CSE-535 ASYNCHRONOUS SYSTEMS

PROJECT REPORT

ViewStamped Replication in DistAlgo

By: (Group Number - 24)
Shobhit Khandelwal (112074908)
Vivek Bansal (112044493)
Nishant Kulkarni (112072405)

Table of Contents

Part 1: Problem and Plan

Motivation

Distributed Consensus algorithms like Paxos are widely known in the industry. By comparison to Paxos, one distributed consensus protocol seems frequently overlooked. This algorithm was first described by the Barbara Liskov and Brian M. Oki in the paper[1] and later revised in the paper “ViewStamped Replication Revisited” by Barbara Liskov and James Cowling [2]**.** At first, it’s easy to believe that two algorithms are similar. The similarities can be seen when looking at how that view number is chosen: VR's view change protocol. In fact, the view change protocol of “Viewstamped Replication Revisited” bears a striking resemblance to the Paxos Synod protocol. Although these algorithms are different from each other and it has been described beautifully in a paper by van Renesse et. al., titled “Vive La Difference: Paxos vs. Viewstamped Replication vs. Zab”[3]. One key difference is the passive replication in VR as compared to the active replication in Paxos. This makes VR to have better performance and efficiency as compared to Paxos or Raft for that matter.

Problem Description

There are numerous algorithms out there which try to solve the problem of node replication (state machine replication) correctly and efficiently. It is however important to justify why these algorithms are correct. This requires thorough reasoning and testing to make sure that the algorithm is robust and will achieve the desired properties even in case of failures upto some known limit. In this project we will be studying the ViewStamped Replication algorithm described in the paper ***“ViewStamped Replication Revisited” by Barbara Liskov and James Cowling[2].*** We have implemented the algorithm from scratch using the pseudo-code described in the paper. We have also tested the algorithm for correctness and generated different performance graphs and analysed them.

Input and Output

Input: We took the existing implementation of ViewStamped Replication in DistAlgo as the pseudo code and implemented it in distAlgo. The program input is defined in more details later on in this report under implementation section.

Output: Tested correct working of the algorithm under various modes of operation and did performance analysis on the same.

Testing and Performance Evaluation

We have implemented the TestManager class which is responsible for collecting running logs from the nodes in the system and perform all correctness and performance evaluation. We tried to add more automated system for error injection in the system by providing the user configurable files which could be read by the testManager when the system is brought up. This framework is however not tested for robustness and requires more work. We did many runs to obtain performance statistics as provided in further section of the report.

Optimization Scope/ Future Work

Several possible optimisations are possible and can be taken up as future work:

1. **Batching:** Instead of running the protocol each time request occurs, bunch of requests are collected in the primary and the protocol is run for the batch once. If the primary is receiving a lot of requests, then we can queue up the requests (using a list) and invoke our protocol for this queue.

2. **Improvement in view-change operation:** The protocol described has a small number of steps, but big messages. A reasonable way to get good behavior most of the time is for replicas to include a suffix of their log in their DO-VIEW-CHANGE messages. Therefore sending the latest log entry, or perhaps the latest two entries, should be sufficient.

Part 2: Design

In VR Algorithm as we mentioned in the problem statement can be understood from the below mentioned diagram.

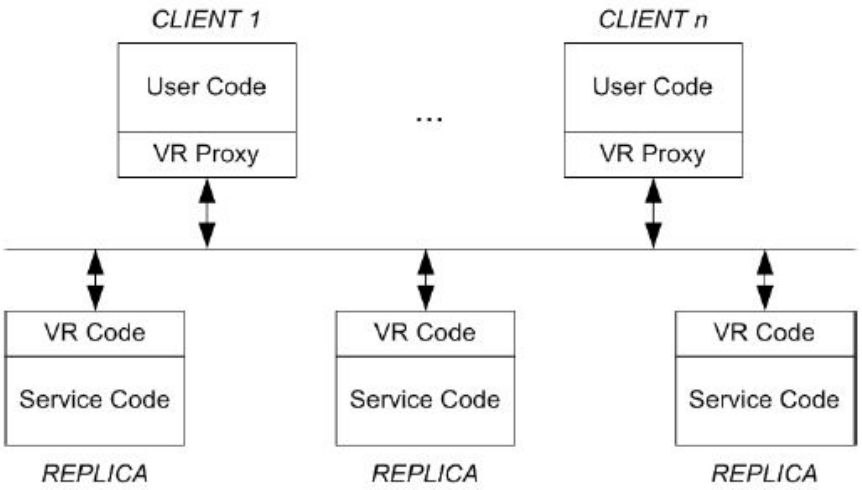


Fig 1: Generic Architecture of a State Replication System

We have designed our implementation as described in the model described in Fig 2(Design diagram). We have 4 different modules/classes in our code which interact with each other as shown by the directional arrows. Each arrow symbolizes the certain information among these classes. These classes are implemented in separate files to maintain modularity in case of future expansion.

