UNIVERSITY OF CALIFORNIA
SANTA CRUZ

REPLICA COORDINATIVE LIFETIME ENHANCEMENT OF FLASH STORAGE IN DISTRIBUTED SYSTEMS

A project submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Narendra Kumar Govinda Raju

March 2017

The Project of Narendra Kumar Govinda Raju
is approved:

Professor Jishen Zhao, Chair

Professor Jose Renau

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Abstract

Replica Coordinative Lifetime Enhancement of Flash Storage in Distributed Systems

by

Narendra Kumar Govinda Raju

NAND Flash memories are replacing the traditional storage systems like the magnetic and optical mediums, due to their better performance during read and write. However, the major drawback with the flash systems is that they have a very low endurance. The lifetime of the flash memory is less compared to the magnetic medium. It is limited due to the fact that they wear out after few thousands of program-erase (PE) cycles. The magnetic or optical drives support 10^{12} - 10^{14} write cycles whereas flash system support just 10^{6} - 10^{7} write cycles. Hence the flash systems are not suitable for write intensive applications.

In this paper, we propose a new solution to reduce the number of writes in flash systems which increases the endurance, thereby increasing their lifetime. On the event of a write, the immediate writes on few of the replicas is delayed and batched at a later stage as a lazy update or through logging.

This solution can be implemented for a direct attached storage system having mirrored data using RAID technology or it can be implemented on a disaster recovery system having the mirrored data across multiple sites. For simplicity and ease of implementation for this project we are creating a simple key-value data store and replicating it across multiple sites

to mimic disaster recovery solution.

This solution can be deployed on a weak consistent system running write intensive applications. Also in distributed systems, all applications do not have to run as strongly consistent always. Depending on the needs whenever the application does not need strong consistency, our solution could be enabled to increase the system‘s endurance. This makes the flash systems usable even for work intensive applications as their endurance is increased.

Acknowledgments

First of all, I would like to express my sincere thanks to Professor Jishen Zhao for letting me work on this project and for all her support and ideas throughout the project.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Flash systems are gaining their presence everywhere from mobile devices to servers with RAID and SAN architectures. SSD has become more popular due to their high speed, low noise, low power consumption and reliability.

There are many reasons for flash storage picking up in the storage industry. ***The main advantages of NAND flash are:***

1. Better performance capabilities, speed in particular. Flash storage can get the system up and running in few seconds. Enterprises that need fast processing of their business applications and retrieve/store data quickly prefer flash storage systems.

2. The durability is very high compared to hard drives which have mechanical parts like the spinning disks, head etc. The chances of losing data due to equipment mishandling is low. This feature is very important for the business who are more concerned about the security of their data. The absence of mechanical moving parts also contributes to higher performance.

3. They consume very less power, thereby reducing the energy costs greatly.

There are a few disadvantages of the flash system as well and **the main disadvantage which is of interest to us is the endurance.**

Endurance is just a fancy name for ***life time*** of the storage systems. Endurance of a storage system is a very important aspect as it directly correlates to the lifetime of data. Compared to the hard drives, the NAND flash has a very low endurance. They have a finite program-erase cycles because of the process involved in the program erase operations for every write. ***NAND flash uses two methodologies to write data:***

1. Quantum Tunneling.
2. Hot Electron Injection.

Each write to these systems causes physical damage due to the above mechanisms. Refer figure 1.1 for quantum tunneling and refer figure 1.2 for hot electron injection method. The damages are due to the high heat generated. The oxide layer in the flash systems which are used to trap the charges is degraded every time a write is performed. The charge stored representing either a 0 or 1 cannot be differentiated due to the damage and hence the flash systems become unusable after this point. The damage eventually piles up to decrease the endurance time of the flash system. The other disadvantage of the flash systems compared to the hard drives is the cost per GB. NAND flash is much more expensive than the hard drives. The price of flash storage is gradually decreasing but currently SSD are more expensive.

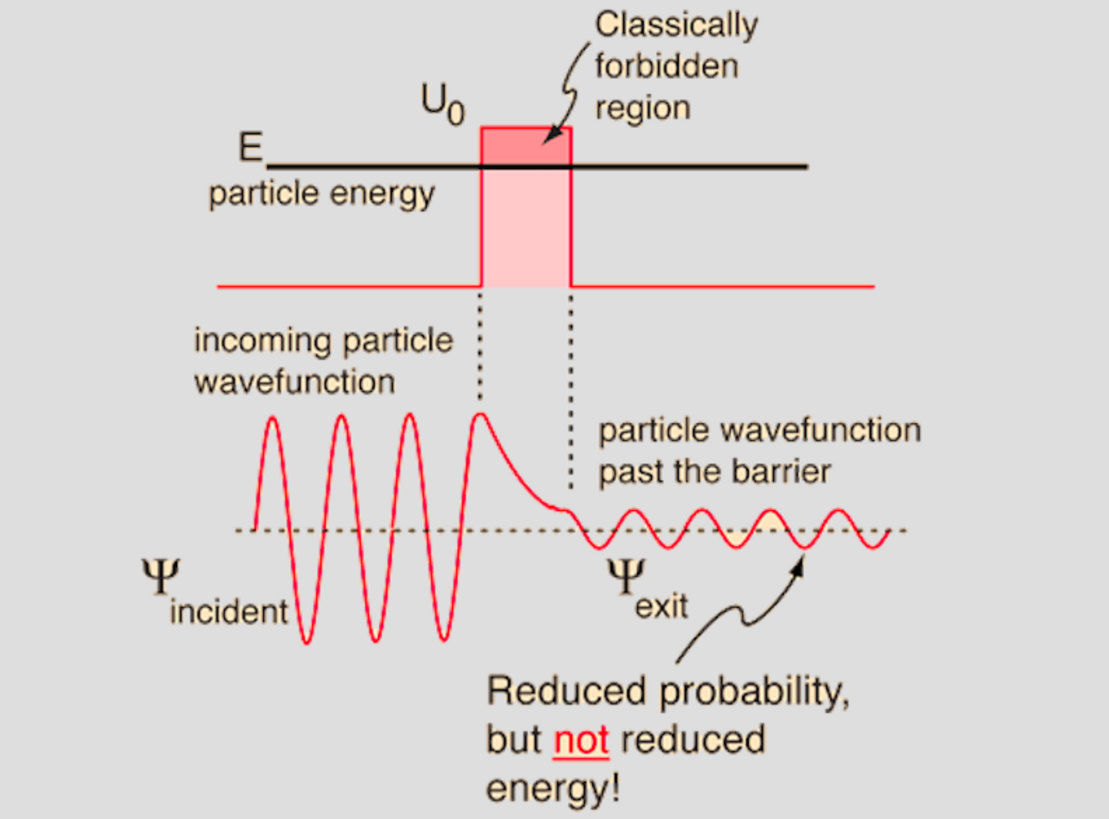


Figure 1.1: Quantum Tunneling [12]

There are many distributed system applications which require a balance out between strong and eventual consistency. For example, consider an online shopping website. Use case such as the billing process should be strongly consistent whereas a recommendation window, based on the users‘ browsing history can be weakly consistent. For such systems, the proposed solution can be deployed for use cases which do not need strong consistency.

