data availability and accessibility, as well as system resiliency and reliability, data replication stores the same data in different locations. Disaster recovery is a typical use of data replication to maintain correct backups in the case of a disaster, hardware failure, or system breach that compromises the data.
Replica can also help enterprises with a large number of sites access data faster. When accessing data from North American data centers, users in Asia and Europe may notice delays. By storing a replica of your data closer to your users, you can optimize access time and balance network load.

The replicated data can also help to improve and optimize the performance of the server. Users can swiftly access data when a corporation runs several replicates on multiple servers. Furthermore, by sending all reads to the replica, the administrator can free up the original server's processing cycle to do more resource-intensive writes.

1.4 Objective

The purpose or objective of this solution is:

- Availability: If one of the sites providing a relationship R fails, another site can provide the relationship R. This means that even if one side fails, the question can still be processed (see Relation R).
- Increased parallelism Sites with R relations can execute queries in parallel (using R relations). This shortens the time it takes for a query to run.

1.5 Purpose and Applicability

1.5.1 Purpose

Data replication is the process of creating numerous copies of your data and storing them in various locations across your network to increase backup, fault tolerance, and general accessibility. Data replication, like data mirroring, can be used on both individual PCs and servers. Data Replicas can be kept on the same system, on onsite and offshore hosts, and in the cloud.

Today's prominent database platforms either feature built-in data replication capabilities or rely on third-party software. Although Oracle Database and Microsoft SQL actively offer data replication, this feature may not be available by default in some legacy technologies.

1.5.2 Applicability

This application's primary and direct area of service is to support high availability, backup, and/or disaster recovery.

2. SYSTEM ANALYSIS

This system analysis aids in the more effective design of the software tool. The program is designed for bioinformaticians who are unfamiliar with the Linux environment. Here we can see what the application's distinct requirements are. The application's requirements are straightforward, but integrating the many technologies into an application necessitates careful design and study. For effective development, the feasibility of doing so must be assessed and determined. The requirement specification, which covers both functional and non-functional needs, is the focus of the system analysis. The block diagram and system requirements are also included in the system analysis. The system requirements describe the application's hardware and software needs, as well as the external tools that were utilized to create it.

2.1 Requirement Specifications

The specification of a requirement is a collection of all the requirements that must be imposed on the product's design and verification. Other information required for the product's design, verification, and maintenance is also included in the specification. It's a description of a software system that's going to be built. It is based on a stakeholder requirements specification, which is also known as a business requirements specification. The functional and non-functional needs are included in the application's requirements definition for next-generation sequencing analysis procedure. The functional requirements define the application's functional requirements, whereas the non-functional requirements define the application's additional feature sets.

2.1.1 Functional Requirements

Product features or functions that developers must implement in order for users to complete their duties are known as functional requirements. As a result, it's critical to make them explicit for both the development team and the stakeholders. Functional requirements, in general, describe how a system behaves under specified circumstances. The application's key functional requirements are to interface tools, pipelines, and applications. The program should develop an interface for collecting the required inputs and other parameters for the tools, and then triggering the execution with a button click. The application may also keep track of how things are going.

2.1.2 Non-Functional Requirements

Security, reliability, performance, maintainability, scalability, and usability are examples of non-functional criteria. They act as limits or limitations on the system's design across the various backlogs.... They ensure that the entire system is usable and effective. The application's non-functional needs include a requirement that sets criteria that can be used to judge a system's functionality rather than specific actions. Functional requirements, on the other hand, define precise behaviour or functions. The application's correct operation is included in the non-functional criteria.

Non-functional requirements also include the application's error-free operation without crashing the system. In the worst-case scenario, the program should not misbehave in any way. The application should also be simple to use for the user, with a minimal user interface. The program should also assist users in quickly and simply finding what they are looking for.

2.2 Problem Definition

A common use for data replication is disaster recovery. This ensures an accurate backup at all times, even in the event of a disaster, hardware failure, or system breach that puts your data at risk. Replicas can also speed up data access, especially in organizations with a large number of sites.

2.3 Block Diagram

The given diagram depicts the overall system design and its functions with associated components. This report explains the overall architecture and working of the system that how master slave communicate with other and what all files are related, how data is fetched.

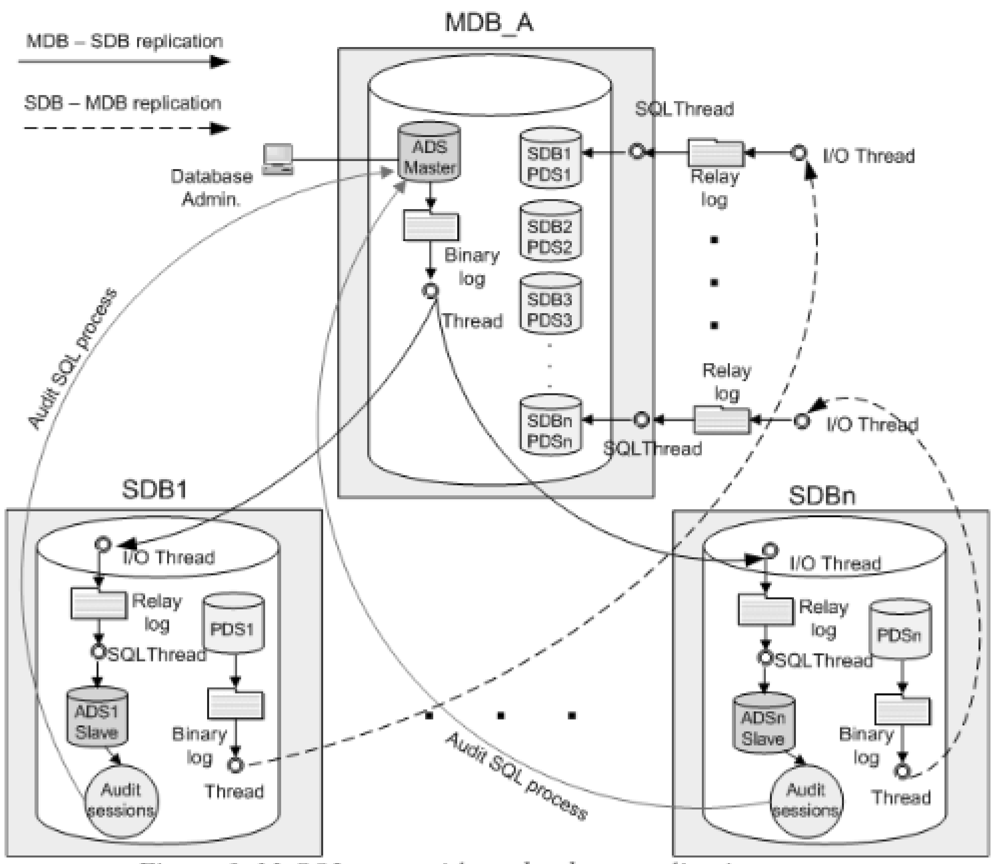


Figure 3. MySQL master/slave database replication process

Fig 2 Block Diagram

2.4 System Requirements

This phase, which served as the foundation for all subsequent development processes, was devoted to analysing the planned and existing systems and acquiring precise requirements from our clients. During this phase, an analysis is carried out at many levels in order to comprehend the actual needs, which serves as a useful input for our system design.

The basic system requirements for the application to work and developed and tested are as follows, it is divided into two, hardware and software requirements. These are the minimum requirements for the application to work smoothly, however, the application with the tools requires a minimum of these requirements to work smoothly.

2.4.1 Hardware requirements

The expected hardware requirements for the effective performance of the application have the following considerations and specifications.

- Hardware: Raspberry Pi 3.0.
- Memory: 500 KB.
- Storage: 8 GB
- Processor: Any processor @2.3 GHz .

2.4.2 Software requirements

The expected software requirements for the effective performance of the application have the following considerations and specifications.

- OS: Ubuntu 20.04.2.0 LTS (64 bit) / Debian 10 .
- Frontend / Backend Tools: Linux Terminal
- Database Tools: MySQL / MariaDB

2.4.2.1 Ubuntu 20.04.2.0 LTS

Ubuntu 20.04 LTS [3] is based on the Linux 5.4 long-term support series. The HWE stack has been updated to version 5.8 of the Linux release series. New hardware is supported, including Intel Comet Lake CPUs and the first Tiger Lake platforms, AMD Navi 12 and 14 GPUs, Arcturus and Renoir APUs, and Navi 12 + Arcturus power features. The ex-FAT filesystem, virtio-fs for sharing filesystems with virtualized guests, and fs-verity for detecting file alterations have all been included.

Support for the Wire Guard VPN is built-in, and lockdown in integrity mode is enabled. Ubuntu Desktop now uses the HWE kernel by default. It means that starting in January 2021 and continuing until the summer of 2022, the Ubuntu Desktop will receive new major kernel versions every six months, even if you installed Ubuntu Desktop earlier. It is a powerful operating system for building analysis systems.

2.4.2.2 MySQL

MySQL is a key component of the LAMP (Linux, Apache, MySQL, and PHP) open source corporate stack. LAMP stands for Linux as an operating system, Apache as a web server, MySQL as a relational database management system, and PHP as an objectoriented scripting language for web development. (Perl or Python are sometimes used instead of PHP.)

MySQL was created by MySQL AB in Sweden and was acquired by Sun Microsystems in 2008, and subsequently by Oracle when it bought Sun in 2010. MySQL is available under the GNU General Public License (GPL) for developers, but businesses must purchase a commercial license from Oracle. MySQL is now the RDBMS of choice for many of the world's most popular websites, as well as a slew of corporate and consumer-facing web-based apps like Facebook and Twitter and YouTube.

MySQL has a client-server architecture. MySQL's fundamental component is the MySQL server, which manages all database commands (or commands). The MySQL server is available as a standalone program for usage in a client-server networked environment, as well as a library that may be integrated (or linked) into other programs. MySQL is used in conjunction with a number of utility packages that aid in the management of MySQL databases. The MySQL client, which is installed on a computer, sends commands to MySQL Server.

MySQL was created with the goal of quickly processing big databases. Despite the fact that MySQL is normally installed on a single system, it can transfer the database to several places because users can access it through various MySQL client interfaces. SQL statements are sent using these interfaces to the server and then display the results.

2.4.2.3 MariaDB

MariaDB is a MySQL fork. To put it another way, it's a better, drop-in substitute for MySQL. A drop-in replacement means that you may use the analog version of the MariaDB server to replace the regular MySQL server and benefit from all of MariaDB's advancements without having to change your application code. MariaDB is a fast, scalable, and reliable database. It has a larger number of storage engines than MySQL. MariaDB also comes with a slew of plugins and utilities that make it adaptable to a wide range of scenarios.
MySQL was purchased by Sun Microsystems in 2008. Then, in 2010, Oracle bought Sun Microsystems, which contained MySQL. For some reason, MySQL's founder,

Michael Monty Widenius, chose to fork MySQL and start a company called Monty Program AB. MariaDB is the name of his second daughter, Maria. The MariaDB Foundation was founded in December 2012 to prevent a firm from acquiring MariaDB, as happened with MySQL. The MariaDB Foundation's goals are to ensure MariaDB's long-term viability and to operate as a global hub for open collaboration.

3. DATABASE REPLICATION

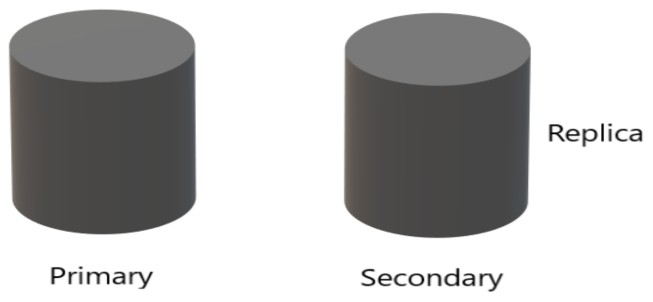


Fig 3

Replication basically means to have a copy or to replicate. So in the case of Databases, we have one database which has all the data that we want to save and we will have the exact copy of that data in another database or machine. We will call another database which has the copy of the first one as a replica. So the Database which has the main data or main source of writes and updates becomes the primary database, known as Master and the database which has the copy or replication from the primary database is known as secondary, known as Slave.

We'll take the example of an internet store selling puppets to better illustrate our point. The store's website allows customers to search the inventory of puppets available and purchase the ones they choose. The corporation maintains a database behind the web server that contains all product information, including photos, stock levels for each product, and pricing information. It also keeps track of clients that are currently or were previously involved in a transaction. So, why would our firm want to duplicate the information.

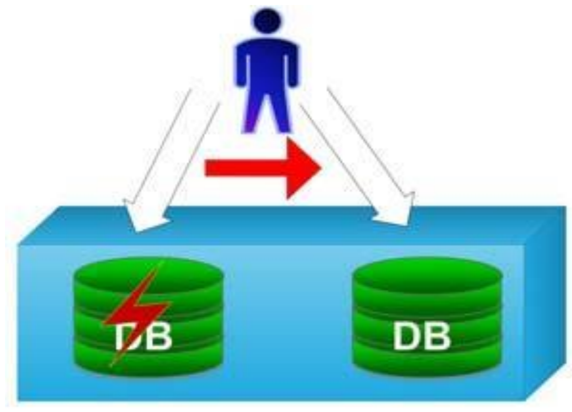


Fig 3.1

The first factor is tolerance for mistakes. Regardless of failures, our puppet firm wants to ensure that its customers have access to their system 24 hours a day, seven days a week. A failure can occur when the database software process fails, the physical machine crashes (for example, due to hardware failure), or the link between the client and server systems is (temporarily) disrupted due to a network failure. Our organization install two copies of the database on different nodes to address these failure scenarios. A node is also referred to as a replica in cases where the complete database is copied.

