Practical Byzantine Fault Tolerance(PBFT) in DistAlgo

Project Report

Abhinav Srivastava- 112129749

Sriram Gattu- 112007687

Rohit Kumar Pandey- 112078737

12.10.2018

CSE 535: Asynchronous Systems

PART 1: PROBLEM AND PLAN

INTRODUCTION

PBFT (Practical Byzantine fault tolerance) is a replication algorithm that can tolerate Byzantine faults. Earlier, Byzantine fault-tolerant algorithms had assumptions and requirements that were infeasible to attain and accept in practice. PBFT offers a practical solution to the BFT problem without imposing too compromising restrictions. In retrospect, PBFT can be seen as an extension of ViewStamped Replication as suggested by Barbara Liskov[9] to handle the possibility of Byzantine-faulty nodes. In addition to extending VR to handle Byzantine nodes, PBFT introduced an innovation in the form of proactive recovery and provided a number of optimizations to improve performance.

PROBLEM DESCRIPTION

Implement PBFT in DistALgo

PBFT - Overview

- Replicas in a PBFT system are sequentially ordered with one replica being the leader(Primary) and others referred to as backup replicas. All replicas in the system communicate with one another with the goal being that all honest replicas will come to an agreement of the state of the system using a majority rule.
- Communication between the replicas has two functions: Replicas must prove that messages came from a specific peer replica, and they must verify that the message was not modified during transmission.
- For the PBFT system to function, the number of malicious replicas must not equal or exceed one-third of all replicas in the system.
- PBFT consensus rounds are called views and are broken into four phases:

  1. A client sends a request to the Primary replica to invoke a service operation.
  2. The Primary replica broadcasts the request to the backup replicas.
  3. The replicas execute the requests and then sends the reply to the client.
  4. The client awaits the f+1 number of replies from different replicas with the same result, where f represents the maximum number of potentially faulty replicas

- The Primary Replica can be replaced with a new replica by a protocol called a

View Change if a certain amount of time has passed without the Primary replica broadcasting the request. Also, a supermajority of honest replicas can determine when a Primary is faulty and replace them with the next Primary replica in

DistAlgo

DistAlgo is a very high-level, simple, powerful language for programming distributed algorithms and has been tested on GNU/Linux and Microsoft Windows.
Some of its salient features are:

- distributed processes as objects, sending messages
- yield points for control flow, handling of received messages
- await and synchronization conditions as queries of msg history
- high-level constructs for system configuration

Input: A set of clients and replicas.

Output: A Byzantine Fault Tolerant system which will eventually reach consensus with legitimate safety and liveness check and the reached consensus output value.

STATE OF ART

While there are many documents over the internet that give an inkling about the practical Byzantine fault tolerance algorithm, we could hardly find any straightforward implementation of the basic pbft algorithm albeit, the paper[1] which encompassed the pseudocode written in plain English, gave us a better understanding of the algorithm, which we would want to take as a reference to moving forward with. We have chosen Distalgo to implement pbft, given how powerful and relatively easy DistAlgo is. There is an open source GitHub implementation of pbft[4], where the code was written in python.

TASK PERFORMED

Program New

Implemented PBFT in DistAlgo. This is the first implementation of PBFT in DistAlgo

Test Correctness

Implemented methods to checks Correctness: Safety and Liveness

Test Performance

Extended Controller.da from DistAlgo Github and added some additional feature to capture the performance of the code with different combinations of input.

PROJECT PLAN

Team Member:	Abhinav	Sriram	Rohit
Week 1	Brainstorm and understand the PBFT paper[1] and the algorithm. Decide performance evaluation matrices. Discuss in detail about the scope of the project	Brainstorm and understand the PBFT paper[1] and the algorithm. Decide performance evaluation matrices. Discuss in detail about the scope of the project	Brainstorm and understand the PBFT paper[1] and the algorithm. Decide performance evaluation matrices. Discuss in detail about the scope of the project
Week 2	Design APIs: Driver, Test and Performance collection matrix code Prepare Design Document Starting with the implementation of APIs	Design APIs: Replica and Client Prepare Design Document Starting with the implementation of APIs	Design APIs: Replica and Client Prepare Design Document Starting with the implementation of APIs
Week 3	Continue with the left implementation Brainstorm for Safety and Liveness Implementation Brainstorm for Performance Matrix collection. Safety, Liveness Implementation, Integration with other modules.	Continue with the left implementation. Brainstorm for Safety and Liveness Implementation. Brainstorm for Performance Matrix collection Safety, Liveness Implementation. Performance matrix collection implementation.	Continue with the left implementation Brainstorm for Safety and Liveness Implementation. Brainstorm for Performance Matrix collection Performance matrix collection implementation. Safety Liveness implementation.
Week 4	Prepare PPT presentation of the work done. Final report submission.	Prepare PPT presentation of the work done. Final report submission.	Prepare PPT presentation of the work done. Final report submission.