We have two main classes for the algorithm implementation:

1.Client class

2. Replica class

They are spawned from the main function. The protocol has 3 modes of operation: Normal mode, View change mode and Recovery mode.

- ●View Change mode occurs whenever the primary node fails.
- ●Recovery mode is the one when a previously failed replica tries to join back on the network. This part is failing for some cases and needs some effort.

For the purpose of testing and performance evaluation we have the build the following design.

1. The Monitor process starts running along with other nodes and keeps a check on all active nodes in the system by means of sending and receiving messages.

2. At the end of the simulation the monitor class receives execution logs from all the processes (These logs can be huge therefore we chunk it to the bare minimum at the client/replica side). Once the logs are received , the Monitor will check for correctness and also provides performance statistics for the entire run. Once this is completed it informs the main program and then the program exits

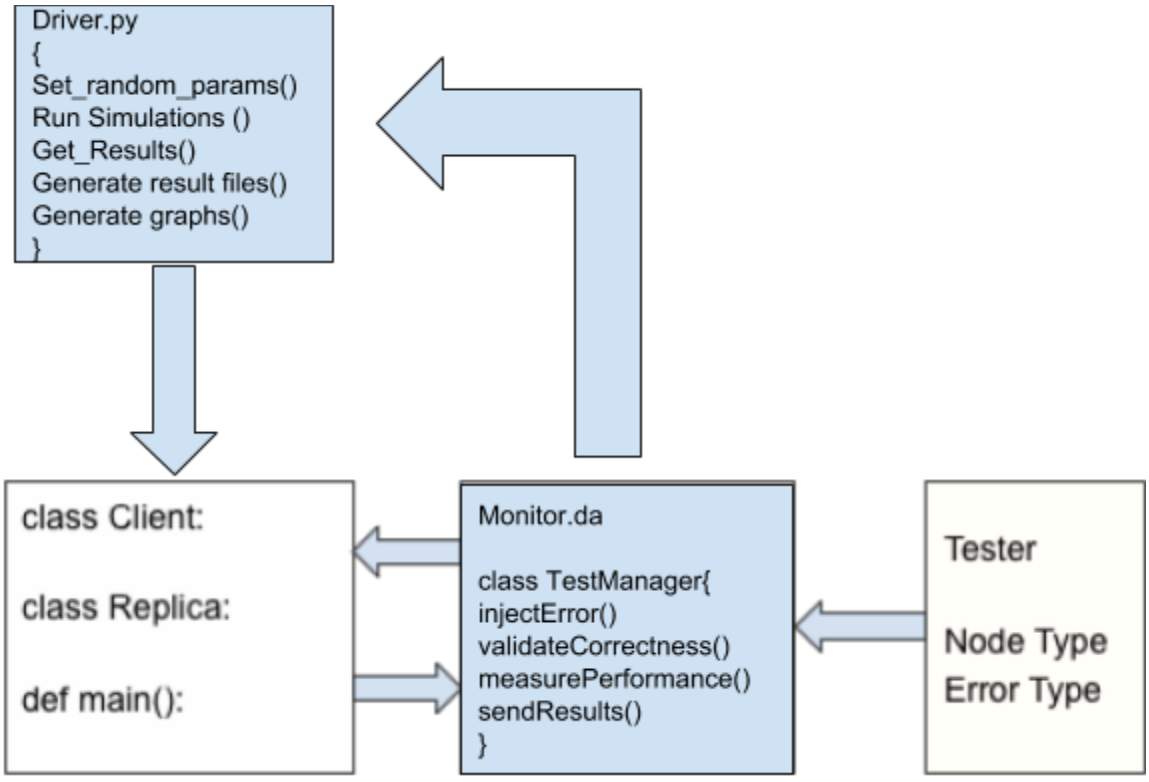


Fig 2 : DESIGN DIAGRAM

.

ADDITIONAL FUNCTIONALITIES FOR TESTING

- The TestManager Class (Monitor.da) is also capable of automating error injection based on configurable file given by the user.
- This TestManager class will read input from this config file. This file contains information regarding what needs to be tested and what are the parameters for error injection

- The TestManager class will have a function called injectError() which will inject the required error. This function is independent on its own; it is not waiting or blocked by some other process or condition.