If one of the replicas fails, there is still one replica that is operational. The system can withstand the loss of one of the copies. Because the service stays available despite failures, this is also known as high availability. The replicas in most high-availability solutions are connected to the same local area network, as shown in Figure 2.1, to allow for quick communication between them. Any failure is detected via a failure detection mechanism, and clients connecting to the failed node are re-joined to the accessible node, where request execution resumes.

However, such a solution cannot handle network failures between clients and databases because the client will be unable to connect to any of the replicas.

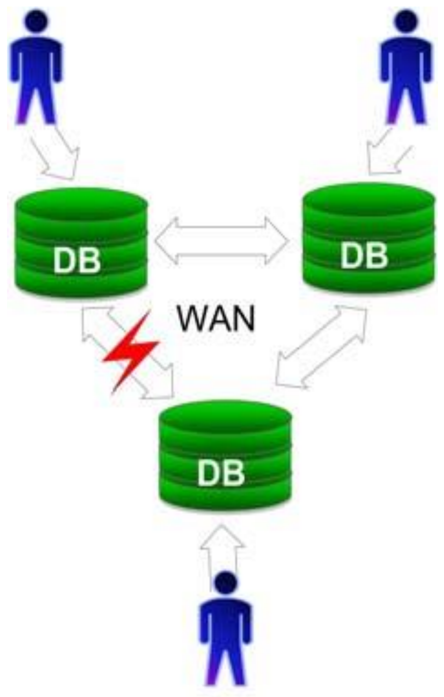


Fig 3.2

Wide area replication can be utilized to solve this problem (see Figure 2.2). The database replicas are geographically distributed in wide area replication, and each client connects to the closest replica. Even if a remote replica is unavailable, the local replica is generally accessible. However, because of the likelihood of network partitions, the latency of the wide area network poses fascinating issues, and failover becomes more complicated.

Performance is a second key application for replication, as it can help boost throughput and minimize response time. Let's return to our puppet shop. The organization can build numerous replicas within a local area network as a first option, resulting in a cluster (see Figure 2.3).

To the outside world, the cluster seems to be one unit, and requests are distributed across the copies as they arrive. The system can scale up to meet rising demand by adding new replicas. Replication fulfils the objective of scalability in this scenario because it can deliver increased throughput.

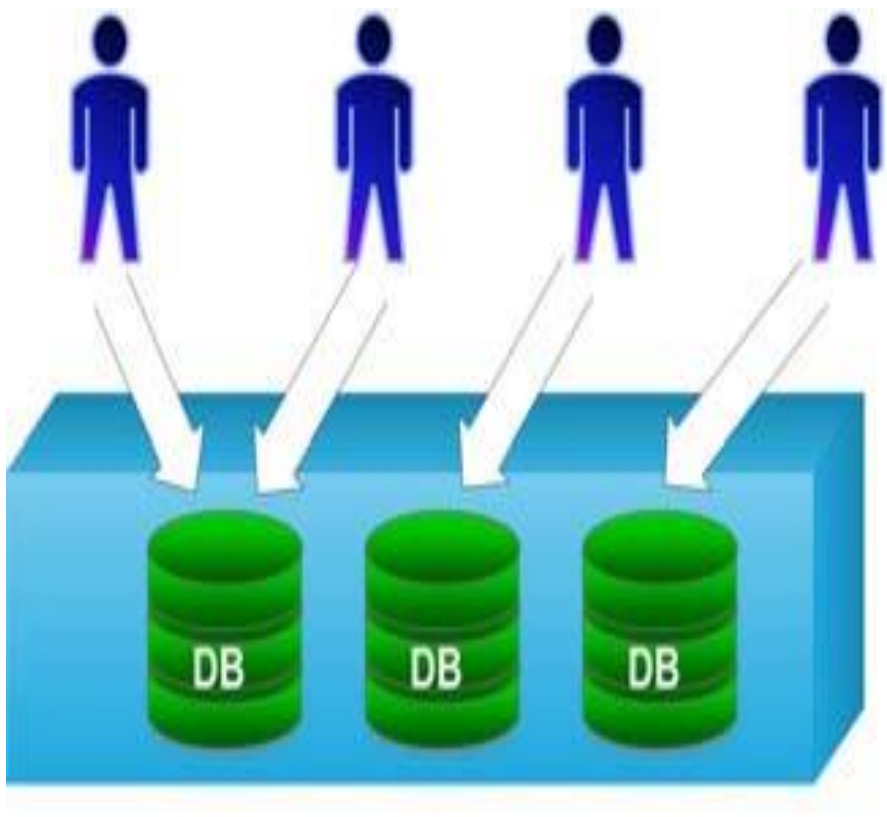


Fig 3.3

3.1 Eager Database Replication

Since Database Replication mostly faces performance issues, eager database replication was introduced. It is all about consistency and is being implemented using three key parameters. Since it is being used for both fault tolerance and performance purposes mostly there would be a compromise between the efficiency of it and the consistency. A great number of research databases are based on asynchronous replication which is also called as lazy update model. But a lot of consistency are being compromised. And consistency is a very significant aspect as well. The consistency issues in the lazy replication are difficult to be solved.

So mostly this eager approach is also being used in synchronous replication as well. A change that is being made synchronises with all the copies that are they before the commit happens.

3.1.1 Eager Database Replication : System & Architecture

Let there be set of clients for database and S={s1,s2,s3,…,s_n}, S being a set. Full Replication of data is there on all the servers that is each server has copy of the whole of the database. The client can connect to any of the servers, s_i to execute a query. Queries basically can do any read operation. But when a write operation is taking place, it is called update transactions. The transactions that are to be done can be done with a single message or operation by operation. Say the client gets connected to the server S_i. Operation by operation basically means interactive transaction. If a client wants to do a read request as well as a write operation, then the client basically sends a request for read first and wait for the server to respond. As soon as client gets the data, they sent the write request next. Stored Procedures is basically as group of statements in sql being considered as a logical unit. Such stored procedures are called Service Request. A transaction outcome also has the corresponding results. If a transaction is not done if S_i fails, then it is up to the client to see if they should retry it or not. They can try connecting to any other server and raise the same service request.

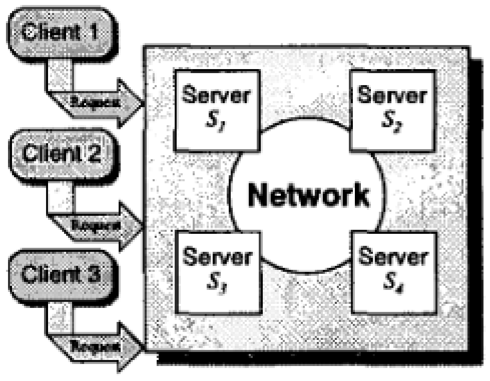


Fig 3.4

The servers of the database communicate with each other through 1 to n or point to point communication. Group communication primitives have two different parameters: Ordering and Reliability of message delivery.

Regarding the reliability of message delivery Reliable Broadcast and Uniform reliable broadcast takes care of it. A message which is being send the right database server or the message that is being delivered by the right database is eventually been delivered to all the correct databases by the Reliable Broadcast. Similarly, a message that is being delivered by any server is eventually send to all the correct databases (This messages can be right or failed right after the delivery) by the Uniform Reliable Broadcast.

The Total Order Broadcast takes care of the order of the messages. It can either be delivered in the same order or in the arbitrary order.

3.1.2 Classification Criteria

Eager replication techniques can be classified using three characteristics that characterise the protocol's nature and properties.

The server architecture (primary copy or update everywhere), how modifications or operations are propagated across servers (per operation or per transaction), and the transaction termination protocol are all aspects to consider (voting or non voting).

3.1.2.1 Classification Criteria : 1) Server Architecture

A specific site is required for the primary copy replication. Whatever changes or updates that are being made should initially go to the primary copy wherein an analysis would be done to fix the serialization order. If primary crashes by any chance, one of the other servers would take up the role of the primary. For avoiding bottlenecks, the database doesn’t usually make only one primary. It would have multiple as in multiple primary for a group of subsets belonging to a particular category.

Updates to a data item can be made from anywhere in the system with update everywhere replication. That is, updates can arrive at two separate copies of the same data item at the same time (which cannot happen with primary copy). Update everywhere techniques are more gentle when dealing with failures as a result of this attribute, as no election protocol is required to continue processing. In the same way, updating everything does not, in theory, cause performance bottlenecks. Update everything, on the other hand, may necessitate that, rather than one site doing the work (the primary copy), all sites execute the same task. If the design isn't thorough, update everywhere can have a considerably bigger impact on performance than primary copy alternatives.

3.1.2.2 Classification Criteria : 2)Server Interaction

The degree of communication among database servers during the execution of a transaction is the second criterion to examine. The quantity of network traffic created by the replication mechanism, as well as the overall overhead of processing transactions, are determined by this. This parameter is a function of the number of messages required to complete a transaction's operations (but not its termination). Furthermore, the type of primitive used to exchange these messages will influence the protocol's serialisation features.

We'll look at two scenarios:

Constant interaction refers to systems that use a certain amount of messages to synchronise the servers for a specific transaction, regardless of the number of operations in the transaction.

Typically, protocols of this category send only one message every transaction, containing all of the transaction's operations in one message. Linear interaction is a term that refers to techniques in which a database server propagates each operation of a transaction one at a time. The operations can be delivered as SQL statements or as log records that include the results of doing the operation on a certain server.

3.1.2.3 Classification Criteria : 3)Transaction Termination

The final parameter to consider is how transactions are managed, or how atomicity is ensured.

We discriminate between two scenarios:

Voting Termination: To coordinate the various copies, an additional round of messages is required at the end of the voting process. This round can be as complicated as an atomic commitment protocol (for example, the two-phase commitment protocol or as simple as a single confirmation message sent by a single site.

Non-voting termination means that sites can determine whether to commit or abort a transaction on their own. Non-voting approaches necessitate deterministic behaviour from replicas. This is not, however, as limiting as it appears at first appearance. Because determinism only impacts transactions that are serialized in relation to one another, Transactions or processes that do not clash can be carried out in different orders at various locations. Because determinism only impacts transactions that are serialized in relation to one another, Transactions or processes that do not clash can be carried out in different orders at various locations.

3.2 The Fault Tolerant Master-Slave Protocol

3.2.1 The Master-Slave Replication Strategy

As previously stated, the master-slave replication approach employs three distinct entities: master, slave, and client. There will be only one master at whatever given moment. There may, however, be a large number of slaves and clients. The replicated data is held by the master and slaves. Clients, on the other hand, only serve as front-ends for processes that rely on duplicated data. As a result, the processes that consume the data, or users, do not need to know where the data replicas are located.

Because entities are frequently represented as processes in implementations, we will now refer to entities as processes. On duplicated data, a user can make two types of queries: read requests and read/write requests. Only the duplicated data is read in read requests. The replicated data may be updated as a result of read/write requests. Both types of requests must be sent to a client by a user. A client must pass a request to a slave when it receives one. After then, two things can happen when the slave receives the request. In the instance of a read request, the slave fulfils the request and responds to the client who made the request.

After that, the client sends the user the response. If the slave receives a read/write request, it sends it to the master. After that, the master processes the request and transmits an update of the replicated data to each slave. This ensures that the slaves have the most recent copy of the duplicated data. The master not only sends the updates, but also responds to the slave who forwarded the request. After getting the response, the slave forwards it to the client who made the request. Finally, the client sends the user the response. There are two conditions that must be followed while handling read requests and read/write requests. The first criterion is that all updates must be applied in the same order by the master and slaves. This ensures that there are no anomalies between copies as a result of differing update orders. The second criterion is that after a user submits a read/write request, the user must send another request that operates on a version of the replicated data that incorporates the read/write request's changes.

When a user's request is causally related to another user's request, this criterion ensures that the user's request is executed on the right data. A different form of condition is that the master can only send updates to each slave if it knows where each slave is. Every newly activated slave could send an announcement message to the master to inform him of the slaves' locations. Another benefit of sending this message is that it allows the master to send a copy of the replicated data to the activated slave. Unfortunately, there is a difficulty in that the slave must somehow discover the master's address. However, we may simply accomplish this by utilising a directory service that contains the necessary information.

3.2.2 The Non Fault Tolerant Master Slave Protocol

Following that, we'll go over a non-fault tolerant master-slave protocol that serves as a warm-up for the fault tolerant protocol. The state diagrams in Figures 2.1, 2.2, and 2.3 are used throughout the explanation. Every sort of procedure has its own state diagram. The diagrams are essentially directed graphs with states at the vertices and state transitions at the directed edges. A start state is a state transition that does not begin in any other state. When the process is not handling any events, it enters the state named after the process in question.