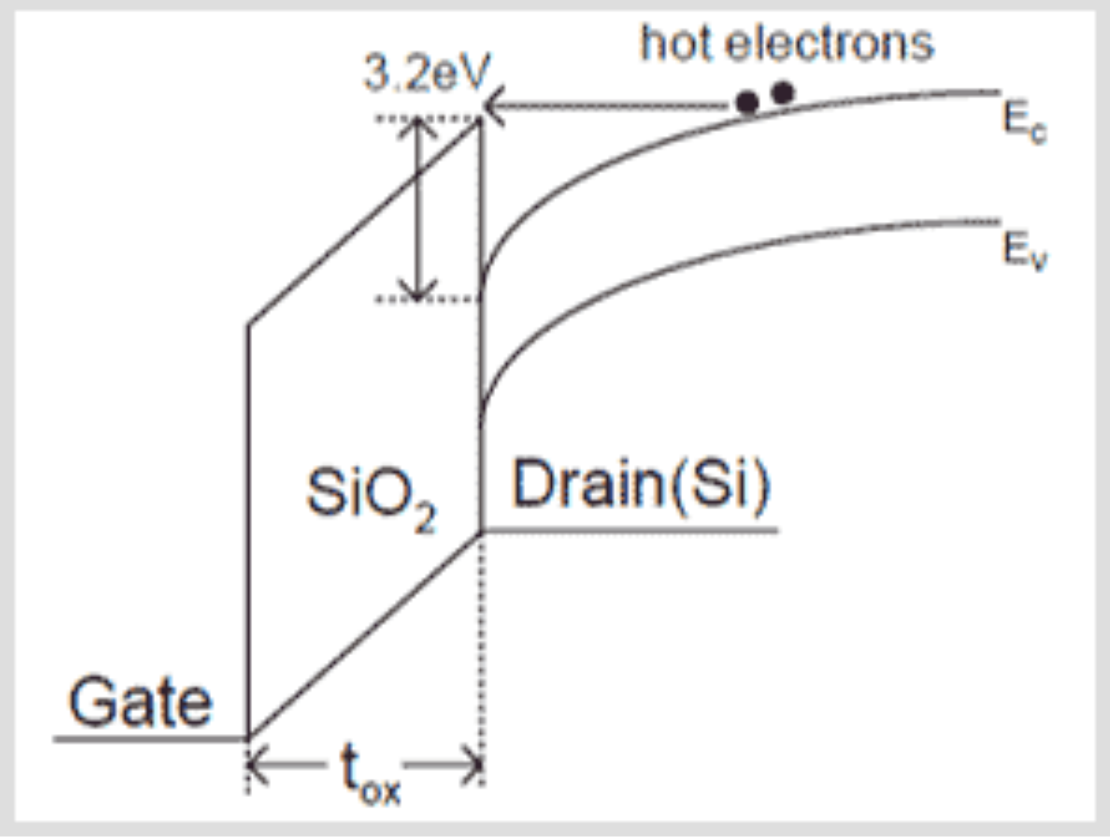


FIGURE 1.2: Hot Electrons Injection[13]

1.2 Motivation

When we consider the flash storage, we know that the number of writes which can be performed on flash storage is in the range of $10^{\mathbf{5}}$ **-** $10^{\mathbf{7}}$. Due to the design of the flash memory, we cannot perform in-place update on a block of data. So we have to go through an erase of the block which completely erases the page by making all bits to zero (by having negative charge) and then perform the write cycle. But this leads to a condition called uneven wearing because a part of the storage is accessed rarely where the data is used only for reading (say a copy of the movie image) and the other part of the data is accessed often for writing (say a part of the disk which is used for paging). To solve this problem **Wear Leveling** was introduced.

But wear leveling increases the number of writes in a flash storage i.e. the **write amplification.**

The main idea of this research is to reduce the number of writes if the same piece of data is present in the RAM or in the storage replica which is about to change shortly.

1.3 Development Environment

The project was mainly developed in University of California Santa Cruz under the guidance of Professor Jishen Zhao in her lab.

Initially it was proposed to develop the code in C language to support and run on Unix based Operating System running any data base. Later for simplicity and to increase the rate of development to implement the proof of concept it was decided to use Python language and pickel DB.

Apart from the pickle DB package for this project we have used python packages like matplotlib, tox, socket and pytest which will be discussed in detail later.

For executing the program, we required minimum 2 separate computers to test the code. Later moved to virtual environment by using the "Oracle virtual box" for dvelopment and test. The code has been tested on different operating systems like the MAC OS, Windows and Linux setup which were used both as a client and server configurations. More details about the architecture and system design is explanined later in the implementation section.

[1] For the project we have used the python version 2.7. Install python version 2.7 by running the below commands on different Operating systems or can also be installed using Anaconda .

1. Ubuntu.

sudo apt-get install build-essential checkinstall sudo apt-get install libreadline-gplv2-dev libncursesw5-dev libssl-dev libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-dev cd /Downloads/ wget https://www.python.org/ftp/python/2.7.12/Python-2.7.12.tgz tar -xvf Python-2.7.12.tgz cd Python-2.7.12 ./configure make sudo checkinstall

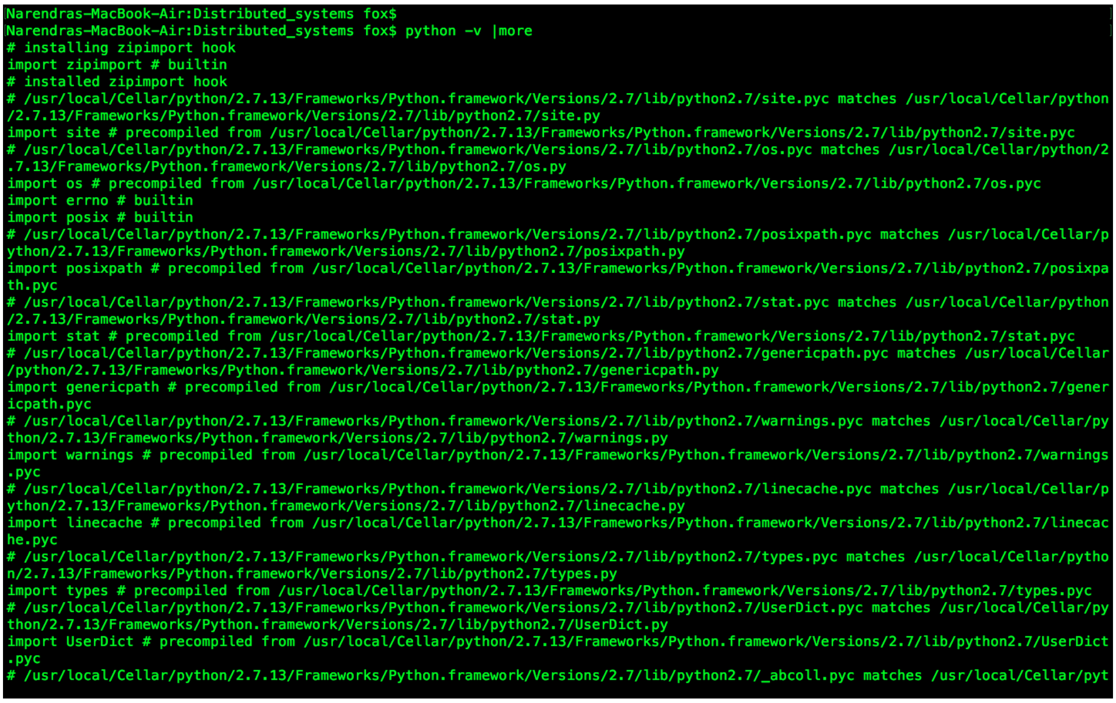
2. MAC OS.

Download package from "https://www.python.org/downloads/mac-osx/" then double click to install python.

3. Windows OS.

Download package from "https://www.python.org/downloads/windows/" then double click to install python.

In order to test if the development environment is working as expected, type "python -v". We will be prompted for python interpreter and check the versions of all the packages and the python version as shown in the figure 1.3.



```
Narendras-MacBook-Air:Distributed_systems fox$
Narendras-MacBook-Air:Distributed_systems fox$ python -v |more
# installing zipimport hook
import zipimport # builtin
# installed zipimport hook
# /usr/local/Cellar/python/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site.pyc matches /usr/local/Cellar/python
/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site.py
import site # precompiled from /usr/local/Cellar/python/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site.pyc
# /usr/local/Cellar/python/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/os.pyc matches /usr/local/Cellar/python/2
.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/os.py
import os # precompiled from /usr/local/Cellar/python/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/os.pyc
import errno # builtin
import posix # builtin
# /usr/local/Cellar/python/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/posixpath.pyc matches /usr/local/Cellar/p
ython/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/posixpath.py
import posixpath # precompiled from /usr/local/Cellar/python/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/posixpa
th.pyc
# /usr/local/Cellar/python/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/stat.pyc matches /usr/local/Cellar/python
/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/stat.py
import stat # precompiled from /usr/local/Cellar/python/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/stat.pyc
# /usr/local/Cellar/python/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/genericpath.pyc matches /usr/local/Cellar
/python/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/genericpath.py
import genericpath # precompiled from /usr/local/Cellar/python/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/gener
icpath.pyc
# /usr/local/Cellar/python/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/warnings.pyc matches /usr/local/Cellar/py
thon/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/warnings.py
import warnings # precompiled from /usr/local/Cellar/python/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/warnings
.pyc
# /usr/local/Cellar/python/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/linecache.pyc matches /usr/local/Cellar/p
ython/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/linecache.py
import linecache # precompiled from /usr/local/Cellar/python/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/linecac
he.pyc
# /usr/local/Cellar/python/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/types.pyc matches /usr/local/Cellar/pytho
n/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/types.py
import types # precompiled from /usr/local/Cellar/python/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/types.pyc
# /usr/local/Cellar/python/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/UserDict.pyc matches /usr/local/Cellar/py
thon/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/UserDict.py
import UserDict # precompiled from /usr/local/Cellar/python/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/UserDict
.pyc
# /usr/local/Cellar/python/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7/_abcoll.pyc matches /usr/local/Cellar/pyt
```