The input to the Testing framework would be the Tester file. It's high - level format is as follows (Example file is given in source directory as test.cfg):

INPUT PARAMETERS	EXAMPLE 1	EXAMPLE 2
Test Name	Test1	Test10
Error Type	Crash	Message Delay/ Isolation
Node Type	Client	Replica
Node Number	3	10
Error Message	"Client 3 has crashed"	"Replica 10 is in message delay"

Table 1: Sample Config File Parameters

Part 3: Implementation

Following are the list of source files which have been written as part of this project.

Files:

File Name	Functionality
vr_revisited_new.da	Main Viewstamped Replication Implementation
Monitor.da	Monitor program
driver.py	Driver program and error injector program
test.cfg	Configuration file for testing

Table 2: Files of the VR algorithm implementation

Input Format:

Our code takes the following as input:

INPUT PARAMETERS	EXPLANATION
f	Number of max replica failures supported, This sets the number of replicas to 2f+1 automatically
c	Number of clients
req	Total number of requests per client
c_tout	Client timeout
r_tout	Replica timeout

Table 3: Input format with meaning of parameters

Running the programs:

There are two modes in which the user can run the program:

Execution Mode 1 :

This mode is the one where the user just want to run the algorithm for a particular set of execution parameters as described above in the table.To run the program in this mode, give the following command in the source directory:

Command: python -m da vr_revisited_new.da f c req c_tout r_tout

Output: In this mode the output will be available as logs from the system on the console.

Execution Mode 2 :

This is an automated testing mode where the user just need to run the driver program and then the driver program takes the job of giving parameters to the algorithm and also takes care of generating correctness and performance statistics.

Run the following command in the source directory:

Command: Python driver.py

Output: In this mode the driver program will run the algorithm for multiple parameter values and in the end will output the correctness and performance statistics in the "**output_vr_rev**" folder as **"validation.csv".** It will also generate the performance graph.

Limitations : In order to change the variable against which the performance statistics should be obtained, one need to change manually in the driver program code to produce graphs.

Implementation Specifics

1. **Client class:** This class is implemented to mimic the client behavior as described in the paper. There were some missing information in the paper as to what client does if the request timed out more than once for the same request number, We have assumed that it will keep resending the request in such case.
2. **Replica Class:** This is the class where most of the algorithm resides. We have kept each operation as a separate API in our code and have kept them as isolated as possible in order to have simplicity and more readability. This class have few

main data structures which are the backbone for all operations. One such data structure is **“logs”** and the other one i**s “client_table”.**

It also implements a heartbeat mechanism between primary and backup replicas which is important for initiating View Change operation. The code has been well documented with reference to paper for each line of the code that converts the algorithm in English to actual code in DistAlgo. Many API’s can be verified for exact line-to-line matchings which is amazing.

3. **TestManager class:** We have implemented the TestManager class takes input from the user in a file. This file contains information regarding what needs to be tested in a particular run**.** We wait for all client operations to finish and then invokes our TestManager to run validation check on the logs collected from all the replicas. The liveness condition is checked by timeouts which have been placed and monitored by the TestManager.

```
vs_max =  get_vs_max(c)          # It compares the request-number in the request with the information in the client table.
if s > vs_max:                   # If the request-number s isn't bigger than the information in the table it drops the request
    n = n + 1                                  # The primary advances op-number
    logs.append((n,m))                         # adds the request to the end of the log
    client_table[c]  = (s, False, None)        # Updates the information for this client in the client-table to contain the new request number 's'
    send(('PREPARE', v, m, n, k), to=replicas)     # Then it sends a (PREPARE v, m, n, k) message to the other replicas,
```

Fig 3 : Code Snippet to show pseudo code to implementation ease in DistAlgo

Given below is the code summary and some useful statistics:

INPUT PARAMETERS	EXAMPLES
Language	DistAlgo
Lines of Code(Main algorithm)	557
Pure Lines of Code(Main algorithm)	477
Lines of Code(Testing)	389

Table 4: Code Statistics

Code Link:

https://github.com/unicomputing/vr-distalgo

Part 4: Testing and Evaluation:

The paper suggests several correctness criteria which we have implemented in our code. These are the abstracts which we have took directly from the paper and implemented in the code:

1. Correctness Testing: For View-Change operation

Safety: The correctness condition for view changes is that every committed operation survives into all subsequent views in the same position in the serial order. This condition implies that any request that had been executed retains its place in the order.

```
def check_lists(l1, l2):
    if len(l1) != len(l2):
        return False
    for i in range(len(l1)):
        if l1[i] != l2[i]:
            return False
    return True

def check_correctness():

    ans = []
    for key, value in replica_logs.items():
        if ans == []:
            ans = value
        else:
            if not check_lists(ans, value):
                return "0"
    return "1"
```

Fig 4: Code Snippet for Safety Check

Liveness: The protocol executes client requests provided at least f + 1 non-failed replicas, including the current primary, are able to communicate. If the primary fails, requests cannot be executed in the current view. However if replicas are unable to execute the client request in the current view, they will move to a new one.

Test Cases:

We performed various testing runs to check the performance and correctness of our code. We changed the number of input parameters, including number of clients, number of max failures and number of replicas. Here is one of many results obtained:

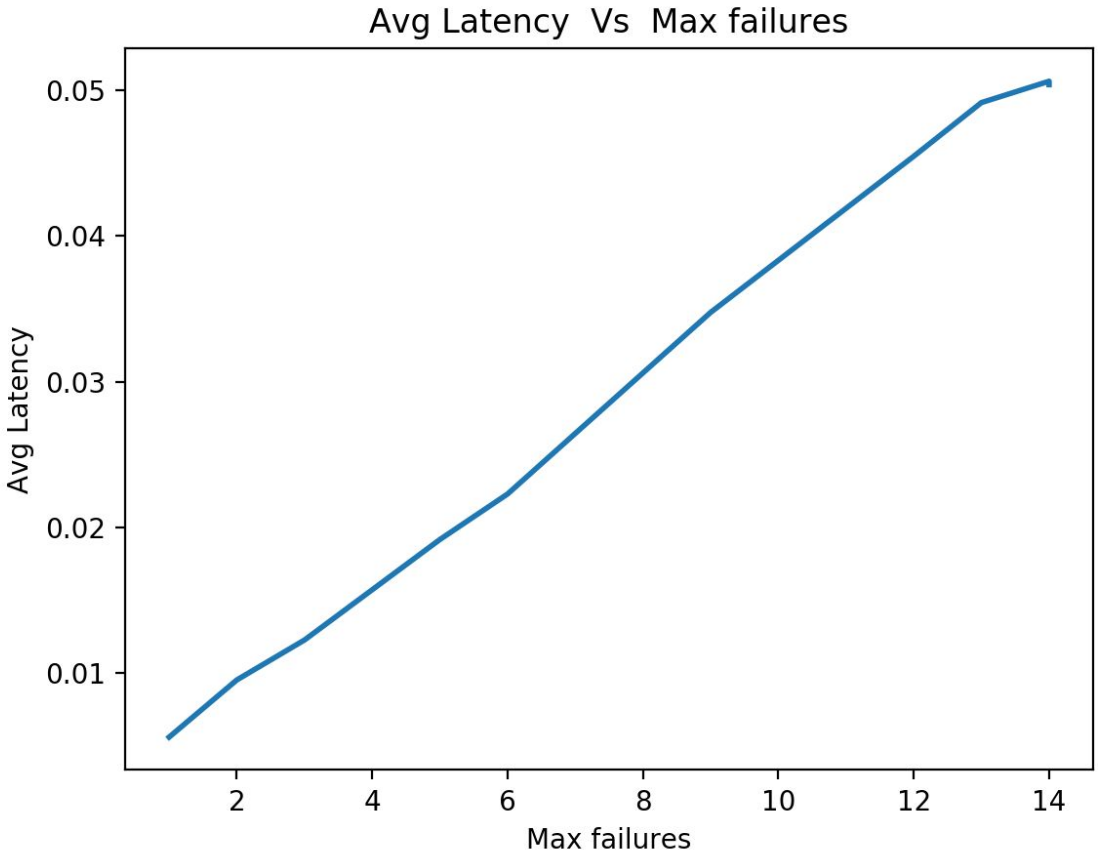
	Num Clients	Max failures	Num of Req/Client	C_Timeout	R_Timeout	Liveness	Correctness
0	24	2	5	10	5	True	True
1	21	2	5	10	5	True	True
2	7	2	5	10	5	True	True
3	45	2	5	10	5	True	True
4	39	2	5	10	5	True	True
5	11	2	5	10	5	True	True
6	23	2	5	10	5	True	True
7	22	2	5	10	5	True	True
8	26	2	5	10	5	True	True
9	10	2	5	10	5	True	True

Table 5: Test Run for Varying Num Client keeping rest same

Interesting Fact: We ran our implementation for over 50 clients , 25 replicas and 50 request per each client and were able to obtain correct working of the algorithm for 0 injected errors and num injected errors = 1.