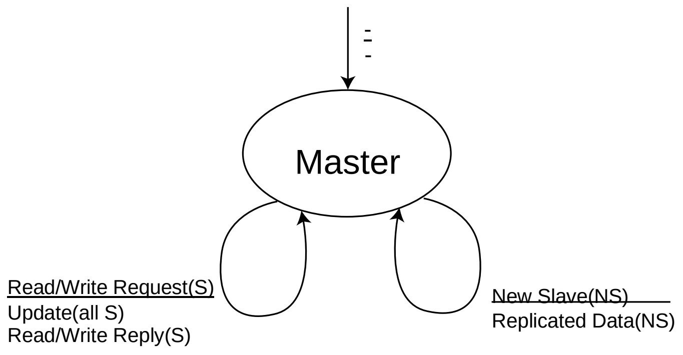


Fig 3.5

Every state and every state transition has a label in the state diagrams. A state's label makes it simple to refer to that state. A state transition's label can be used to describe two things. The label first describes the transition's reason. This could be a message from a process or a request from a user. Second, the label also specifies the process's externally evident reaction. This response could be a message sent to one or more processes, or a user response. Both portions of a transition label are separated by a horizontal line. The cause is always on the top of the line, and the response is always on the bottom. The internal changes induced by state transitions are not reflected in the transition labels. These modifications are only mentioned in the text.

Both the text of the cause and the text of the reaction may be a single dash in the labels associated with state transitions. No external event is required for the transition to occur if the cause is a dash. A dash indicates that no external reply is seen in the case of a response. We identify one or more letters between parentheses behind every cause and response that isn't a dash. The letters above the line in a state transition label tell us who caused the occurrence. The letters below the line indicate to whom a reaction is meant. Table 2.1 shows the many letter combinations that can be used.

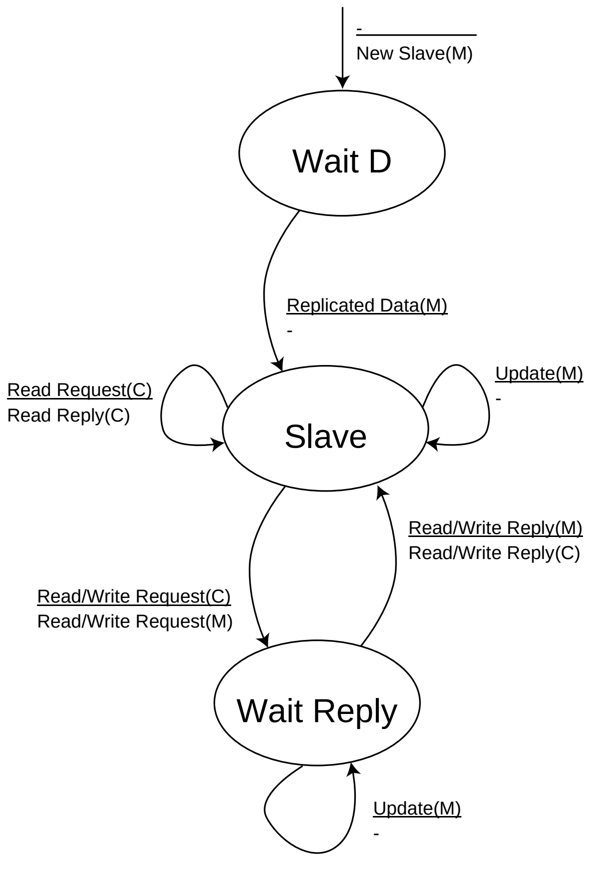


Fig 3.6

We now explain the non-fault tolerant protocol by means of scenarios that refer to the state diagrams. The scenarios revolve around the events given in the general explanation of the master-slave replication strategy. Only two types of events are possible: activation events and requests sent by users. There are three activation events:

master activation, slave activation, and client activation. The sending of requests has two possibilities: sending of a read request and sending of a read/write request.

Activation of the Master

The master's activation is the first situation we'll look at. The master enters the only state of Figure 2.1 when it becomes active. There are no acts that are apparent from the outside. When slaves use a directory service to find the master, the time of activation is, of course, a good time to register with the directory service.

Activation of Slave

Slave activation is more difficult than master activation. The first thing a slave must do is determine the master's address. It might be able to accomplish this by calling a directory service.

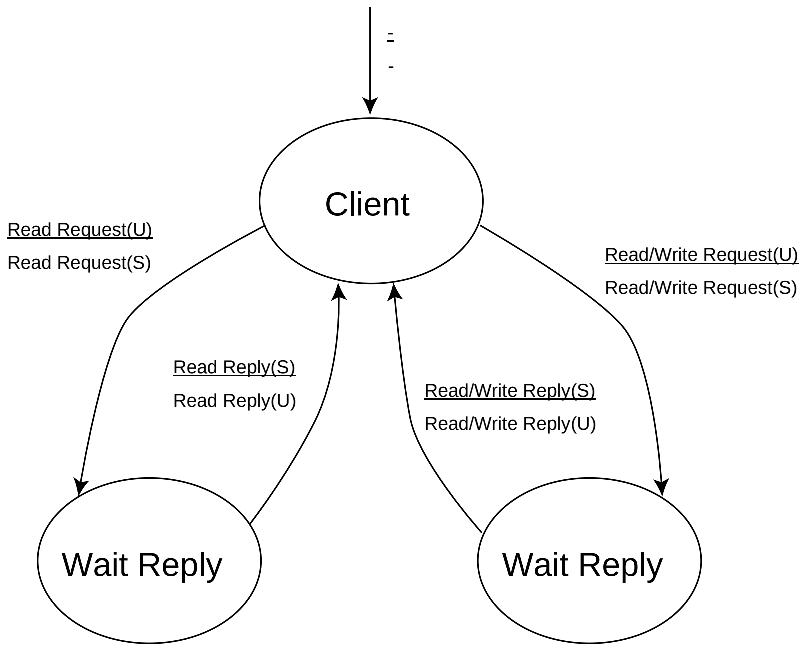


Fig 3.7

Meaning of all possible letter combinations in Figures

Letter combination	**Meaning**
M	Master
S	Slave
NS	New slave (an activating slave)
All S	All known slaves
C	Client
U	User

After that, The slave must inform the master that it has arrived. The slave accomplishes this by allowing the slave to send the master a message containing address information. Figure 2.2 shows the New Slave notification at the top. The slave enters the Wait D state after transmitting the message and waits for a response. The master eventually receives the New Slave message. As seen in the master state diagram, the master responds by providing a copy of the duplicated data. The slave's information is likewise saved by the master so that it can send updates later. A transition to the Slave state occurs when the slave receives the replicated data. This completes the slave's activation. The slave, like the master, may need to register with a directory service. This is a good time to do it right after receiving the replicated data, because the slave is only completely functional from that point on.

Activation of Customers

During activation, a client, like the master, does not need to do any externally observable actions. A client, on the other hand, requires the address of a slave in order to function. The moment of activation is an ideal time to get such an address. Alternatively, the client can wait until a user submits a request, as it is only then that the client requires the address. When a client is activated, it enters the Client state (see Figure 2.3).

Sending a Request to Read

When a user makes a read request to a client, the client passes it on to a slave. This is only done by the client when it is in the Client state. It is busy handling a request when it is in any of the previous stages, and the newly sent request must wait. Regardless, the client sends the request to one of the slaves, who gets it. When this occurs, the slave processes the request and responds to the client. The slave, like the client, only handles requests when it is in the Slave state. After forwarding the request, the client enters the Wait Reply state on the left side of Figure 2.3 to await the response. When the client receives the response, it returns it to the user, and the transition back to the Client state occurs.

Requesting a Read/Write Access

A client transmits a sent read/write request to a slave in the same way that a read request is forwarded to a slave. Unlike a read request, however, the client waits in the right Wait Reply state in the state diagram. Furthermore, the slave passes the request on to the master. When the master receives the request, he fulfils it. Following execution, the master gives each slave an update of the replicated data. In both their Slave and Wait Reply states, the slaves can handle this update. In addition to sending updates, the master responds to the slave who sent the request. This slave is in the Wait Reply state, waiting for a response. The slave returns to the Slave state after receiving the response. In addition, the slave relays the response to the right client. That client sends a response to the user who made the request.

We've now covered every feasible possibility. We didn't go into detail about how the protocol meets the requirements for ordering updates and executing requests with the most up-to-date version of replicated data. Fortunately, the protocol can meet both needs by relying on the FIFO network assumption described in Section 2.1, allowing only one network connection between each pair of processes, enabling each user to use a unique client, and allowing each slave to use a unique client. Let's start with how we can ensure that all updates are applied in the same order on both the master and slave machines. We know that the master is in charge of all read/write requests, which are the source of all updates. As a result, all slaves must apply all updates received in the same sequence as the master conducts the read/write requests. By providing all updates in the order of request execution, the master may quickly inform the slaves of the order. Because each slave has just one network connection and all network connections are FIFO, the network transmits all updates in the order that the master sends them.

The necessity that a request issued by a user after a read/write request sent by that same user execute with an up-to-date version of the replicated data is now discussed. Because there are two types of requests, we must evaluate two scenarios. However, in both circumstances, the request that comes after the read/write request is not treated until the read/write request's response is received. This is the result of users utilizing the same client all of the time. The client only processes one request at a time, and the FIFO assumption ensures that the requests are received in the correct sequence. We can see that in the case where both requests are read/write requests, the master must process both. The master is aware of the changes to the duplicated data made by the first request since the client sends the second read/write request only after receiving a response to the first. As a result, it can carry out the second request with the most recent version of the duplicated data.

When we consider the case when the second request is a read request, we can see that we have a situation where both the master and the slave must handle the request. What we know is that the master communicates with the slave via a single connection, which sends the read/write request. The FIFO assumption ensures that the slave knows which update is connected with the request since the master transmits the update associated with the request just before sending the reply associated with the request. Because a client always uses the same slave, the read/write and read requests must be seen by that slave. As a result, the slave knows which changes relate to the read/write request before it receives the read request since the client ensures the slave does not receive the read request until after receiving the reply to the read/write request. As a result, the slave can carry out the read request using the most recent version of the duplicated data.

It's worth noting that the procedure described above lacks the ability to shut down processes. A shutdown, on the other hand, is quite similar to a crash failure in that both stop the generation of output. We chose to postpone the addition of shutdowns to the next part because we will only address crash failures in that area.

3.2.3 The Fault Tolerant Master-Slave Protocol

We enhance the non-fault tolerant protocol by allowing processes to stop in addition to crash failure robustness. Every process is expendable, which is the underlying notion behind our fault tolerant protocol. What this means varies depending on the process. Being disposable in the case of the master process means that another process must be able to take over the master's job. Every slave process must be able to take over the role, we decided. Slave processes are appropriate since they have replicated data. To make the takeover totally conceivable, we decided that every slave, like the master, must have knowledge about the other slaves present.

Expendability has a different meaning for slaves. It's likely that there are numerous slaves because we're replicating for performance. This means that if a slave crashes or shuts down, other slaves can easily take over its clients. There are two issues to consider here. To stop providing updates, the master must first notice the slave that has shut down or crashed. Second, in order to achieve the greatest performance, we may need to activate a new slave. The protocol addresses the first issue by allowing the master to run a crash detection system to detect slave crashes, as well as allowing slaves to send a message to the master when they shut down. The second issue can be addressed by allowing the protocol to create new slaves dynamically. This was not mentioned in the protocol. This capability, however, should be simple to implement.
When it comes to clients, expendability refers to the ability for a user to start a new client in the event of a shutdown or crash. As might be deduced from the above description, we treat shutdowns similarly to crashes. We can accomplish this since shutdowns are extremely similar to crash failures in that they both stop output production. Because the crash detection system we suggest may be delayed, we don't treat shutdowns and crashes in the same way, as we'll see later in this section. During shutdowns, we take several steps to avoid the crash detection method.

The protocol is once again described using state diagrams and scenarios. The state diagrams are shown in Figures 2.4, 2.5, and 2.6, one for each type of process. The elements in the diagrams are the same as in the preceding section. There are, however, five new elements. To begin with, several of the state transition labels now incorporate italicized text. These italics briefly define a requirement that must be met before a process can proceed to the labelled transition. Second, an end state is now included in all state diagrams. When a process shuts down, it has reached its final state. End states are indicated by vertices with a double border. Third, a state transition can now be caused by a variety of factors. Backslashes separate the numerous causes in the labels associated with state changes. Fourth, the cause of some transitions does not include letters between parenthesis. The letters are missing because identifying a process or user from whence the event comes is impossible. More information can be found in the scenarios below. Between parenthesis, three new letter combinations appear: NM, all S - NS, and all C. The meanings of these novel letter combinations, as well as those that have already occurred, are listed in Table 2.2.