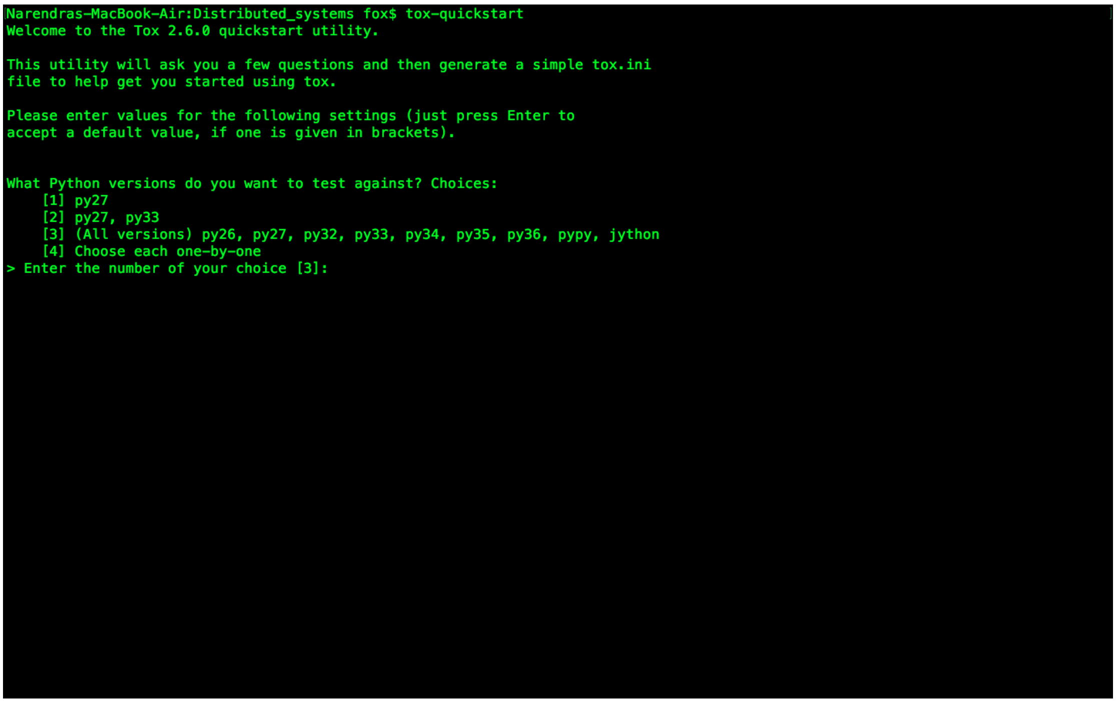
Figure 1.3: Python version

1.3.1 TOX

Tox is a virtual environment for python and it is very help for the creating a separate environment for our development use the command ”pip install tox” to install tox and then run the command ”tox-quickstart” for setting up the environment as shown the below figure 1.4.

1.3.2 PYTEST

For testing and debugging of the python code have used the python test frame work pytest. To install pytest use the command ”pip install pytest” then for debugging using pytest use the inbuilt function ”assert” which will be discussed later in detail. To test and debug run the command pytest as shown in the figure 1.5.



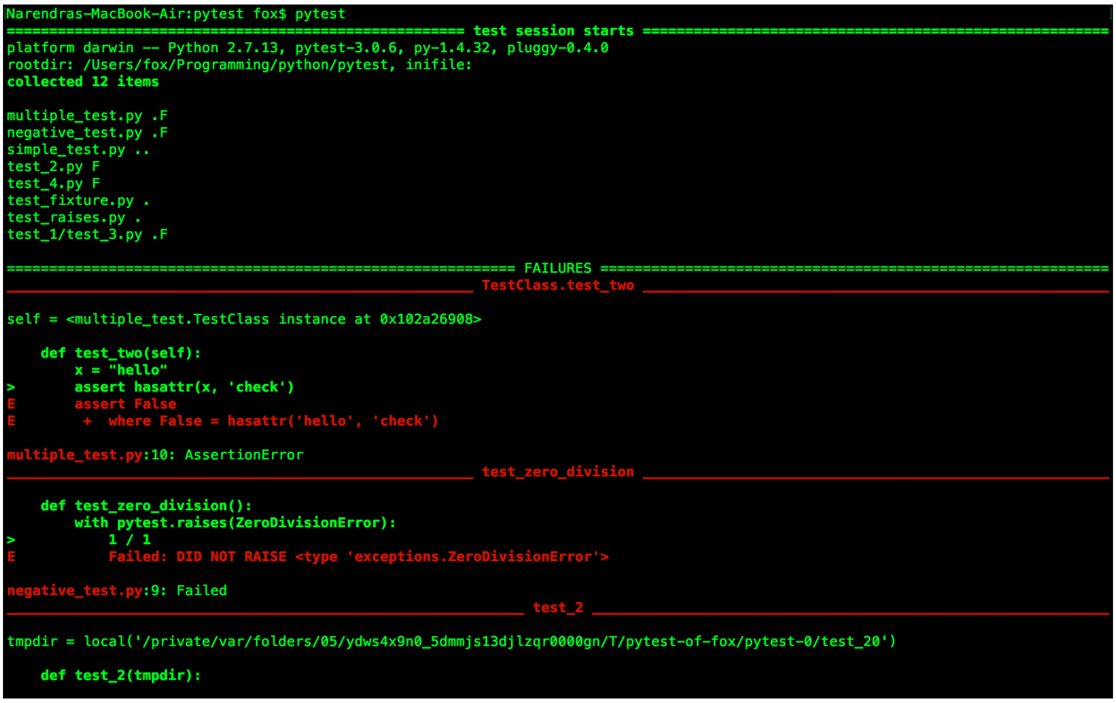
```
Narendras-MacBook-Air:Distributed_systems fox$ tox-quickstart
Welcome to the Tox 2.6.0 quickstart utility.

This utility will ask you a few questions and then generate a simple tox.ini
file to help get you started using tox.

Please enter values for the following settings (just press Enter to
accept a default value, if one is given in brackets).


What Python versions do you want to test against? Choices:
    [1] py27
    [2] py27, py33
    [3] (All versions) py26, py27, py32, py33, py34, py35, py36, pypy, jython
    [4] Choose each one-by-one
> Enter the number of your choice [3]:
```

Figure 1.4: tox-quickstart



```
Narendras-MacBook-Air:pytest fox$ pytest
====================================================== test session starts =======================================================
platform darwin -- Python 2.7.13, pytest-3.0.6, py-1.4.32, pluggy-0.4.0
rootdir: /Users/fox/Programming/python/pytest, inifile:
collected 12 items

multiple_test.py .F
negative_test.py .F
simple_test.py ..
test_2.py F
test_4.py F
test_fixture.py .
test_raises.py .
test_1/test_3.py .F

============================================================ FAILURES ============================================================
_______________________________________________________ TestClass.test_two _______________________________________________________

self = <multiple_test.TestClass instance at 0x102a26908>

    def test_two(self):
        x = "hello"
>       assert hasattr(x, 'check')
E       assert False
E        +  where False = hasattr('hello', 'check')

multiple_test.py:10: AssertionError
_______________________________________________________ test_zero_division _______________________________________________________

    def test_zero_division():
        with pytest.raises(ZeroDivisionError):
>           1 / 1
E           Failed: DID NOT RAISE <type 'exceptions.ZeroDivisionError'>

negative_test.py:9: Failed
_____________________________________________________________ test_2 _____________________________________________________________

tmpdir = local('/private/var/folders/05/ydws4x9n0_5dmmjs13djlzqr0000gn/T/pytest-of-fox/pytest-0/test_20')

    def test_2(tmpdir):
```

Figure 1.5: pytest for testing and debugging

Chapter 2

Implementation

2.1 Design

The main objective of our project is to reduce the total number of writes.

We have two approaches to achieve this.