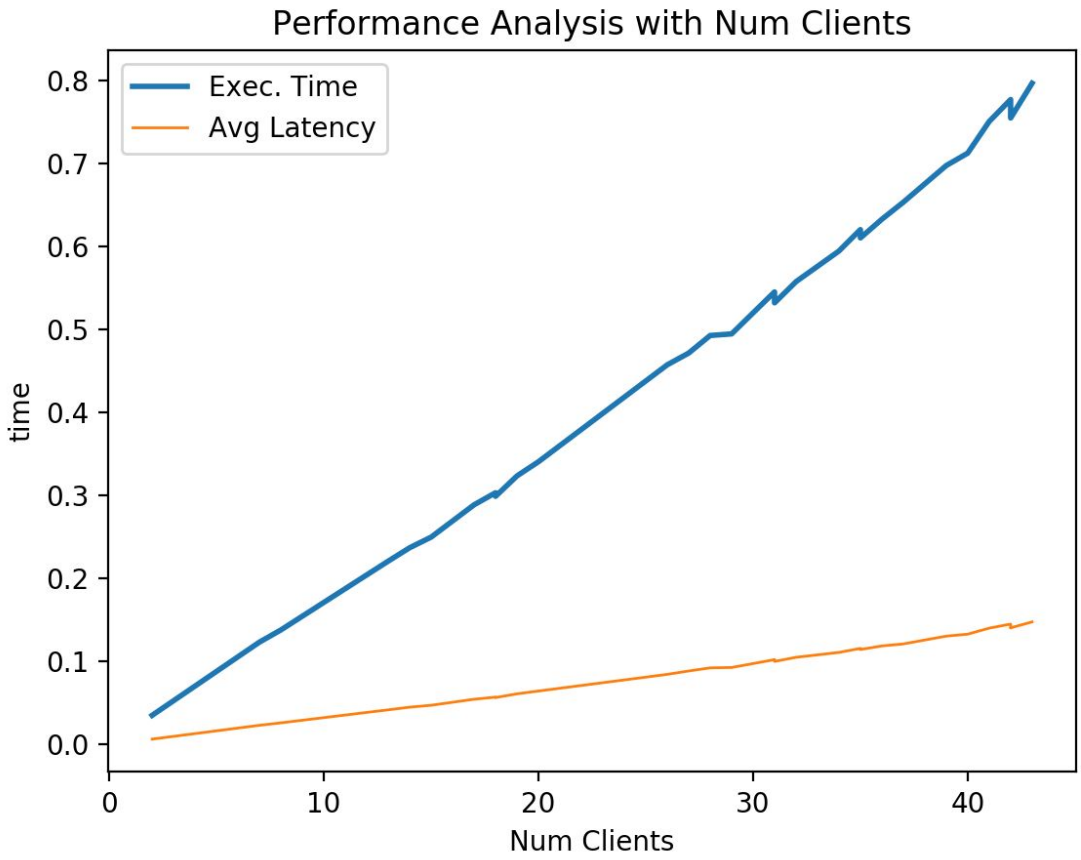
Performance Testing Results:

Here are some interesting plots we made through our performance testing.