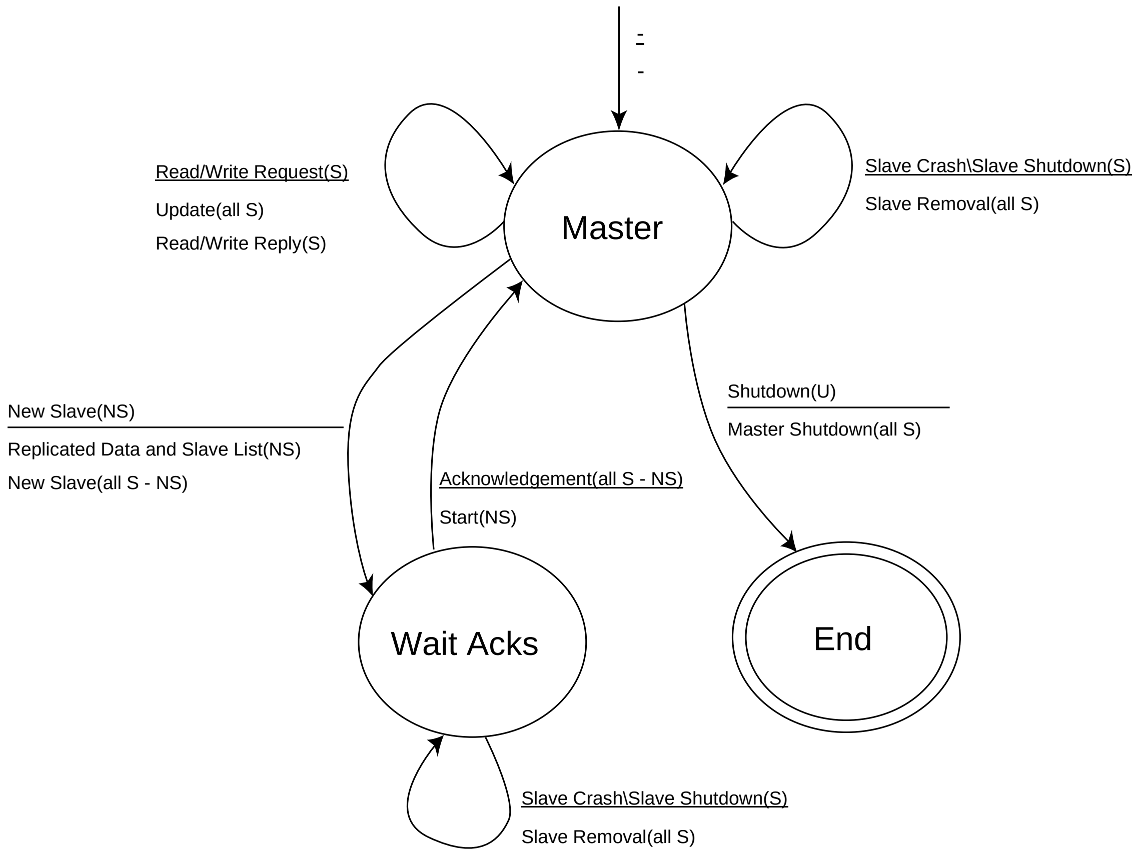


Fig 3.8

Meaning of all possible letter combinations in Figures

Letter combination	**Meaning**
M	Master
NM	New master after master crash
S	Slave
NS	New slave (an activating slave)
All S	All known slaves
All S - NS	All known slaves except a new slave
C	Client
All C	All clients associated with a specific slave
U	User

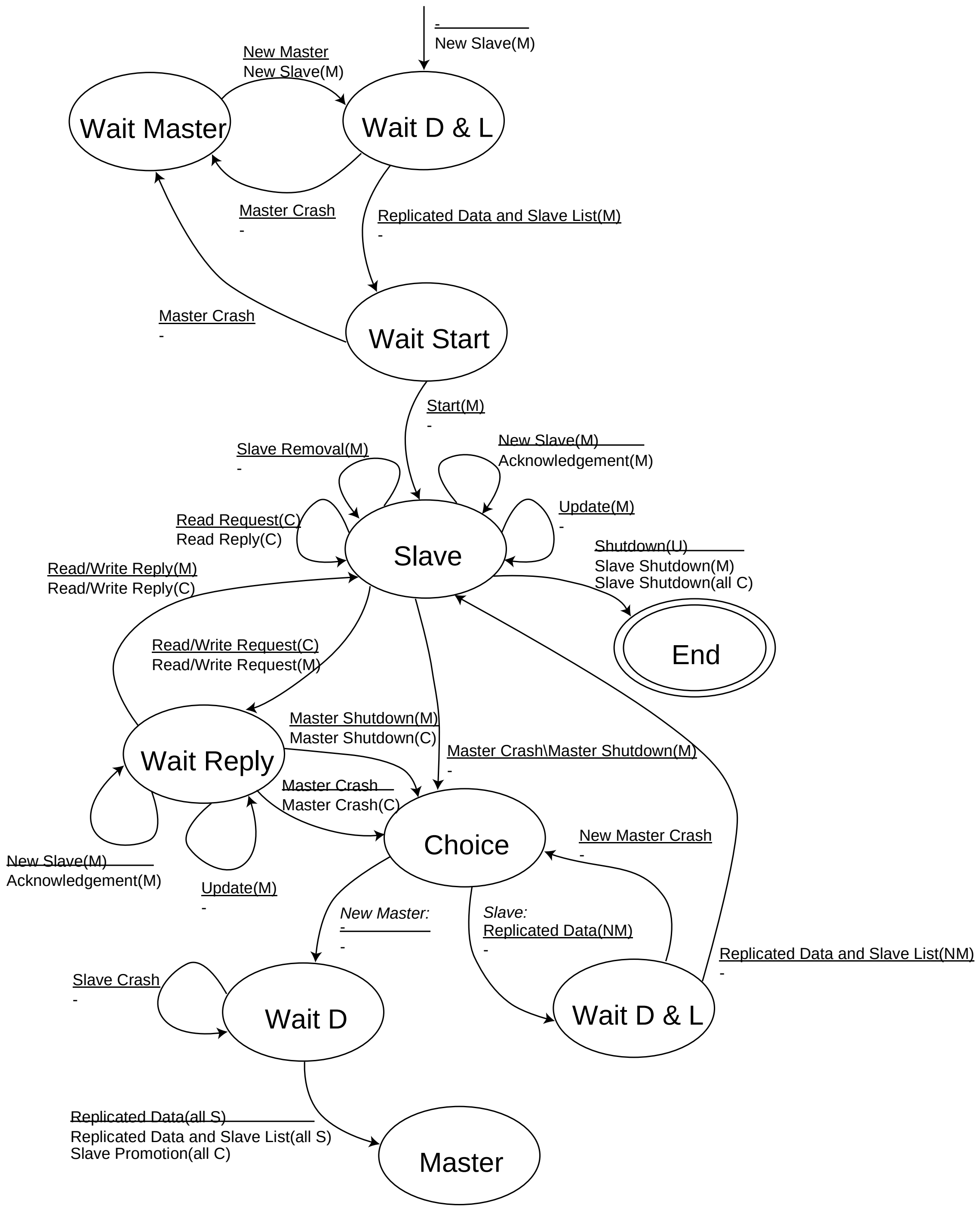

-
New Slave(M)
New Master
New Slave(M)
Wait Master
Wait D & L
Master Crash
-
Replicated Data and Slave List(M)
-
Master Crash
-
Wait Start
Start(M)
-
Slave Removal(M)
-
New Slave(M)
Acknowledgement(M)
Update(M)
-
Read Request(C)
Read Reply(C)
Slave
Shutdown(U)
Slave Shutdown(M)
Slave Shutdown(all C)
Read/Write Reply(M)
Read/Write Reply(C)
End
Read/Write Request(C)
Read/Write Request(M)
Master Shutdown(M)
Master Shutdown(C)
Master Crash\Master Shutdown(M)
-
Wait Reply
Master Crash
Master Crash(C)
Choice
New Master Crash
-
New Slave(M)
Acknowledgement(M)
Update(M)
-
New Master:
-
-
Slave:
Replicated Data(NM)
-
Replicated Data and Slave List(NM)
-
Slave Crash
-
Wait D
Wait D & L
Replicated Data(all S)
Replicated Data and Slave List(all S)
Slave Promotion(all C)
Master

Fig 3.9

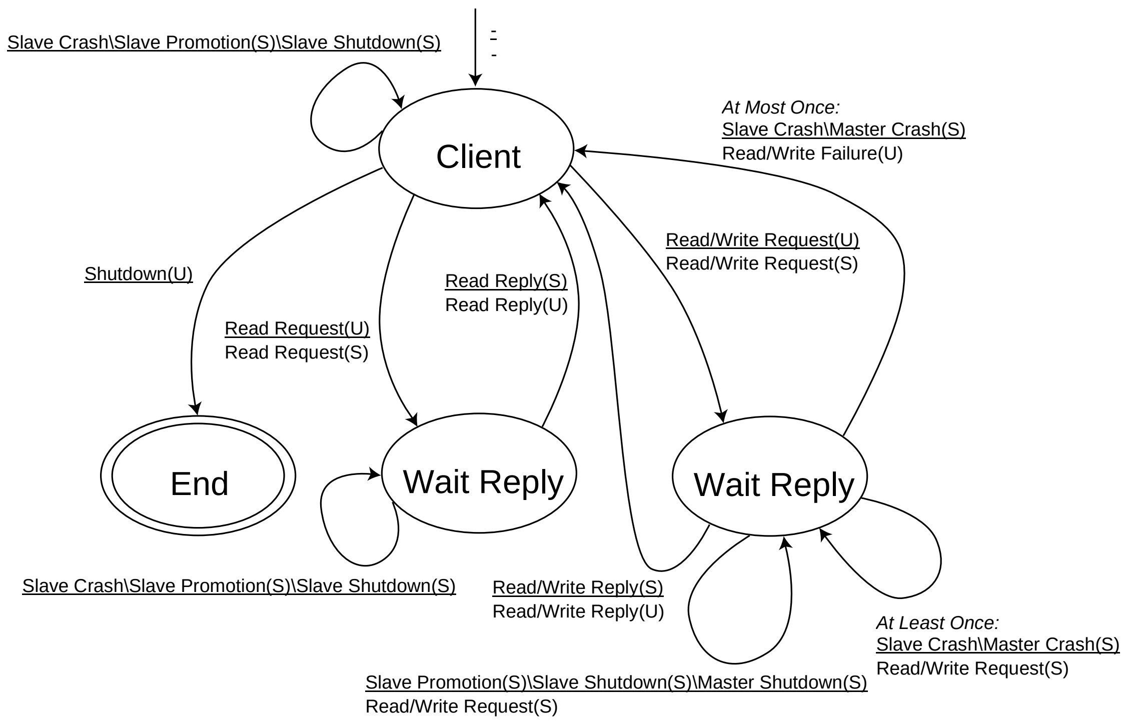


Fig 3.10

Slave Activation (Without Crashes During Activation)

In contrast to master and client activation, slave activation in the fault tolerant protocol varies from slave activation in the non-fault tolerant protocol. Slave activation is different because we want each slave to be capable of taking over the master's job. As previously stated, we do this by forcing each slave to be aware of the presence of other slaves. When a new slave is activated, this means that every slave must learn about the new slave. It also means that the newly activated slave must learn about all of the other slaves. In a data structure known as the slave list, both the master and the slaves save address information about the slaves they believe are active.

Slave activation starts with the new slave sending the master a New Slave message. This message contains information on the sender's address. The information is added to the slave list by the master when it gets the message. After that, the master sends the new slave a copy of the replicated data as well as a copy of its slave list. The slave list ensures that the new slave is aware of all other slaves. The master also informs the other slaves about the new slave by forwarding the New Slave message to them. In the Wait

D & L stage at the top of Figure 2.5, the new slave waits for replicated data and the slave list.

Slave Crash (No Process is Handling an Event)

Now we'll look at the first scenario, which involves a crash failure. To be more specific, the scenario concerns a slave's crash failure.

When a process detects a crash failure, it's likely that it'll need to take action. State transitions are depicted in the state diagrams. Slave Crash is always the text of the cause linked with such state transitions. The Slave Crash cause does not include any letters between parenthesis. Because the cause of the collision may be uncertain, the letters are missing. When a slave crashes, almost all processes must take some action. Client entities that send requests to other non-crashed slaves are the only ones who don't need to do so. The master must take steps to prevent the crashed slave from receiving updates and to remove the slave from its slave list. Slaves who did not crash are required to delete the crashed slave from their slave lists as well. Clients who are relying on the crashed slave to forward requests must switch to other slaves.

The procedure begins with the master detecting the crash in the Master state and the relevant clients detecting the crash in their client states, assuming no events occur during a slave crash. The wrecked slave gets removed from the master's slave list as a result of the crash. Assuming the master uses the slave list to determine who to update, deleting the slave ensures that the master no longer sends updates to the crashed slave. All non-crashed slaves receive a Slave Removal notification from the master. Slaves in the Slave state can also remove the crashed slave from their slave lists after receiving this notice. When a client detects a slave crash, it searches for a new slave to which it can send requests. The system determines how to locate another slave. When the system stores slave addresses in a directory service, querying the service may be sufficient. Of course, the addresses of crashed slaves should be removed from the directory service for the searching of slaves to work properly. When the master, for example, detects a slave crash, it can do so. A client may obtain the address of a crashed slave if the addresses of crashed slaves are not removed from the directory service.

There are two options for which messages to use currently. We can use messages that are currently required by the protocol, or we can create new ones. The inconsistency in the sending of protocol messages is a concern while employing them. The period between a crash and the discovery of the crash becomes depending on the frequency with which events occur since the processes transmit protocol messages in response to events. This isn't an issue with special messages. The messages can be sent at regular intervals by the processes. Processes, for example, can send I-am-alive signals every few seconds to indicate that they are still running and have not crashed.