1. The first approach is by not replicating data to one of the replica, every time there is a write request from the client. The write is delayed for a fixed time and then initiated, this technique is called the **Lazy update.**
2. The second is not replicating the data to any of the replicas instead log all the writes and then update the replicas after certain time. This avoids considerable program-erase cycles if same blocks are rewritten. We will call this **Logging.**

With the first approach the replication is delayed to one of the replica nodes whereas with the second approach the delay is to all the replicas. Thus these approaches are more suitable for weak consistency applications. We wanted to design this approach on CEPH system. But due to time constraints and huge code base of CEPH we have gone ahead to design the model on a simple key, value based distributed system.

Without our solution, the system would update the latest data immediately to all the replicas without any delay as shown in the figure 2.1. Whenever there is a write request from the client, the data is replicated to all the replicas 1-4 immediately and simultaneously. This leads to more program-erase cycle thus lowering the endurance of the flash system.

The data flow diagram with the first approach of our solution is as shown in the figure 2.2. A single replica is chosen from all the replicas for the lazy update. We have chosen Replica 4 for illustration purpose. On the event of a write request from the client,

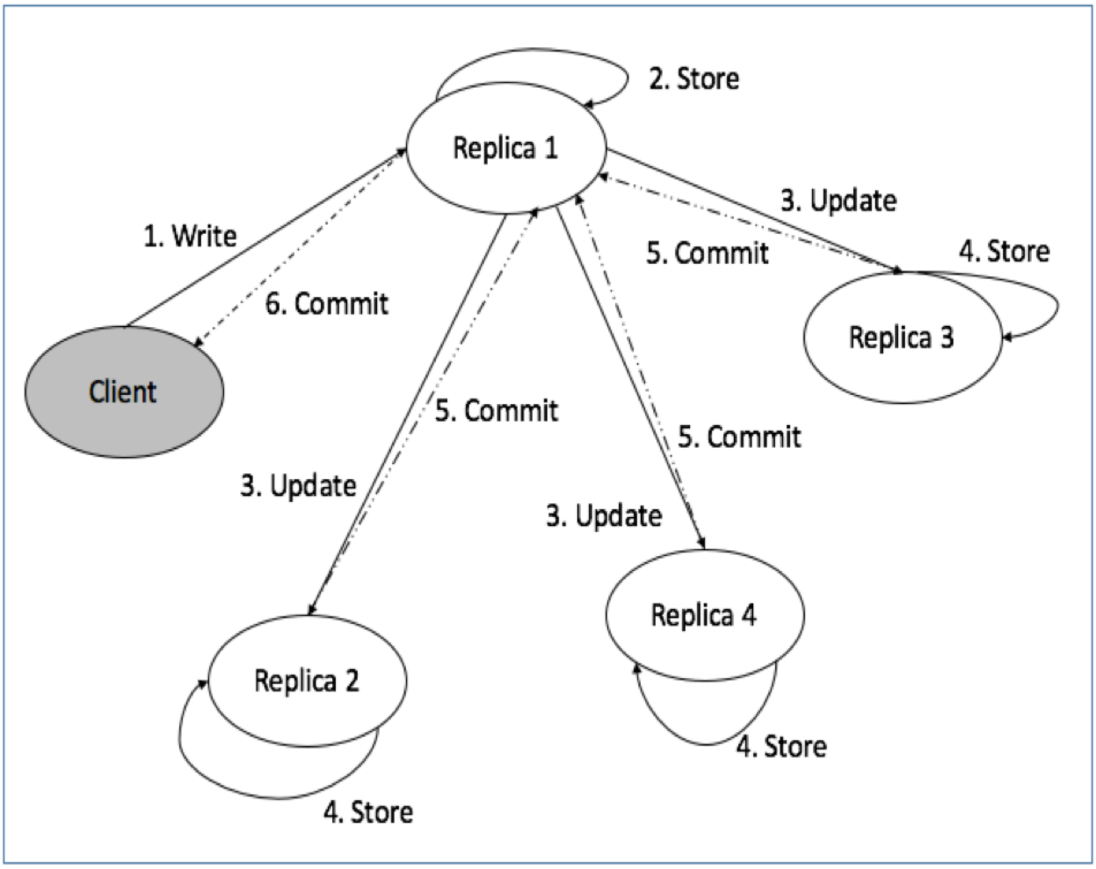


Figure 2.1: Simple key, value pair Distributed System without our solution

the data is replicated to all replicas immediately except for replica 4. Once all the other replicas store the new data and committed, the commit signal is sent back to the client. However, the update to replica 4 is delayed up to 2 minutes or until acceptable consistency delay. This way the number of writes to replica 4 is reduced. This increases the endurance of replica 4. With this approach, we will be able to improve the endurance of one node. The lazy update can be done to any one of the replicas for each write request on a round robin basis, thus improving endurance of all the nodes.

With the second approach, whenever a client sends a write request, it is logged in and commit signal is sent to the client. A finite number of writes is accumulated until it is actually written to all the replicas as shown in figure 2.3 below. After certain time, the accumulated writes are sent, all at once to all the replicas. This reduces the program-erase cycle of the flash system. With this approach, the intention is to improve the endurance of all the replicas.

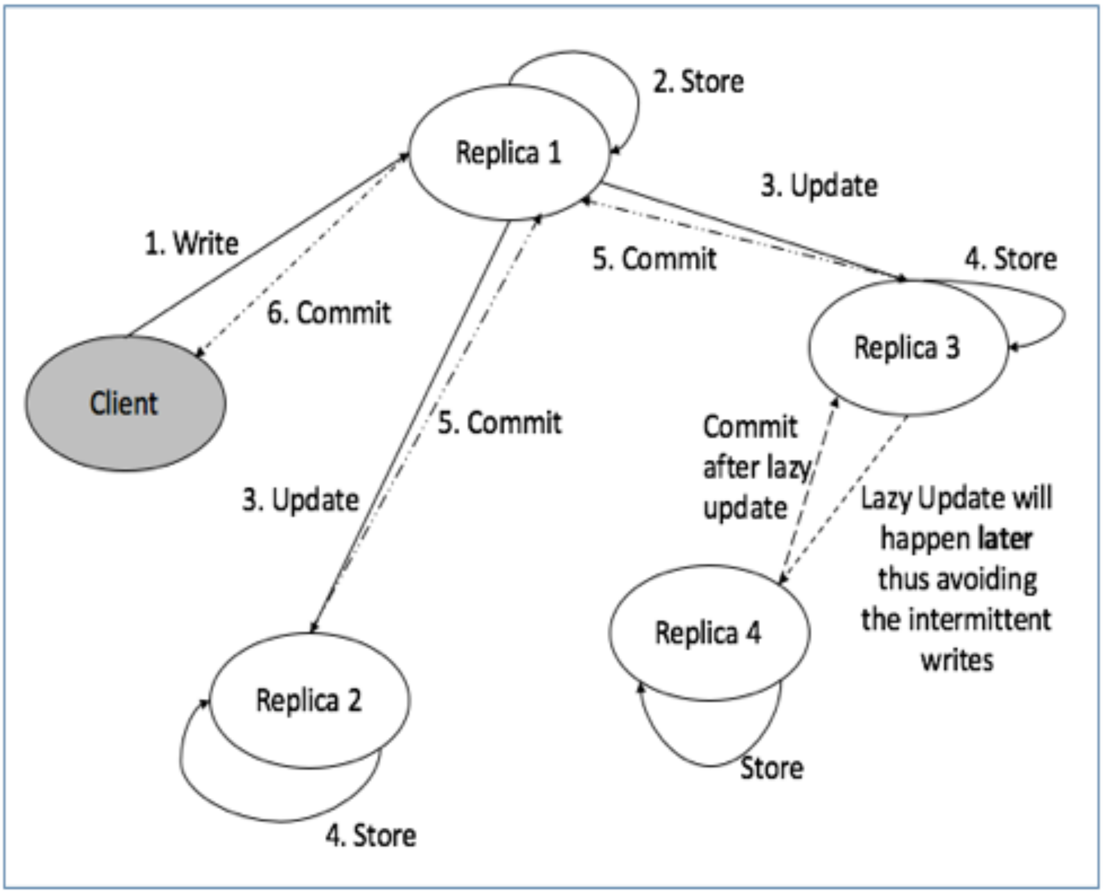


Figure 2.2: Key, Value pair distributed system with lazy update on Replica 4

2.2 File Structure

The code is designed as Client Server model, where the server does the replication task and the client does the load generation. We have designed to support four instances of server code (replicas) running on different physical or virtual machine or at different network ports which is the **replicator.py** code. For the load generation we have clients running connected to different servers which is the **client.py** code.