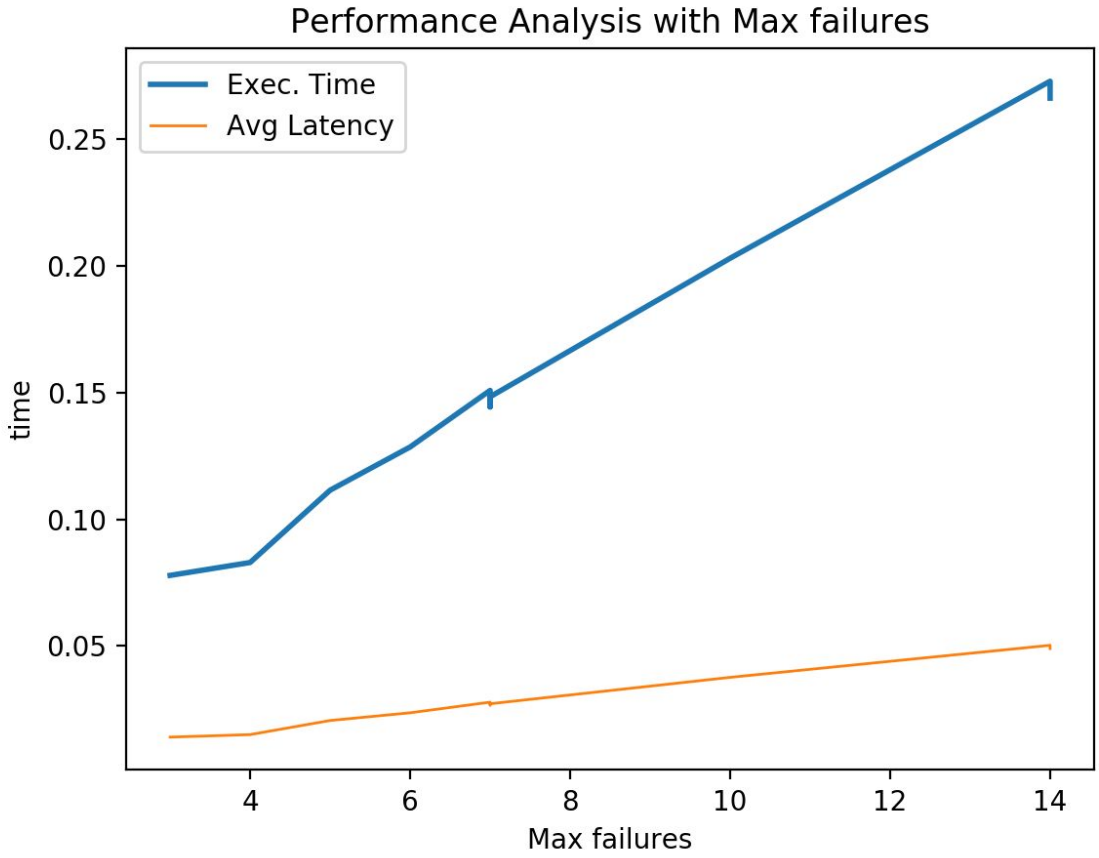


Interpretation: We see that the average latency increases linearly with increasing number of failures of replicas. This is because when a replica fails, it goes into view change mode, which is requires more computation.

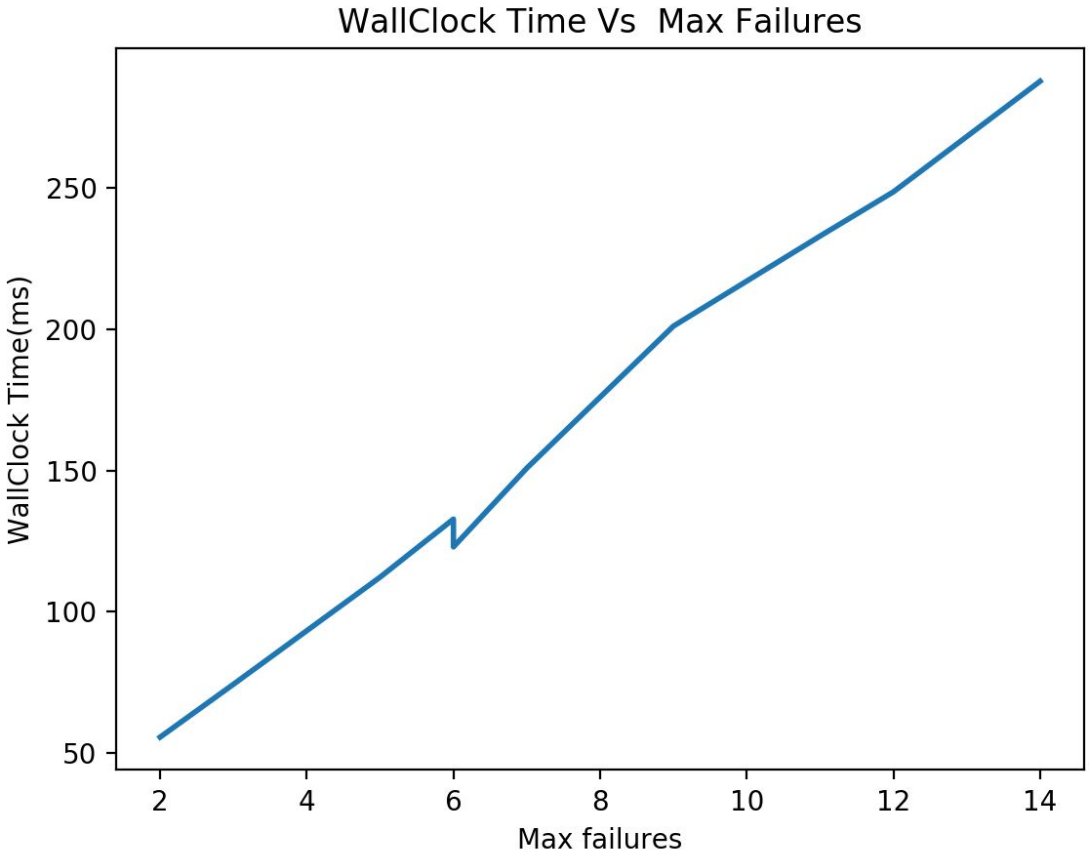
Interpretation: We see that the overall execution time also increases linearly with increasing number of failures of replicas. This is because when a replica fails, it goes into view change mode, which is requires more computation.