Master Crash (No Process Is Handling an Event)

A crash of the master process is discussed as a second scenario including crash failures. There are no requests, slave activations, or shutdowns in progress in this situation. We described how each slave learns which other slaves are present in the previous examples. In the current scenario, the slaves use this information to take the master's place. In essence, all slaves must choose the same slave from their list of slaves, and that slave becomes the new master. The slaves, on the other hand, may not all have access to the same slave list. Minor differences are possible. Some slaves may have received messages such as New Slave and Slave Removal that alter their slave lists, while others may still need to do so.

To begin, assume that each slave is aware of which other slaves are active, and that the slave assuming the role of master does not crash during its transition. Each slave recognizes this in its Slave state when the current master crashes. We assumed there was no event processing going on, therefore a slave couldn't be in any other state. Each slave follows the transition to the Choice state in Figure 2.5 after detecting the master crash. In this condition, each slave selects who becomes the new master in a deterministic manner. After a slave has made a decision, the slave is removed from the slave list. The slave does this because he or she has been chosen to be the master, and a master does not engage in slave actions. There are two options after removing the slave from the slave list. To begin, if a slave wishes to be the new master, it must follow the transition marked by the italicized text New Master. Second, if the slave does not succeed in becoming the new master, it proceeds to the next possible transition from the Choice stage.

When a slave chooses another slave to be its new master, it transmits a copy of its replicated data to that slave. In the Wait D state of Figure 2.5, the new master waits for duplicated data from all slaves. The master computes the most recent version of the replicated data after receiving all copies of data. We want the new master to continue operating with the most recent version of the duplicated data, so it does this. The new master may not have this version because it detected the old master's crash before receiving all of the old master's upgrades. When the new master is finished computing, it distributes the computed data, as well as a copy of its slave list, to all slaves. In the current circumstance, the reason for the new master sending the slave list becomes obvious later. The slaves utilize the data and slave list to update their replicated data and slave list after receiving it. The slaves resume their customary routines after the update. The slave actions of the new owner are not continued. As a result, the new master notifies any clients that send it requests that it is no longer a slave. This is accomplished by the new master issuing a Slave Promotion message to all clients. Because the new master may not know which clients sent requests, a multicast channel can be helpful.

It's possible that a slave will crash when the new master is in the Wait D stage. If the slave performs this before providing replicated data, the new master will have to wait indefinitely for data from the slave. To avoid this, the new master must detect slave crashes while in the Wait D state. When the new master detects a slave crash, it can ensure that it no longer waits. Furthermore, the new master can delete the crashed slave from its slave list, ensuring that when the slave list is sent along with the most recent version of the duplicated data, the other slaves are aware that a slave has crashed. Although we thought that each slave knew which slaves were active up until now, this may not be the case. Certain slave crashes may have gone undetected, and some slave list modifying signals may have gone unnoticed. However, because the current scenario assumes only slave crashes and no slave activations, each slave must at the very least be aware of a subset of the active slaves.

Looking at the superset, we can see that a slave can choose a slave who is no longer active as the new master by accident. Because a non-existing new master cannot reply, the chosen slave transmits its copied data to a non-existing process and then waits for a response. We need the chosen slave to identify the non-existence to address this problem. This must be done while the slave is in the Wait D & L condition shown at the bottom of Figure 2.5. Because no messages are sent by non-existing processes, the slave can detect their absence as a crash. When the slave notices the crash, a transition back to the Choice state is possible. When the slave returns to this condition, it can choose another slave from its slave list. Because the slave eliminated the non-existent slave the last time it exited the Choice stage, it is unable to choose that slave again. In the current case, the final assumption we must discard is that the new master will not crash during its installation. If we abandon the assumption, it is possible that some slaves will receive the message containing the computed version of duplicated data while others will not. The crash seems to the slaves receiving it to be identical to a standard master crash, as they return to the slave state after getting the data. The other slaves, on the other hand, remain in the Wait D & L stage. A slave recognizes the absence of a new master in this state, but because detection is based on missing messages, the slave can also detect a new master's crash. When the slave detects a crash, it returns to the Choice state, allowing it to choose a new master. The current situation, like previous scenarios, necessitates changes to a directory service when one is used. For one thing, because the crashed master is no longer available, the address of the crashed master must be deleted from the service. In addition, because the new master is no longer a slave, the entry of the new master must be changed. When the new master detects a slave crash in the Wait D state, it is also required to remove the slave's address from the directory service. Obviously, the new master can use the directory service to accomplish all three activities.

Process Crash During Slave Activation (No Process Is Handling any other Event)

Now we'll look at a third situation that involves accidents. During slave activation, the master and slave both crash in this circumstance. The slave crashes are addressed first in the section below. After that, we'll take a look at master crashes. We assume no requests and shutdowns are in progress, like in the preceding instances. The need that each slave provide an acknowledgement to the master after receiving a New Slave message causes slave crashes during slave activation. When a slave crashes before

sending the acknowledgement or before receiving the New Slave message, the master is forced to wait indefinitely for the acknowledgement.
The master must identify slave crashes while in the Wait Ack state to avoid this problem. When the master tries to receive acknowledgements, he or she enters the Wait Ack state. If the master detects a slave crash, it can ensure that it no longer waits for the crashed slave's acknowledgement. The master can also delete the crashed slave from its slave list and send a Slave Removal message to all other slaves, including the newly activated one. The slaves will learn about the slave crash if the Slave Removal notification is sent. It's worth noting that this technique is remarkably similar to what the new master performs when receiving replicated data from slaves.

The situation becomes more serious when the master crashes during slave activation. It is possible for situations to arise in which some slaves are aware of a new slave while others are not. This can happen because not all slaves have yet received the New Slave notification. As a result, it's likely that some slave lists won't have a set of slaves that's a superset of the active slaves. When the new slave joins a new master instatement, it may be necessary for the new slave to communicate its replicated data to a process that is unaware of it. The process may have stopped waiting because it did not expect the data. We need the new slave to deactivate when it detects a master crash to fix this problem. When the new slave deactivates, every non-crash slave becomes a superset of the active slaves once more. The protocol employs the process with the acknowledgements and the Start message during slave activation, as we detailed in the slave activation scenario, to solve the problem of the new master not knowing about a new slave. When the master crashes, the new slave uses the method to determine when all other slaves are aware of it and when it no longer needs to deactivate. The deactivation of a newly activated slave is depicted as a state transition in the slave state diagram. The transitions between the Wait D&L state and the Wait Start state to the Wait Master state are deactivation transitions, to be exact. When a new slave reaches the Wait Master state, it waits for a master to reappear. After that, the new slave tries to repeat the activation operation. It accomplishes this by returning to the Wait D & L state. Of course, the question is how to tell when a new master is available. Using a directory service is one solution to this problem. If the master's address must be stored in the directory service, and a new master changes this address to its own after installation, a deactivated slave just needs to poll the directory service until the address changes.

Process Crash During Request Handling

Until now, all crash failure scenarios assumed that no read requests or read/write requests were being made. This assumption is not made in the current case. As a result, when a process crashes, clients and slaves may be in the Wait Reply state.
Let's start with what happens if a slave crashes while a read request is being processed. A problem arises in this situation when a client sends a request to a slave that crashes, and the slave crashes before returning a response to the client.

Obviously, the client will have to wait an eternity for a response. To avoid this, the client must be able to recognize when the slave for which it is waiting crashes. The client can then search for another slave and forward the read request to that slave if it detects a crash. This ensures that the client receives a response that it can pass on to the user. Because read requests are idempotent actions, the client can easily forward the read request many times. When a client submits a read request to the slave that becomes the new master after the master fails, a problem develops. Because the new master does not process read requests, the client is left waiting for a response that never comes. During the instatement, however, the new master sends a Slave Promotion message. As a result, if a client can get such a message while waiting for a response, we can avoid the never-ending wait. After receiving the message, the client might look for another slave to forward the read request to. Now it's time to read and write requests. Read/write requests, unlike read requests, do not need to be idempotent. To work around this, the master might keep track of the queries that lead to the current version of the replicated data. If a client sends a request that has already been executed, the master might simply ignore it. Of course, the master must respond, because a client will only resend a request if it has not received a response the first time. Returning a reply becomes possible once the master saves the reply associated with each executed request.

Associating a unique identifier with each read/write request can aid in remembering the requests that have been completed. 5 When the master saves the identifiers of all performed requests, it can compare the identifiers of newly received requests to those already saved. As a result, the master can pinpoint exactly which requests need to be completed. To make this fully functional, the identifiers must be stored alongside each copy of the replicated data. This ensures that, in the event of a master crash, the new master is aware of the requests that lead to the current version of the replicated data. When master crashes are possible, one downside of utilizing unique identifiers is that it necessitates delivering the identification and the update associated with each request at the same time. Regrettably, this is not always achievable, especially when queries may refer to peripherals. Consider a request for the usage of a printer, for example. There are two options available to the master in this situation. The master can either print before or after sending the update and the unique identifier. The slaves believe the master did not perform the request if the master prints before transmitting and crashes immediately after printing.

If, on the other hand, the master prints after transmitting but crashes just before printing, the slaves believe the master has completed the request when it has not. We don't use unique identifiers to tackle this problem. What we do is distinguish between read/write requests for which it doesn't matter if the master executes them several times and requests for which it does. At-least-once requests are those in which the outcome is irrelevant. At-most-once requests are those in which the outcome is critical.

Let's take a look at the issues that arise when a slave fails to handle a read/write request. A client may be waiting for a response from a crashed slave, just as in the case of a read request. As a result, the answer is to allow the client to identify when the slave for which it is waiting crashes. After detecting a crash, the client can look for another slave. What occurs next is determined by the type of request. If the request is an at-most-once request, the client returns to the Client state and delivers a failure to the user who made the request. The client does this because the request is an at-most-once request, and the request execution may have ended but only the response may have been lost. On the other hand, if the request is an at-least-once request, the client can send it again. In the case of an at-least-once request, it makes no difference how many times we execute it.

Two issues can arise when the master crashes while handling a read/write request. The first occurs when a slave is in the Wait Reply stage, waiting for the master to respond.

The slave does not receive a response when the master crashes, and it waits indefinitely. To avoid this, slaves must detect master crashes while in the Wait Reply state. When a slave detects a master crash, it might initiate the master selection procedure. The slave must also deal with the unanswered question. To keep the slave from becoming overly complicated, we only allow it to deliver a Master Crash message to the client who made the request. The client must be able to detect this message while waiting, and once it does, it must decide whether or not to relay the request again, depending on the request. Now we'll look at the second issue that can arise when the master crashes while processing a read/write request. When a slave detects a master crash, it does not immediately begin to process the read/write request. In this instance, the slave must choose a new master as well. When the slave becomes the new master, a dilemma arises. The client waits for a response from the slave, but since the slave is no longer a slave, it ignores the client's request. The slave, on the other hand, sends a Slave Promotion message. We can ensure that the waiting client does not wait indefinitely if we allow it to detect this message.

When the client receives the Slave Promotion message, it might look for another slave to send the request to. It can always forward the request since it only receives a Slave Promotion message from the slave that becomes master as the initial message, if the slave was not already handling the read/write request. The client receives a Master Crash message before the Slave Promotion message when the slave is handling the request. This is a result of the preceding problem's solution, as well as the assumption established in the previous section that every slave has only one FIFO connection to each of its clients.

3.3 Shutdowns

We've reached the final scenario, which is about shutdowns. The handling of shutdowns was predicated on the notion that shutdowns resemble crashes in that they both stop the generation of output. When a process terminates, we allow other processes to carry out operations that are identical to or similar to those carried out in the event of a crash. Shutdowns and crashes are clearly not the same thing. Shutdowns typically provide time to execute various activities, which is not the case in the event of a crash. We take advantage of this distinction by allowing processes to handle shutdowns only when they are not processing any other event.

These are the states with the names of the processes in the state diagrams. The processes are simplified by simply dealing with shutdowns in these states. They are not required to test for their shutdown in all potential states. Let’s begin by explaining shutdowns in terms of client shutdowns. Only users are affected in this scenario. The client is unknown to the master and slaves. A user initiates a client shutdown, as depicted in Figure 2.6. As a result, it's safe to infer that the user no longer requires the duplicated data to which the client offers access, and that the user doesn't need to launch a new client to which it can submit future requests. Despite the fact that this is true, other users may still require it. Despite this, other users may still require the client that is being shut down. These users will need to download and install a new client.

When a slave stops working, it sends a Slave Shutdown message to any clients who are sending requests to it, as well as to the master. Because the slave does not need to know who its clients are, sending the message to them may require a multicast channel. To avoid the crash detection method, the slave sends the Slave Shutdown message, which is likely to be slower than sending a message. In every state where slave breakdowns may be detected, both master and clients must be able to receive the Slave Shutdown message. The master reacts the same way to the Slave Shutdown notification as it does to a slave crash. It's the same thing, because a slave that has been turned off is no longer accessible to take over the master's job. The client's reaction isn't entirely consistent. The client, on the other hand, hunts for another slave. The difference is that when the client is waiting for a response during the shutdown, it always passes the request again. Even though the request is an at-most-once request, the client does so. It can do this because it knows the slave was in the Slave state when the Slave Shutdown message was sent.