Apart from the above main code, the grapher.py code is used to plot the number of writes received from all the servers to get the dynamic write counts and to get the number of actual writes written from client which uses the writesfromclient.csv file.

For testing and debugging we have created the tox-quickstart and test cases files under the pytest folder. Pytest framework is used for testing.

Refer figure 2.4 for file structure details.

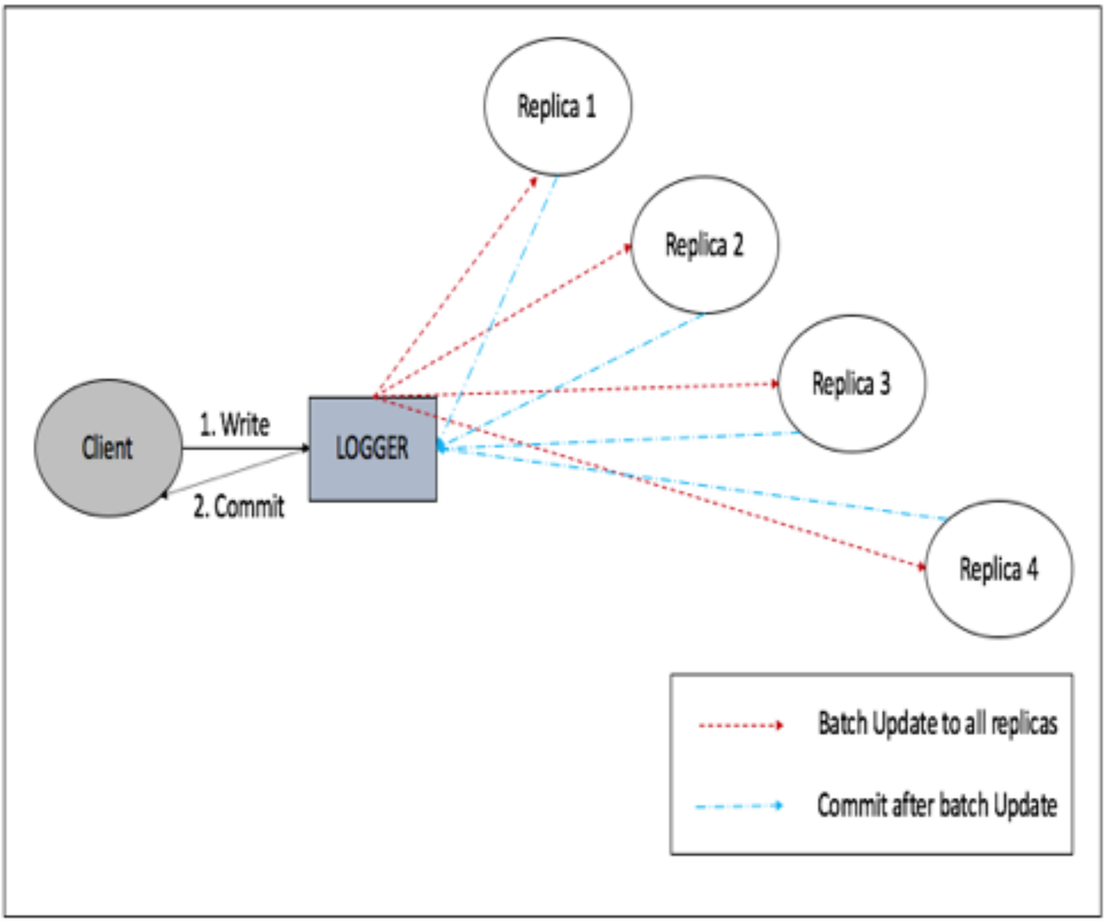


Figure 2.3: Key, Value distributed system with logging for write events

2.3 Data Structures

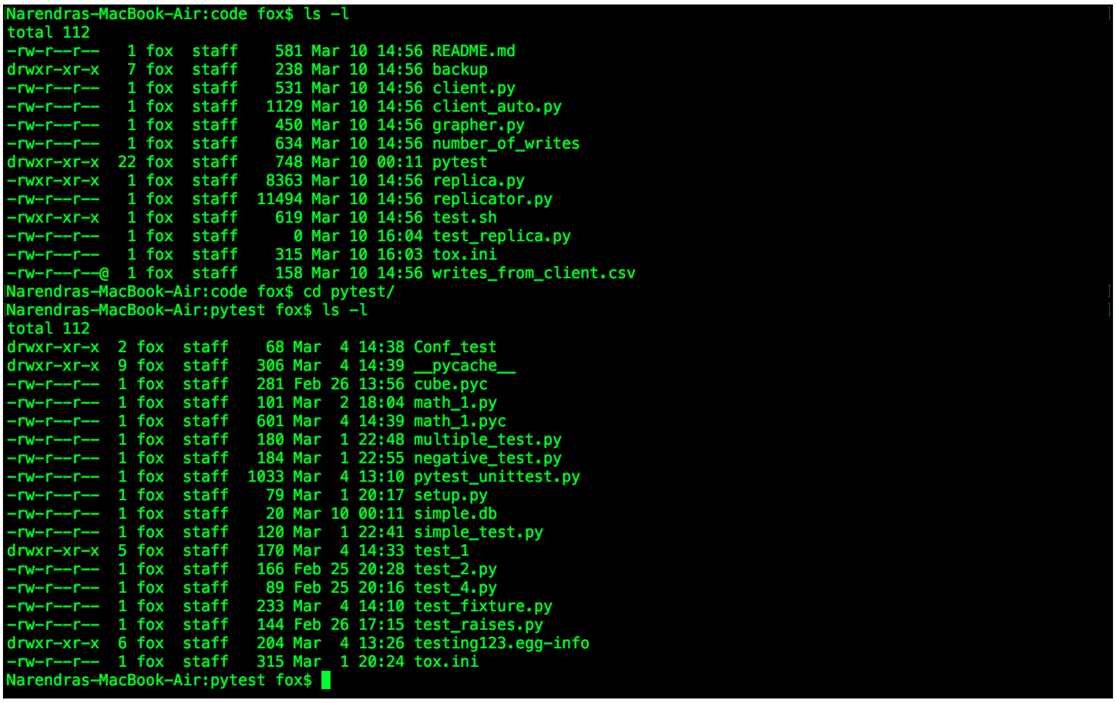
This Distributed system project is a type of dictionary data structure, where each key is unique and it stores a value. This data structure is replicated across all the disaster recovery sites using the network programming socket package which uses the TCP protocol.

And again the same data structure which is the dictionary, is used to store the values at each of the sites.

For this project we are using the pickledb package at each of the replicator and below are the commands supported by pickledb.