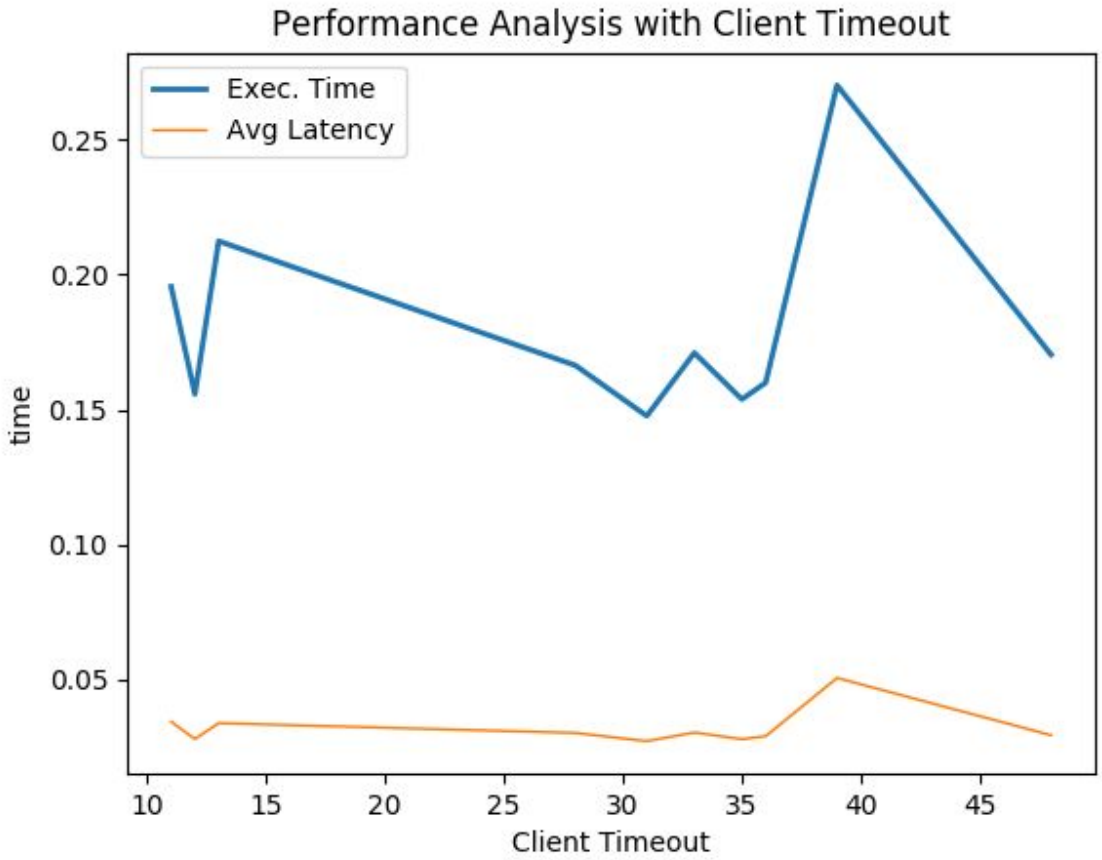
Interpretation: We see that latency and execution time increases linearly with increasing number of clients. This result is quite obvious.