The master's response to the Slave Shutdown message is identical to the slave's response to a crash. It's the same thing, because a turned-off slave can't assume the master's job. The client's response isn't entirely consistent. The customer, on the other hand, is looking for a new slave. When a request was waiting for a response during the

shutdown, the client always passes it again. When the request is an at-most-once request, the client performs the same. Because it knows the slave was in the Slave state when it issued the Slave Shutdown message, it can do this.
Let's have a look at what occurs when the master computer is turned off. The master sends a Master Shutdown message to all slaves in this situation, as this is likely faster than utilizing the crash detection system. In both their Slave and Wait Reply stages, the slaves must be able to detect the message. When the slaves discover the message, they must perform the procedure for picking a new master. A slave must also send a Master Shutdown message to the client that sent the associated request if it is waiting for a response. After that, the client can resend the request. Because the master did not handle the request before it was shut off, it can always do this, regardless of the type of request. Due to a single FIFO network connection, the slave receives a response before it receives the Master Shutdown message if the master handled the request. Of course, if the client forwards the request again, the slave to whom the request is forwarded may become the new master. This is not a problem because the client can detect the Slave Promotion message when it is sent as a response to an incoming request by the promoting slave.

It's possible that a process will crash during its shutdown operation. When this happens, either as a master or as a slave, the process may not have sent all shutdown notifications. Fortunately, this isn't a serious issue because, usually always, a process responds to a shutdown notification in the same way it does when it detects a crash. The behaviour of a client who waits for a response to an at-most-once read/write request is the lone exception. When the client receives a shutdown notification, it resends the request, which it does not do when the client detects a crash. We've now covered practically every scenario that could occur in the event of a shutdown. There are only two remaining extraordinary cases. The first of these scenarios occurs when the master goes down while a slave is being activated.

During activation, a slave sends a New Slave message, as stated above. If the master shuts down before receiving the New Slave message, the message is not handled. Because the new slave is not yet on the master's slave list, the master does not send a Master Shutdown message. In principle, the new slave will have to wait indefinitely. In actuality, however, it does not. This is due to the fact that the new slave detects master crashes and the master ceases to produce output after shutdown.

When the master crashes during a slave shutdown, the second exceptional shutdown situation happens. The owner does not or only partially send Slave Removal messages to other slaves in this situation. Fortunately, this isn't an issue. When a slave ceases working, it also stops producing output. When a new master does not receive the Slave Removal message, it can identify the shutdown using its crash detection method.

3.4 Protocol Optimisations

Following our discussion of the fault-tolerant protocol, we'll look at three changes that can help you enhance your efficiency. Multithreading within processes is the first adaptation. The second adaption is to provide only differences as updates. After a master crash, the third adaptation is to stop delivering replicated data to the new master. We didn't mention these changes in the prior section's protocol since they would have detracted from the protocol's main point.

3.4.1 Multithreading

The usage of multithreading within processes is the first adaption that improves efficiency. Until now, we didn't employ multithreading because each process could only accomplish one thing at a time. Regrettably, this is quite restricted. A procedure can usually accomplish numerous things without breaching the protocol. Consider the read requests that a slave executes. The data is not changed by read queries. As a result, a slave may easily handle numerous read requests at once. As

1. Provide the new master with the version number (all S)
2. Determine the most current version number (NM)
3. Request data from a slave with the most recent version number (NM)
4. Hold your breath and wait for the data to arrive.
5. Distribute the data to all slaves who supplied an older version number (NM)

Consider a slave who is waiting for a response to a read/write request from a certain user. While waiting, this slave can easily handle several read requests from other users. The previous examples have one thing in common: they may both be implemented using threads. Adding threads can enhance performance because each example allows several events to be processed at the same time.

Differences as Updates

We treat the updates supplied by the master as the second adaption. We haven't given a precise definition of what an update is until now. However, it's easiest to think of an update as a transfer of all replicated data. Regrettably, the modifications to the data may be minor. This makes sending all data inefficient. When the variations are minor in comparison to the quantity of the replicated data, it is preferable to let the master send them. To put it another way, the master must only communicate the replicated data that has changed.

Not Sending the Replicated Data

Now we'll look at the third efficiency-enhancing adaption. After a master crash, this adaption prohibits each slave from transferring its copied data to the new master. Of course, this is impossible to completely avoid. We still want the new master to get the most recent version of the replicated data, as well as every slave. Allowing each slave to communicate its data to the new master, on the other hand, is meaningless. Only one copy of the most recent version of the duplicated data is required by the new master. There are no ancient versions or more than one most recent version required by the new master. We can use version numbering to ensure that only one copy is sent.

We may utilize the protocol in Figure 2.7 to limit the sending of duplicated data if we include a version number in the replicated data and increment the number after each performed read/write request.

When the new master attempts to receive duplicated data from one of the slaves, it must be able to recognize a slave crash. When a crash happens, the new master does not have to wait indefinitely. The new master can try to obtain the data from another slave once a slave crashes.

The procedure in Figure 2.7 has the added benefit of not sending the most recent version of the data to slaves that already have it. This, too, boosts productivity.

3.5 Assessment of the Master-Slave Protocol

3.5.1 Validity of the Protocol

We discuss two topics in terms of the fault tolerant protocol's correctness. The protocol's liveness is the first item we'll look at. This implies we check to see if the protocol is free of deadlocks and starvation, even in the event of a crash. The protocol's functional soundness is discussed as a second issue. This topic necessitates the definition of a sensible semantics to which queries operating replicated data must follow, as well as the verification that the requests do so.

3.5.2 The Protocol's Viability

Even in the presence of crash failures, liveness necessitates the absence of deadlocks and starvation, as mentioned above. As a result, in order to test the fault tolerant protocol's liveness, we must provide a formal demonstration that the protocol is free of deadlocks and starvation. The scope of this thesis does not include a formal proof verifying the lack of deadlocks and starvation regardless of the number of slaves, clients, or users. Nonetheless, we didn't want to fully dismiss the question of liveness. As a result, we double-checked the liveness for situations with three or less slaves.

Spin, a protocol verification tool, assisted us in verifying the liveness of our protocol. 6 Spin 3.4.1 was utilized in this experiment. In Spin, verification begins with the creation of a finite automata for each process. The Cartesian product of all automatons is then computed, and each attainable state in the product is checked for liveness. Spin locates states by doing a depth first search that begins at a predetermined start state. Spin requires process definitions given in a programming language called Promela, which is a basic imperative language, to build the finite automatons.

The algorithms in tools like Spin work by recording every state that can be reached. Due to the vast number of states that processes can have, the tools may require a lot of memory. Spin, in our situation, required more RAM than was available at the time. To address this, we decided to group all slaves along with the users and clients who send requests to them.

The number of states, and hence the amount of memory required, is reduced by combining slaves with clients and users. The restricting effect stems from our decision to include only actions involving the master process, rather than those involving slaves, clients, or users. This means that huge portions of the slaves, clients, and users did not need to be implemented.

When clients and users are combined with slaves, the clients and users crash when the slave crashes or when the slave promotes to master. Separating the clients and users from the slaves is the only way to overcome this problem. Regrettably, this results in additional states. It was feasible to check the protocol's liveness by pairing slaves with clients and users. Spin did not run out of memory, despite the fact that it still required 500 MB. There were no deadlocks or starvation discovered during the liveness check. When there are three slaves or fewer, and when the slaves are combined with clients and users, we can conclude that our protocol meets the liveness property.

3.5.3 The Protocol's Functional Correctness

As previously stated, determining functional correctness necessitates the establishment of semantics to which requests using replicated data must follow. We concluded that each request must meet two criteria. First, when a user submits a request, the time it takes for the user to receive a response must be limited. Second, if a user sends a read/write request and then sends another, the second request must operate on a version of the replicated data that contains the read/write request's changes. This second circumstance is known as the See-Your-Writes condition.

3.5.4 Finite Amount of Time

Let's start with the condition of a finite amount of time. What we can see is that when there are no crashes and the actual execution of a request takes only a fixed amount of time, a user must receive a response within that time frame. The explanation for this is straightforward. Because the network is synchronous, all messages are delivered in a set length of time. As a result, the network cannot make a user wait indefinitely for a response. Furthermore, it is acceptable to assume that all processes handle messages in the order in which they are received. As a result, no message received by a process has to wait indefinitely for it to be processed. As a result, a user does not have to wait indefinitely because the processing of a request takes a finite period of time.

When the master or slave crashes, we can see that a user still receives a response within a defined length of time. The reason for this is that when a client crashes, it re-forwards read and at-least-once read/write requests, as well as returning failures as answers in the case of at-most-once read/write requests. The user does not have to wait indefinitely if a client crashes. In the event of a shutdown, the same logic that applies to crashes applies. As a result, when a shutdown occurs, a user receives a response in a finite length of time.

3.5.5 Up-To-Date Replicated Data

When we examine the See-Your-Writes condition, we can see that it is identical to one of the criteria from section 2.2.1, which restricts the updates the master delivers. We may meet the criterion by allowing a user to always use the same client, a client to always use the same slave, and only permitting a single connection between each pair of processes. As a result, the fault tolerant protocol meets the See-Your-Writes requirement as long as no crash occurs.

In the event of a crash, the question is whether the protocol also meets the See-Your-Writes requirement. Unfortunately, this does not always turn out to be the case. When a slave responds to a read/write request and then crashes, there is an issue. If the client receiving the response sends another request to the user who issued the read/write request, the request must be forwarded to a different slave. However, because the protocol does not ensure that the other slave has received and implemented all necessary updates, it is possible that the request will not be executed with the correct version of the duplicated data. To make matters worse, if the master crashes as well, the necessary changes may be completely lost. Other than the wrecked slave, no other slave may have received the updates. The only way to fix the concerns described above is for the slaves to acknowledge receipt of an update before the master delivers the accompanying response. Regrettably, this is ineffective.

3.5.6 The Protocol's Efficiency

Efficiency is a concept that is always relative. As a result, we compare our fault tolerant protocol to a non-fault tolerant protocol to determine its efficiency. We compare only the operation in a crash-free environment because the non-fault tolerant protocol does not work when crashes occur. When we compare the protocols, we can observe that practically all of the operations they share are the same. As a result, the efficacy of both regimens must be almost comparable. The sole variation between the protocols is how new slaves are handled. In the case of the fault tolerant protocol, this operation is more difficult. Activation of new slaves, on the other hand, is unlikely to occur frequently because activations are unrelated to activities on duplicated data. As a result, it should have no impact on efficiency. The protocol's handling of read/write requests is complicated by acknowledgements. As a result, handling read/write requests becomes less efficient, and because read/write requests operate on duplicated data, rather than new slaves, they are more likely to occur often than slave activations.

3.5.7 Using the Protocol in Timed Asynchronous Networks

Until recently, we've believed that networks transmit all communications in a predetermined length of time. These so-called synchronous networks are, unfortunately, uncommon. Most networks are timed asynchronous, which implies that they send the majority of messages in a set length of time but not all of them.

We must alter our fault-tolerant master-slave protocol in order for it to work in a timed asynchronous network. We can't merely use the protocol specified because it implies a crash if a message isn't received. A crash did not always happen when we used a timed asynchronous network. It's possible that the network is simply delaying the message.

We did not attempt to modify the master-slave protocol for timed asynchronous networks, despite the fact that it is certainly possible. Adapting a protocol will almost certainly be difficult. Take, for example, Cristian's membership protocol. The protocol grew much more complicated when Cristian and Schmuck developed it for a timed asynchronous network. Of course, we don't want to dismiss the topic of timed asynchronous networks entirely. As a result, we'll proceed to a brief examination of the issues at hand. Furthermore, we provide a method that does not necessitate any changes to our fault-tolerant protocol. The solution, however, places significant demands on the underlying distributed system.

3.5.8 Analysis of the Problem

As previously stated, a message arriving late does not always imply that a procedure has failed. For different sorts of processes, this discovery has varied implications.
When we examine our clients, we can see that there are no actual consequences. When a client notices that a slave's message is missing, it simply searches for another slave. After finding a slave, the client can resume normal operations. The master and slaves are unaffected by the client moving to another slave. They are completely unaware of the client's existence. Of course, a user may be impacted by the transition because the client may have been processing a request when it discovered the missing message. A client not only communicates with slaves, but also with users. Because a client is unconcerned about whether or not a user is present, it is equally unconcerned about missing a communication from that user. When a user misses a message from a client, the situation is similar to when a client misses a message from a slave. The reason for this is that the protocol expects a user to behave similarly to a client in the event of a crash.

Let us now focus our attention to the master. It's worth noting that the master only communicates with slaves. As a result, a missing communication can only be interpreted by the master as a slave crash. Again, this isn't a serious issue. The master simply does one thing: it deletes the slave from its slave list and sends a Slave Removal message to all other slaves. As a result, the slave from which the message is absent is no longer a member of the slave group. The slave, on the other hand, can easily get around this by using the slave activation protocol. Of course, the slave must be aware when it is no longer a member of the slave group. This can be accomplished by allowing the master to send the slave a removal notification message. Obviously, the notification message has the potential to be misplaced. This leads us to the slaves' penalties of missing messages.