DB operations available

1. LOAD path dump [Load a database from a file]
2. SET key value [Set the string value of a key]
3. GET key [Get the value of a key]



```
Narendras-MacBook-Air:code fox$ ls -l
total 112
-rw-r--r--   1 fox  staff    581 Mar 10 14:56 README.md
drwxr-xr-x   7 fox  staff    238 Mar 10 14:56 backup
-rw-r--r--   1 fox  staff    531 Mar 10 14:56 client.py
-rw-r--r--   1 fox  staff   1129 Mar 10 14:56 client_auto.py
-rw-r--r--   1 fox  staff    450 Mar 10 14:56 grapher.py
-rw-r--r--   1 fox  staff    634 Mar 10 14:56 number_of_writes
drwxr-xr-x  22 fox  staff    748 Mar 10 00:11 pytest
-rwxr-xr-x   1 fox  staff   8363 Mar 10 14:56 replica.py
-rw-r--r--   1 fox  staff  11494 Mar 10 14:56 replicator.py
-rwxr-xr-x   1 fox  staff    619 Mar 10 14:56 test.sh
-rw-r--r--   1 fox  staff      0 Mar 10 16:04 test_replica.py
-rw-r--r--   1 fox  staff    315 Mar 10 16:03 tox.ini
-rw-r--r--@  1 fox  staff    158 Mar 10 14:56 writes_from_client.csv
Narendras-MacBook-Air:code fox$ cd pytest/
Narendras-MacBook-Air:pytest fox$ ls -l
total 112
drwxr-xr-x  2 fox  staff    68 Mar  4 14:38 Conf_test
drwxr-xr-x  9 fox  staff   306 Mar  4 14:39 __pycache__
-rw-r--r--  1 fox  staff   281 Feb 26 13:56 cube.pyc
-rw-r--r--  1 fox  staff   101 Mar  2 18:04 math_1.py
-rw-r--r--  1 fox  staff   601 Mar  4 14:39 math_1.pyc
-rw-r--r--  1 fox  staff   180 Mar  1 22:48 multiple_test.py
-rw-r--r--  1 fox  staff   184 Mar  1 22:55 negative_test.py
-rw-r--r--  1 fox  staff  1033 Mar  4 13:10 pytest_unittest.py
-rw-r--r--  1 fox  staff    79 Mar  1 20:17 setup.py
-rw-r--r--  1 fox  staff    20 Mar 10 00:11 simple.db
-rw-r--r--  1 fox  staff   120 Mar  1 22:41 simple_test.py
drwxr-xr-x  5 fox  staff   170 Mar  4 14:33 test_1
-rw-r--r--  1 fox  staff   166 Feb 25 20:28 test_2.py
-rw-r--r--  1 fox  staff    89 Feb 25 20:16 test_4.py
-rw-r--r--  1 fox  staff   233 Mar  4 14:10 test_fixture.py
-rw-r--r--  1 fox  staff   144 Feb 26 17:15 test_raises.py
drwxr-xr-x  6 fox  staff   204 Mar  4 13:26 testing123.egg-info
-rw-r--r--  1 fox  staff   315 Mar  1 20:24 tox.ini
Narendras-MacBook-Air:pytest fox$ █
```

Figure 2.4: File Structure

4. GETALL [Return a list of all keys in database]

5. REM key [Delete a key]

6. APPEND key more [Add more to a key's value]

7. LCREATE name [Create a list]

8. LADD name value [Add a value to a list]

9. LGETALL name [Return all values in a list]

10. LEXTEND name seq [Extend a list with a sequence]

11. LGET name pos [Return one value in a list]

12. LREM name [Remove a list and all of its values]

13. LPOP name pos [Remove one value in a list]

14. LLEN name [Return the length of a list]

15. LAPPEND name pos more [Add more to a value in a list]

16. DCREATE name [Create a dict]

17. DADD name pair [Add a key-value pair to a dict, ”pair” is a tuple]

18. DGETALL name [Return all key-value pairs from a dict]

19. DGET name key [Return the value for a key in a dict]

20. DKEYS name [Return all the keys for a dict]

21. DVALS name [Return all the values for a dict]

22. DEXISTS name key [Determine if a key exists]

23. DREM name [Remove a dict and all of its pairs]

24. DPOP name key [Remove one key-value in a dict]

25. DELDB [Delete everything from the database]

26. DUMP [Save the database from memory to a file specified in LOAD]

2.4 Code Explanation

It all starts by calling the python script replicator.py which will become the master replicator and the script will scan the ports and the mode of operation (Lazy update/Logging) provided by the user.

The master replictor starts listening for the client‘s connection request at the intiated port and parallely listens for all other slave replicators by using the hardcoded port numbers provided in the code.

Later the user has to start other slave replicators on the TCP port which the master port is listening to. Thus establishing the connection between the master replicator and all the slave replicators.

Next the client script should be started to make a connection to the master replicator. the client script is the load generator that will create the key value pairs. This generated load will be sent over the TCP socket created by the master replicator.

Once the master replicator receives the data from the client, it will be first copied to its local pickledb. After few seconds the data will be written to the other slave replicas for disaster recovery or back up purpose. This mechanism is logging.

Later to enhance the design, we added a mode called the lazy update which has the option for slave replicators to handle their own clients. If there is any update on the slave replicas as discussed above, data will be first written to the local pickledb and then it will be propogated to other slave nodes and the master node. But having the master node up all the time is a must, as it initiates the listening session for all the slave nodes.

The initial design and implementation was done on a python disctionary later it was enhanced to support pickledb. Adding support for other data bases should be simple.

The difficulty was to debug the issues when we ran the code as we have distributed codes running in multiple sites. inconsistent behaviour was seen while corelating any syncing up the multiple replication sites and clients to act as one system. Triaging was very difficult while manual testing. To overcome this we automated the test cases and used pytest for testing each function and debugging to see if we could achieve consistent behaviour. More about pytests and tox will be discussed later in Testing and Debugging section.

Chapter 3

Results

3.1 Overview

As mentioned in the implementation we designed our own simple key value pair distributed data base, that supports simple DB commands and we have used the below two commands extensively.

DB Commands used extensively

1. SET
2. GET

Once the data is sent by the client to any of the distributed system, instantly data will be replicated to all the available replicas. (Here for simplicity we have used 4 replicas).

As the names of the commands represent we will be able to set a key with a value using the command: “set 1 10“, that sets key 1 with value 10. And to retrieve the data we can execute the get command: “get 1“ which fetches the value 10 in this example.

3.2 Theoretical Analysis

Theoretically the number of writes in a key value pair for a dictionary data structure will take 1 write for each value writes or update. In a replication setup the number of writes, will be equal to the number of replication sites times the number of writes.

For analysis purpose lets consider we have 4 replication sites and for the regular key-value pair data structure, we will have 4 times the number of key-value update. Which is depicted in the graph 3.1.

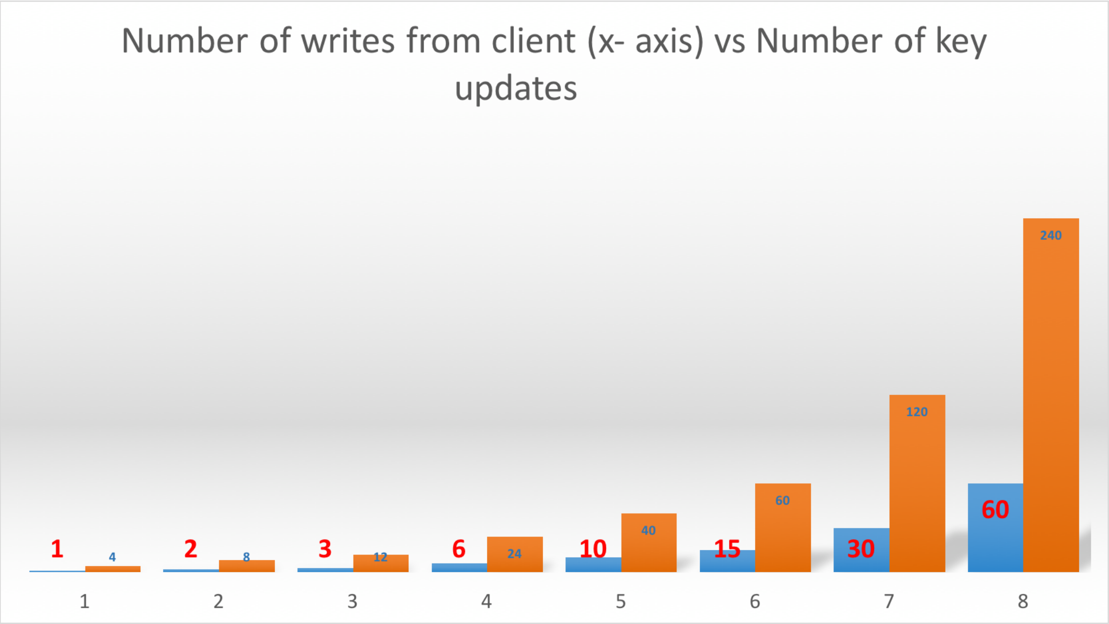


Figure 3.1: Theoretical estimate for simple key, value pair distributed system without our solution.