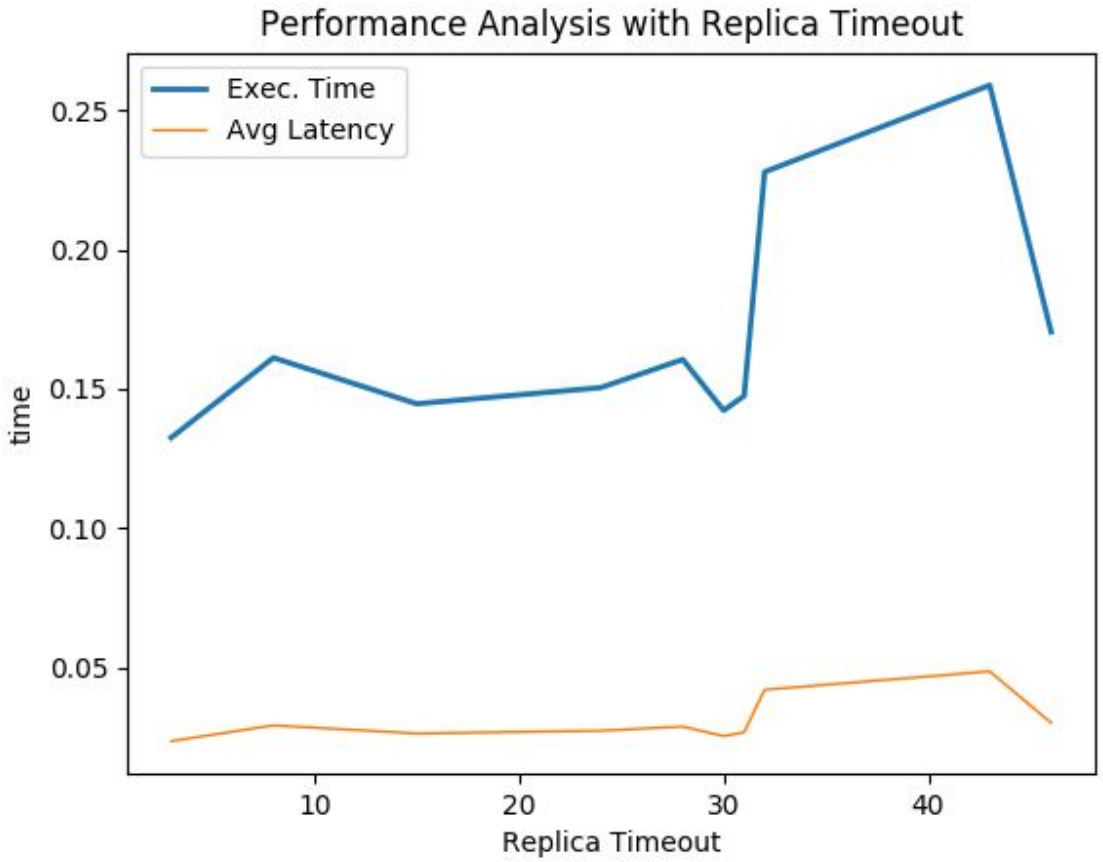
Interpretation: We see that the execution time increases as we increase the number of failures. This is because when a replica fails, the protocol undergoes view change operation, which takes lot of computational time. Also the avg latency remains more or less same which is quite interesting.



Interpretation: Max Failures here can be thought of as directly proportional to number of replicas in the system(2*f +1). We observed that the wall clock time increases as we increase the number backup replicas. This can be explained by the fact that in order to achieve consensus from the quorum of replicas primary now has to wait for more time for every request thus we see a linear increase in the wall clock time.



Interpretation: As we can see, there is no correlation between client timeout and Execution time since the protocol behaves independently of the client timeout.Although certain spikes in latency plot in the graph reflects view change happening.



Interpretation: As we can see, there is no correlation between replica timeout and performance. This is correct because there should not be any correlation between them.

Also we do see that the initial low execution time could be due to the fact that the replica were timing out more quickly and resulting in system exiting for failing liveness criteria.

Observations Summary :

From the above results, we conclude that the code is following safety conditions every time. From the performance analysis, we see that the execution time of the system increases when we increase the number of clients and also when we increase the number of failures of replica nodes which aligns well with our expectations.

Part 5: References

1. Barbara Liskov and Brian M. Oki, “Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems”, 1988
2. Barbara Liskov and James Cowling, ”ViewStamped Replication Revisited”, 2002
   http://pmg.csail.mit.edu/papers/vr-revisited.pdf

3. Vive La Différence:Paxos vs Viewstamped Replication vs Zab”, Robbert van Renesse, Nicolas Schiper, Fred B. Schneider, “ 2014

   Paxos vs. Viewstamped Replication vs. Zab