A missing message observed by a slave, in contrast to a missing message noticed by a master or client, can have catastrophic repercussions. It is possible for a split-brain to arise, in which many masters emerge at the same moment. 3 This is a bad situation because the master is in charge of ordering updates. The ordering can be messed up if there are multiple masters.

Consider an error that momentarily splits the network as an illustration of how a split-brain can occur. The error can divide a group of slaves from the master and other slaves. As a result, the separated slaves believe the master and other slaves have crashed. To get around this, the slaves use the master selection technique. This results in the creation of a new master within the partition containing the separated slaves. Furthermore, the master that was already running before to the partition problem continues to run. The master only thinks the divided slaves have crashed because of the partition problem. As a result, we now have two masters. Other factors can contribute to a split-brain condition. Despite the fact that we do not discuss any other causes here, it should be evident that missing messages in the master-slave communication are at the root of the problem. Slaves, in addition to communicating with the master, also communicate with clients. Due to the communication method used by the clients and slaves, this exchange cannot cause any issues. A client submits a request to a slave initially in this pattern. After that, the slave receives the request and processes it. Finally, the slave sends the client a response. Clients send requests at random times, so slaves have no idea when they will arrive. As a result, slaves have no way of knowing when a client message is absent.

3.5.8.1 Solution

The adoption of a membership service is one solution to the split-brain outlined above. Our fault tolerant protocol can use this service to register the master and slaves when the underlying distributed system provides it. Obviously, a membership service must be able to function successfully in a timed asynchronous network in order to be valuable. There are at least two systems that provide a membership service. Isis9 and Transis.10 are the two. Isis provides a virtual synchrony-based membership service. By enabling just the processes in one partition of a partitioned network to continue, virtual synchrony prevents split-brains. By utilizing extended virtual synchrony, Transis provides a membership service. Split-brains are not excluded by extended virtual synchrony, but the masters are merged once the network is no longer partitioned. When there is a problem between the masters, the protocol may need to step in to help.

4. IMPLEMENTATION

Few set of steps of the implementations of some particular configurations:

4.1 Creating EC2 Instances in AWS for Master and Slave

1. Go to EC2 and create 2 instances for master and slave
2. In rules of master and slave, add

a. HTTP

b. HTTPS

c. SSH : Secure Shell is a network communication protocol that enables two computers to communicate

d. All ICMP : Internet Control Message Protocol (ICMP) is used to send control messages to network devices and hosts.(Only for slave and add IP Address of master in the specified IP Address column of All ICMP)

e. MySql

f. Custom tcp rule

3. Install PUTTY

a. Go to putty gen

b. Load key pair

c. Generate ppk file

d. Go to putty

e. In Hostname give, ubuntu@18.208.114.45 (Master/Slave IP). IP would be automatically extracted.

f. Go inside SSH, then Auth and upload the corresponding ppk file

g. Your PUTTY command prompt is loaded

h. Do this two times for master and slave

i. Create a file using touch named master/slave just to know which is which cmd. (To verify if master abd slave are connected. Type
ping ip_address_of slave
to see if packets are flowing and vice versa)

4. Go to Master cmd,

a) sudo get-apt update

b) sudo get-apt mysql-server mysql-client

c) After installation is done, check if status is active

Sudo service mysql status

d) Sudo nano /etc/mysql/mysql.conf.d/mysqld.cnf

e) Change bind address to 0.0.0.

The bind-address configuration within MySQL tells MySQL on which networks it can listen for connections.

In IPv4-based routing, 0.0. 0.0 serves as a default route. This means no particular address has been designated in the routing table as the next hop in the packet's path to its final destination.

f) Uncomment serverid=1 and also the log_bin and save using ctrl+o

g) Sudo service mysql restart

To restart for effective changes to take place in the configuration file.

h) Type

show databases

To see all the databases currently created

i) Type

create user ‘replica’@’172.31.44.149’ identified by ‘root’;

(@ in this helps the system to distinguish the username and its corresponding private ipv4 address and ‘root’ is the password

The MySQL user is a record in the USER table of the MySQL server that contains the login information, account privileges, and the host information for MySQL account. It is essential to create a user in MySQL for accessing and managing the databases.)

j) Type

alter user ‘replica’@’172.31.44.149’ identified with sql_native_password by ‘root’;

alter will check for the corresponding user in the table and gives access to that particular user.

k) grant replication slave on *.* to ‘replica’@’172.31.44.149;

. means all privileges.

l) flush privileges;

To make into effect changes made to privileges of the user.

All that has to be done in master is done!

Now go to the cmd of slave;

1) install and sudo into mysql
2) edit config file

change bind address to masters private ipv4 address
and server id to 2 to make it unique.

3) Restart mysql
4) Sudo into mysql
5) CHANGE REPLICATION SOURCE TO SOURCE_HOST='34.207.67.115', SOURCE_USER='replica', SOURCE_PASSWORD='password', SOURCE_LOG_FILE='mysql-bin.000011', SOURCE_LOG_POS=2381;

(To find the details of the content of this line)GO to master cmd and
Type
show master status;
in the sql in the master cmd.

6) show slave status;
7) show slave status\G;

(Check if

a)This error is there,

https://serverfault.com/questions/872911/slave-sql-running-no-mysql-replication-stopped-working
Run this in slave and check the status again.

```
STOP SLAVE;
SET GLOBAL SQL_SLAVE_SKIP_COUNTER = 1;
START SLAVE;
```

if you have manually created the user on your slave first and on your master second, the CREATE USER command executed on the master has been replicated to the slave. Attempted subsequent execution of the statement failed, because the user was already present. So my suggestions to fix "this":

Tell the replication engine to skip the statement and move on
Do not mess with slaves in a replication setup in the future)

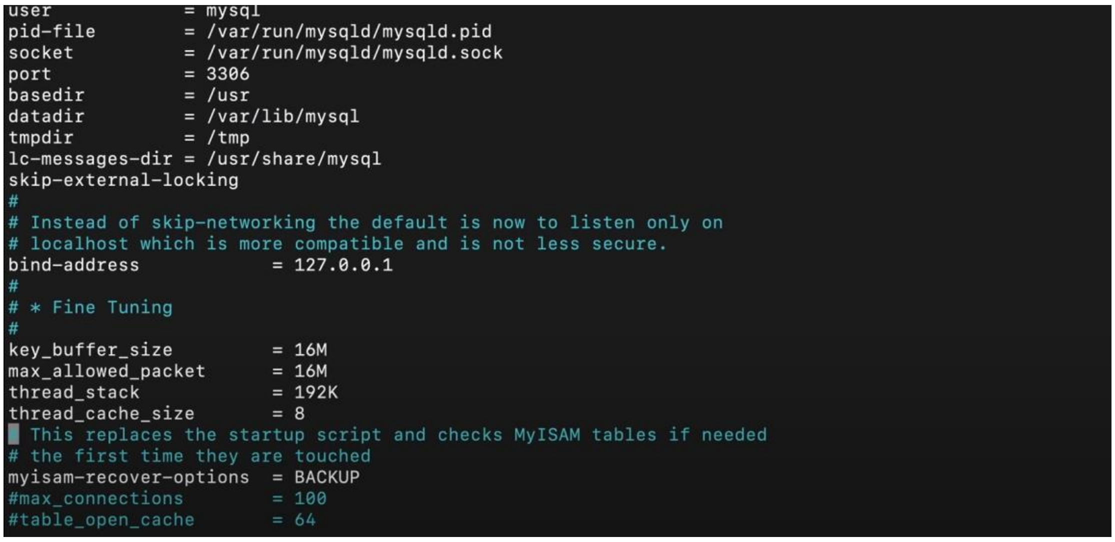
8) After errors are solved and slave is completely connected,
Go to the master cmd
type
use database_name
database_name being whatever is the name of ur db in master cmd.
Create table and Insert values into it
Select * into table and see if values are present.

Go to slave
Select * into table and see
(Simply stop and start slave if needed)
9) Connection is properly done if changes in master are visible in slave.

Common Error:
By any chance if you forget to write the alter statement in master
ERROR Authentication plugin 'caching_sha2_password' cannot be loaded
Both of these are running. If not it means you forgot to add the alter statement in master.
The error being Authentication plugin 'caching_sha2_password' cannot be loaded

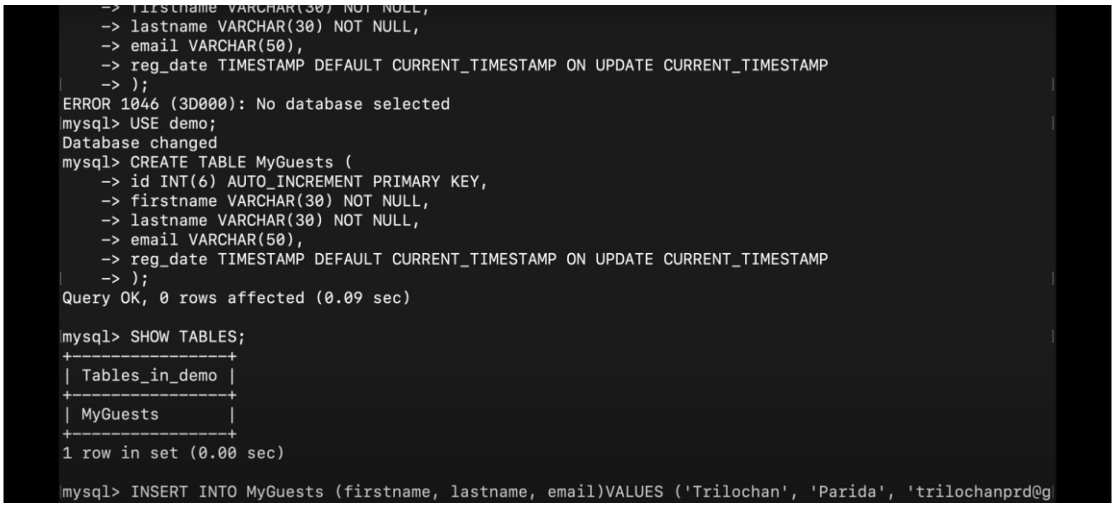
4.2 SOME OUTPUT SCREENSHOTS

- **This is the configuration file screenshot, which has all the details and location of all important files. Here we are setting the default bind address to 0.0.0.0 which means that that particular server can be get connected to by any server.**



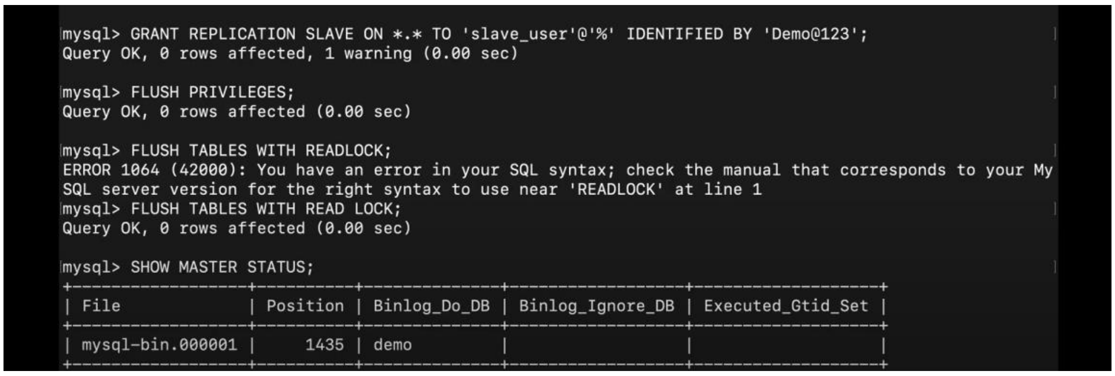
```
user            = mysql
pid-file        = /var/run/mysqld/mysqld.pid
socket          = /var/run/mysqld/mysqld.sock
port            = 3306
basedir         = /usr
datadir         = /var/lib/mysql
tmpdir          = /tmp
lc-messages-dir = /usr/share/mysql
skip-external-locking
#
# Instead of skip-networking the default is now to listen only on
# localhost which is more compatible and is not less secure.
bind-address            = 127.0.0.1
#
# * Fine Tuning
#
key_buffer_size         = 16M
max_allowed_packet      = 16M
thread_stack            = 192K
thread_cache_size       = 8
# This replaces the startup script and checks MyISAM tables if needed
# the first time they are touched
myisam-recover-options  = BACKUP
#max_connections        = 100
#table_open_cache       = 64
```

- **A sample table is created in the master and is being verified and displayed. It is to see if the corresponding table in the master is reflecting in the slave.**