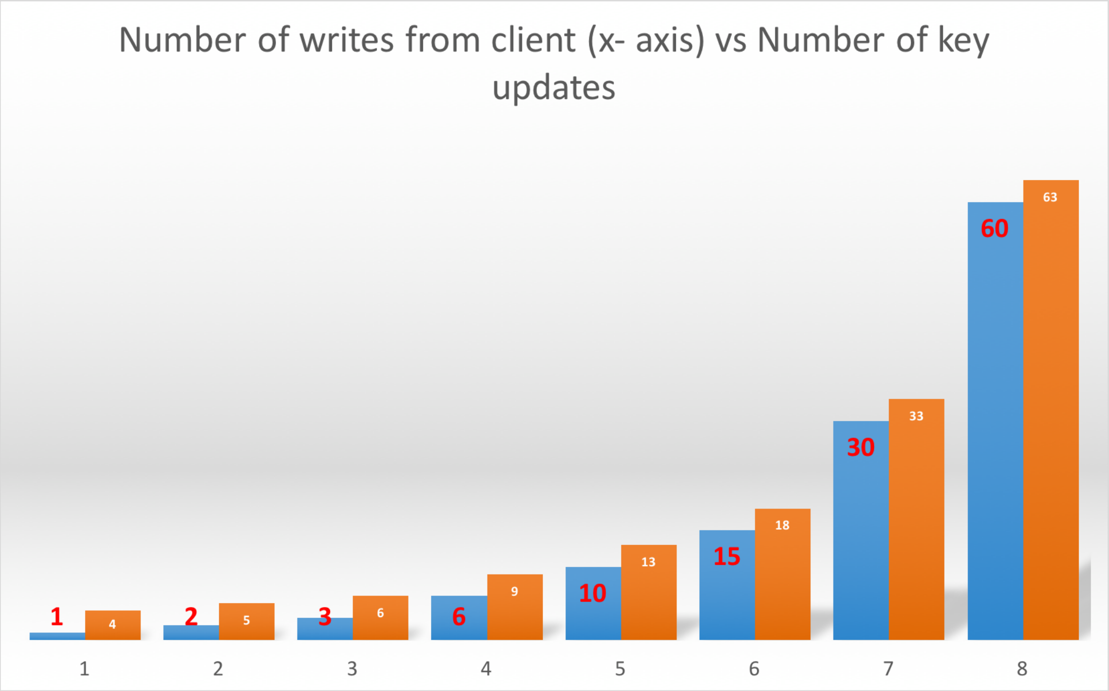


Figure 3.2: Theoretical estimate for simple key, value pair distributed system with our solution of logging.

Similarly for theoretical analysis lets consider we have the logging mechanism where we have same 4 replication sites. Instead of writing all the writes immediately, the writes are delayed and replicated after 30 seconds. So theoritcally we will be writing only once for each of the key-value update until 30 seconds. Lets assume we will have around 60 writes during 30 seconds, in this scenario there will be only 60 writes to the main replicator and plus 3 for other replictors, which totalls to 63 writes instead of 240 writes. Here we are assuming that we will have 60 update to the same key-value. So we are saving 240 - 63 = 177 writes.

3.3 Practical Results

To implement logging solution as discussed in implementation, we created a timer mechanism in which captured all the writes in the log and wait for specified interval of time say 30 seconds. (This can be tuned as per the requirement and for the remainder of the example we will consider timer of 30 seconds.)

Practically with our solution we were able to observe the below comparison and there is difference in the theoretical and the practical values. This is because we are using a random number gnerator for load generation and also the timer will not be synced with all the 4 sites.

Figure 3.3 and 3.4 are the comparison of total number of writes actually generated at the clients to the total number of writes done at the replicas without and with our solution respectively.

The total number of writes for 67 updates with out our solution were 268 writes. Whereas with our solution the number of writes at all the replicas totalled to 157 for the same 67 updates from the client. We can see from this that we were able to save a total of 111 writes, so there was a reduction of 58 percent of writes.

Complete project code is available under: GIT HUB link

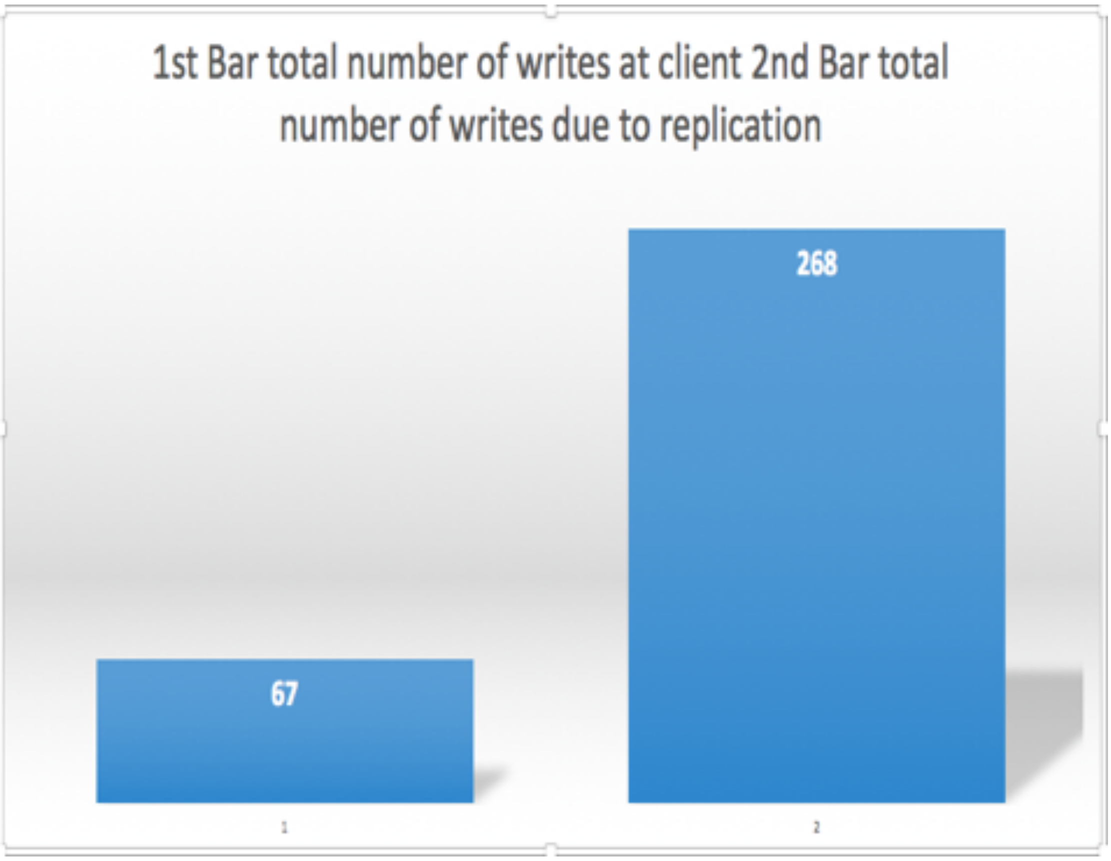


Figure 3.3: Practical results: Total number of writes from all the clients against the total number of key value updates across replicas without our solution.

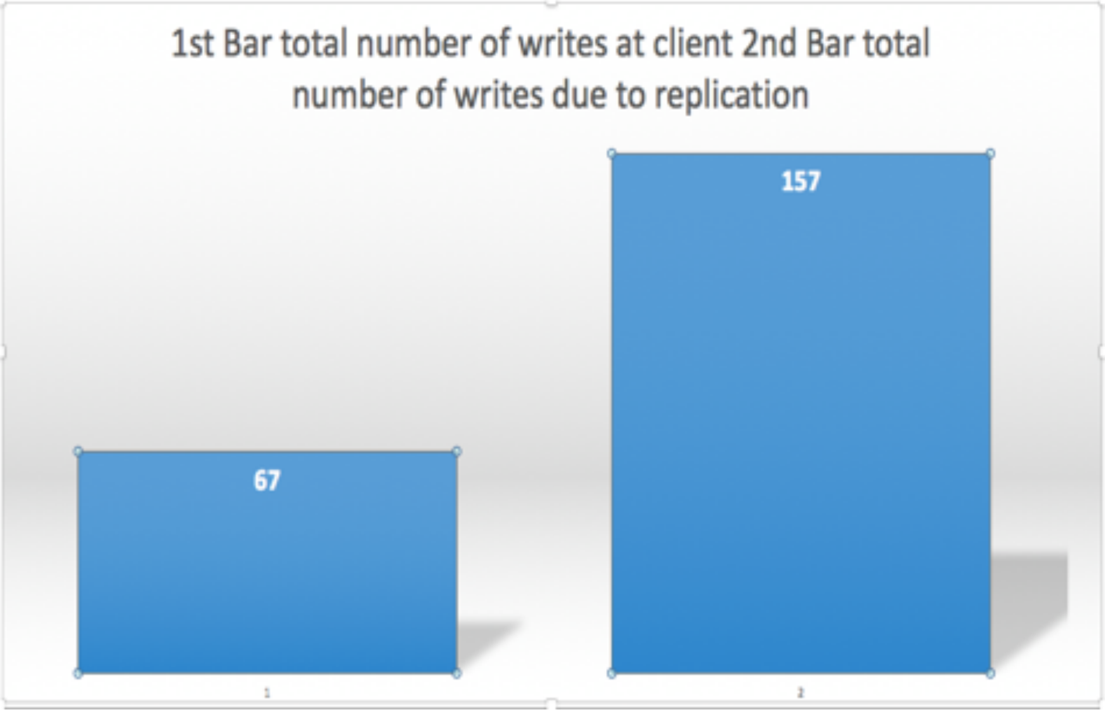


Figure 3.4: Practical results: Total number of writes from all clients against the total number of key value updates across replicas using the logging solution.

Chapter 4

Debugging and Testing

In order to test and debug, “pytest” needs to be installed. Once it is installed, it is as simple as calling the pytest with the executable script to run and verify the results.

Brief description of pytest

It is a test framework written in python to test code written in python.

Very easy to start using the framework and supports to scale complex functional testing.

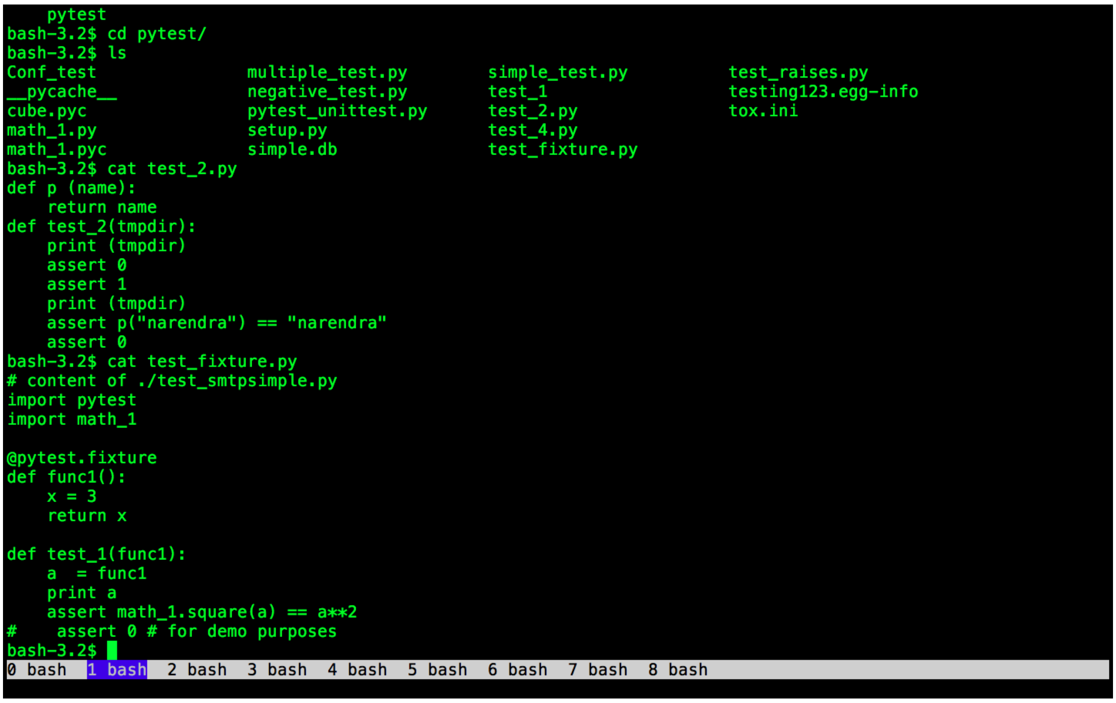
Community support and documentation is very good.

Supports 150+ plugins.

For our project we used below functionalities which is available in python and part of pytest for tetsing and debugging.

1. assertion [For testing each function, works like ”if not” and raises an exception]
2. fixtures [For setting up configuration before and then to do clean up after testing is completed]
3. pytest.raises [To perform negative testing and to test exceptions]
4. conftest [Used to provide the config details and to provide scope of the fixtures]
5. tox [To create virtual environment which provides option to select different verisons of the packages and python versions]
6. devpi[Repository manager]

The figure 4.1 displays all the above operation done in python.



```
    pytest
bash-3.2$ cd pytest/
bash-3.2$ ls
Conf_test               multiple_test.py        simple_test.py          test_raises.py
__pycache__             negative_test.py        test_1                  testing123.egg-info
cube.pyc                pytest_unittest.py      test_2.py               tox.ini
math_1.py               setup.py                test_4.py
math_1.pyc              simple.db               test_fixture.py
bash-3.2$ cat test_2.py
def p (name):
    return name
def test_2(tmpdir):
    print (tmpdir)
    assert 0
    assert 1
    print (tmpdir)
    assert p("narendra") == "narendra"
    assert 0
bash-3.2$ cat test_fixture.py
# content of ./test_smtpsimple.py
import pytest
import math_1

@pytest.fixture
def func1():
    x = 3
    return x

def test_1(func1):
    a  = func1
    print a
    assert math_1.square(a) == a**2
#    assert 0 # for demo purposes
bash-3.2$
0 bash  1 bash  2 bash  3 bash  4 bash  5 bash  6 bash  7 bash  8 bash
```

Figure 4.1: Testing and Debugging in pytest

Chapter 5

Improvements

The main idea of the project was to explore memory and storage coordinative ECC and wear leveling schemes across memory and storage components to implement a solution where the number of writes on the disk can be reduced as we have the same data on the RAM.

This requires access to SSD firmware and the OS driver which are proprietary softwares. It is difficult to get access and requires high technical knowledge to understand and implement the firmware code which also requires lots of time effort.

Our project is the proof of concept to show that we can have write reduction and this should be extended to support for the SSD drives in the future.

Other features which can be extended for this project are:

- Support general data bases like SQL DB.
- Use data from real scenario for data generation.
- Integrate IO-meter as the data generator.

References

1. RELIABLY ERASING DATA FROM FLASH-BASED SOLID STATE DRIVES: Michael Wei, Laura M. Grupp, Frederick E. Spada , Steven Swanson University of California, San Diego

2. STUDY OF BAD BLOCK MANAGEMENT AND WEAR LEVELING IN NAND FLASH MEMORIES: Supriya Kulkarni P1, Jisha, Electronics and Communication Dept, MVJ Engineering, Bangalore

3. HRAID6ML: A HYBRID RAID6 STORAGE ARCHITECTURE WITH MIRRORED LOGGING: Lingfang Zeng, Dan Feng , Janxi Chen Qingsong Wei, Bharadwaj Veeravalli , Wenguo Liu

4. CSWL: CROSS-SSD WEAR-LEVELLING METHOD IN SSD-BASED RAID SYSTEMS FOR SYSTEM ENDURANCE AND PERFORMANCE: Kwanghee Park, Dong-Hwan Lee, Youngjoo Woo, Geunhyung Lee, Ju-Hong Lee, Deok-Hwan Kim Dept. of Electronic Engineering, Inha University.

5. RELAIBILITY AND PERFORMANCE ENHANCEMENT TECHNIQUE FOR SSD ARRAY STORAGE SYSTEM USING RAID MECHANISM: Kwanghee Park, Dong-Hwan Lee, Youngjoo Woo, Geunhyung Lee, Ju-Hong Lee, Deok-Hwan Kim Dept. of Electronic Engineering, Inha University.

6. AN EMBEDDED FTL FOR SSD RAID: Alistair A. McEwan and Irfan Mir Department o f Engineering University of Leicester, Leicester LE1 7RH, UK

7. RELIABILITY MANAGMENET TECHNIQUES IN SSD STORAGE SYSTEMS : Irfan F. Mir University of Leicester

8. BUILDING FLEXIBLE, FAULT TOLERANT FLASH BASED STORAGE SYSTEMS: Kevin M. Greenan Darrell D.E. Long Ethan L. Miller Thomas J. E. Schwarz, S.J. Avani Wildani Univ. of California, Santa Cruz Santa Clara University

9. WEAR LEVELLING USING MACHINE LEARNING: Coughlin Associates, Flash Memory Summit 2016.

10. REJUVENATOR: A STATIC WEAR LEVELLING ALGORITHM FOR FLASH MEMORY: Murugan.M, Du. D, Department of Computer Science, University of Minnesota

11. INCITS Technical Committee T10 TRIM command RFEs

12. Image 1.1 from hyperphysics

13. Image 1.2 from fhwa.gov