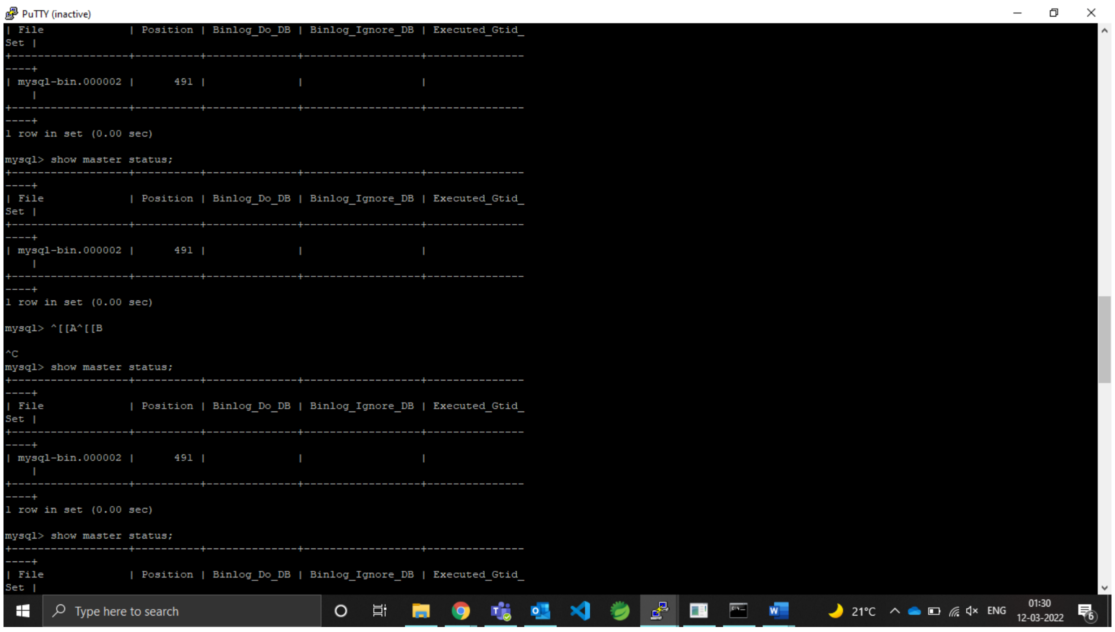
```
    -> firstname VARCHAR(30) NOT NULL,
    -> lastname VARCHAR(30) NOT NULL,
    -> email VARCHAR(50),
    -> reg_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
    -> );
ERROR 1046 (3D000): No database selected
mysql> USE demo;
Database changed
mysql> CREATE TABLE MyGuests (
    -> id INT(6) AUTO_INCREMENT PRIMARY KEY,
    -> firstname VARCHAR(30) NOT NULL,
    -> lastname VARCHAR(30) NOT NULL,
    -> email VARCHAR(50),
    -> reg_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
    -> );
Query OK, 0 rows affected (0.09 sec)

mysql> SHOW TABLES;
+----------------+
| Tables_in_demo |
+----------------+
| MyGuests       |
+----------------+
1 row in set (0.00 sec)

mysql> INSERT INTO MyGuests (firstname, lastname, email)VALUES ('Trilochan', 'Parida', 'trilochanprd@g
```

- **This screenshots show the granting of access to master for slave. *.* means complete access and flushing the privileges to reset the all the changes that are being made.**



```
mysql> GRANT REPLICATION SLAVE ON *.* TO 'slave_user'@'%' IDENTIFIED BY 'Demo@123';
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> FLUSH PRIVILEGES;
Query OK, 0 rows affected (0.00 sec)

mysql> FLUSH TABLES WITH READLOCK;
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your My
SQL server version for the right syntax to use near 'READLOCK' at line 1
mysql> FLUSH TABLES WITH READ LOCK;
Query OK, 0 rows affected (0.00 sec)

mysql> SHOW MASTER STATUS;
+------------------+----------+--------------+------------------+-------------------+
| File             | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+------------------+----------+--------------+------------------+-------------------+
| mysql-bin.000001 |     1435 | demo         |                  |                   |
+------------------+----------+--------------+------------------+-------------------+
```

- **All the bin log files and there corresponding positions are being shown. This position helps in the connection statement of master and a slave.**

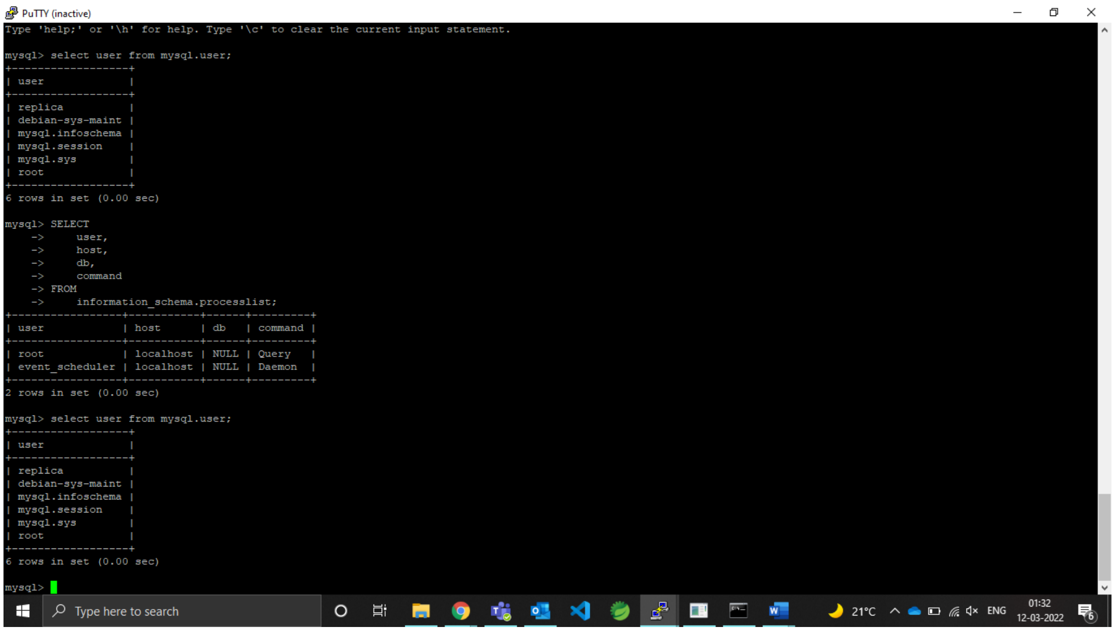


```
| File             | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_
Set |
+------------------+----------+--------------+------------------+---------------
----+
| mysql-bin.000002 |      491 |              |                  |
    |
+------------------+----------+--------------+------------------+---------------
----+
1 row in set (0.00 sec)

mysql> show master status;
+------------------+----------+--------------+------------------+---------------
----+
| File             | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_
Set |
+------------------+----------+--------------+------------------+---------------
----+
| mysql-bin.000002 |      491 |              |                  |
    |
+------------------+----------+--------------+------------------+---------------
----+
1 row in set (0.00 sec)

mysql> ^[[A^[[B

^C
mysql> show master status;
+------------------+----------+--------------+------------------+---------------
----+
| File             | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_
Set |
+------------------+----------+--------------+------------------+---------------
----+
| mysql-bin.000002 |      491 |              |                  |
    |
+------------------+----------+--------------+------------------+---------------
----+
1 row in set (0.00 sec)

mysql> show master status;
+------------------+----------+--------------+------------------+---------------
----+
| File             | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_
Set |
```

- **This particular code links master and slave using the bin log address**

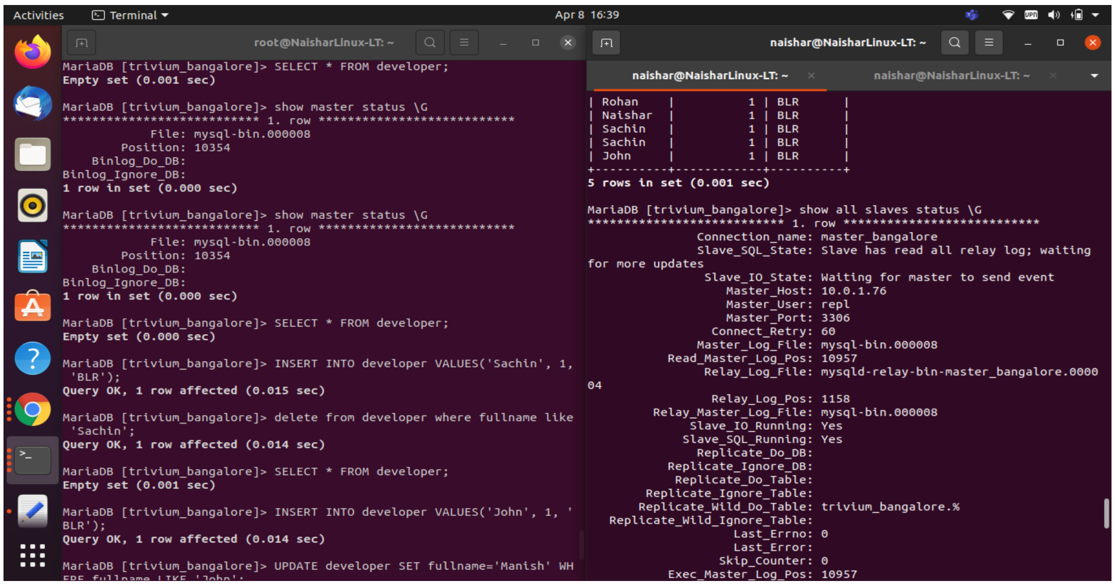
CHANGE REPLICATION SOURCE TO SOURCE_HOST='34.207.67.115', SOURCE_USER='replica',SOURCE_PASSWORD='password', SOURCE_LOG_FILE='mysql-bin.000011', SOURCE_LOG_POS=2381;

- **Seeing all the corresponding tables in the master are here**



```
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> select user from mysql.user;
+------------------+
| user             |
+------------------+
| replica          |
| debian-sys-maint |
| mysql.infoschema |
| mysql.session    |
| mysql.sys        |
| root             |
+------------------+
6 rows in set (0.00 sec)

mysql> SELECT
    ->     user,
    ->     host,
    ->     db,
    ->     command
    -> FROM
    ->     information_schema.processlist;
+-----------------+-----------+------+---------+
| user            | host      | db   | command |
+-----------------+-----------+------+---------+
| root            | localhost | NULL | Query   |
| event_scheduler | localhost | NULL | Daemon  |
+-----------------+-----------+------+---------+
2 rows in set (0.00 sec)

mysql> select user from mysql.user;
+------------------+
| user             |
+------------------+
| replica          |
| debian-sys-maint |
| mysql.infoschema |
| mysql.session    |
| mysql.sys        |
| root             |
+------------------+
6 rows in set (0.00 sec)

mysql>
```

- **As part of multisource replication, two tables of bangalore branch and munich branch of trivium esolutions are made and linked. These two tables are masters and can be accessed by a slave with read lock.**


Activities
Terminal
Apr 8 16:39
root@NaisharLinux-LT: ~
naishar@NaisharLinux-LT: ~
naishar@NaisharLinux-LT: ~
naishar@NaisharLinux-LT: ~

```
MariaDB [trivium_bangalore]> SELECT * FROM developer;
Empty set (0.001 sec)

MariaDB [trivium_bangalore]> show master status \G
*************************** 1. row ***************************
            File: mysql-bin.000008
        Position: 10354
    Binlog_Do_DB:
Binlog_Ignore_DB:
1 row in set (0.000 sec)

MariaDB [trivium_bangalore]> show master status \G
*************************** 1. row ***************************
            File: mysql-bin.000008
        Position: 10354
    Binlog_Do_DB:
Binlog_Ignore_DB:
1 row in set (0.000 sec)

MariaDB [trivium_bangalore]> SELECT * FROM developer;
Empty set (0.000 sec)

MariaDB [trivium_bangalore]> INSERT INTO developer VALUES('Sachin', 1,
 'BLR');
Query OK, 1 row affected (0.015 sec)

MariaDB [trivium_bangalore]> delete from developer where fullname like
 'Sachin';
Query OK, 1 row affected (0.014 sec)

MariaDB [trivium_bangalore]> SELECT * FROM developer;
Empty set (0.001 sec)

MariaDB [trivium_bangalore]> INSERT INTO developer VALUES('John', 1, '
BLR');
Query OK, 1 row affected (0.014 sec)

MariaDB [trivium_bangalore]> UPDATE developer SET fullname='Manish' WH
```

```
| Rohan    |           1 | BLR      |
| Naishar  |           1 | BLR      |
| Sachin   |           1 | BLR      |
| Sachin   |           1 | BLR      |
| John     |           1 | BLR      |
+----------+-------------+----------+
5 rows in set (0.001 sec)

MariaDB [trivium_bangalore]> show all slaves status \G
*************************** 1. row ***************************
               Connection_name: master_bangalore
               Slave_SQL_State: Slave has read all relay log; waiting
for more updates
                Slave_IO_State: Waiting for master to send event
                   Master_Host: 10.0.1.76
                   Master_User: repl
                   Master_Port: 3306
                 Connect_Retry: 60
               Master_Log_File: mysql-bin.000008
           Read_Master_Log_Pos: 10957
                Relay_Log_File: mysqld-relay-bin-master_bangalore.0000
04
                 Relay_Log_Pos: 1158
         Relay_Master_Log_File: mysql-bin.000008
              Slave_IO_Running: Yes
             Slave_SQL_Running: Yes
               Replicate_Do_DB:
           Replicate_Ignore_DB:
            Replicate_Do_Table:
        Replicate_Ignore_Table:
       Replicate_Wild_Do_Table: trivium_bangalore.%
   Replicate_Wild_Ignore_Table:
                    Last_Errno: 0
                    Last_Error:
                  Skip_Counter: 0
           Exec_Master_Log_Pos: 10957
```