PART 2: DESIGN

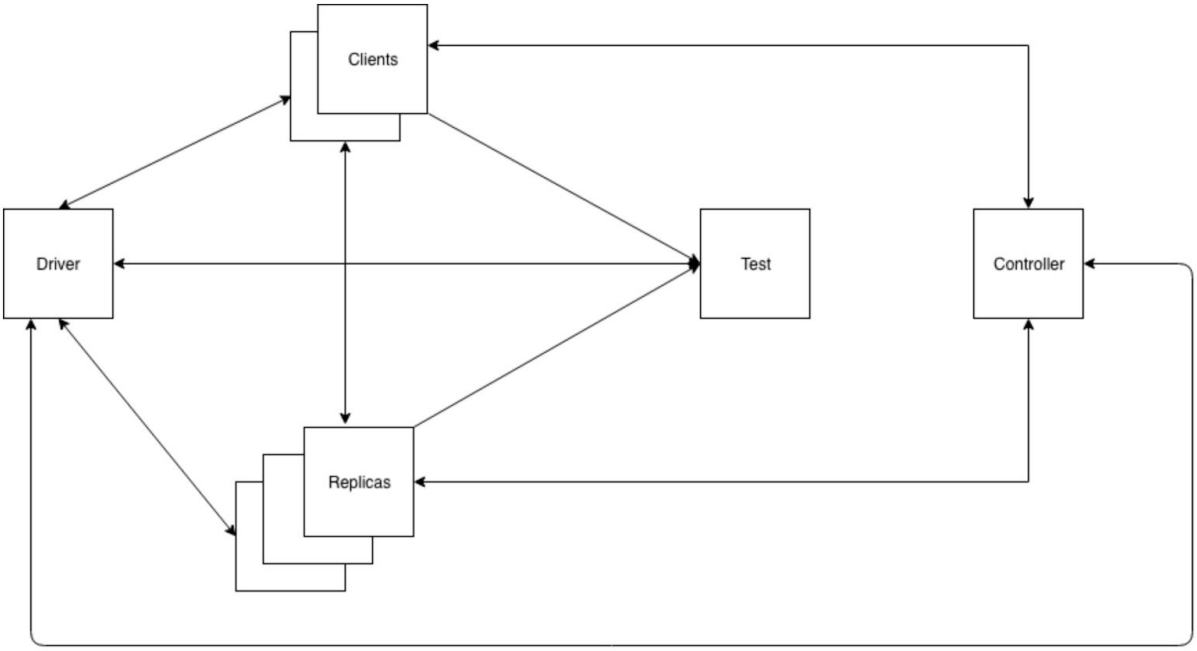


Figure 1: Basic block diagram of the processes and their interactions/dependencies

- **Driver**: This is the driver process which spawns the requested number of clients and replicas. It also creates a test process and a controller process. It awaits done message from all the clients, replicas, and the test process. This will create the CSV file for the performance metric collected from the controller process.
- **Clients**: Clients sends the request operation to the replica and wait for the operation to be completed. Once the operation is completed i.e. the result is received from the replicas it sends this information to the Test process for tracking Liveness property and to the Driver for termination.
- **Replicas**: Replicas accept the operation request from the Clients and perform the required operation. It also interacts with the Test process in case of view changes to track the Safety property and to the Driver for termination.
- **Test**: Listens for the messages from the Clients and Replicas to track the Liveness and Safety properties. Also, passes this information to the Driver process to collate all the information.

- **Controller**: This is the code taken from DistAlgo repo [7] with some modifications done to it. This basically does performance benchmarking for all the processes.

Replica APIs

S.No.	API	Functionality(Overview)
1.	receive(msg= ('REQUEST',m))	To receive the request from Client: When the primary, p, receives a client request, m, it starts a three-phase protocol to atomically multicast the request to the replicas
2.	receive(msg= ('PREPREPARE', v, n, m,d))	Backup accepts the PRE-PREPARE message, it enters the prepare phase by multicasting a PREPARE message to all other replicas. V: View number, n: Sequence number, m: message, d: digest
3.	receive(msg= ('PREPARE', v, n, m, i,d))	A replica (including the primary) accepts prepare messages; v: View Number, n: Sequence number, m: message, i: Replica ID, d: message Digest
4.	receive(msg= ('COMMIT', v, n, m, i,d))	Replicas accept COMMIT messages; v: View Number, n: Sequence Number; m: message; i: Replica ID; d; Digest
5.	receive(msg = ('View-Change',nv,m))	To receive the View-Change message during View change protocol. Nv: New view number, m: message
6.	receive(msg = ('New-View',v,np))	To receive the New-View message during view- change protocol. A backup accepts a new-view message for view v+1.
7.	initiatebackuptimer(m)	Utility function to start the timer of a backup when it receives the request from the client and the request has not been executed before. This will trigger View change protocol once the timer expires.
8.	configuredsend(MainMessage,destination)	Utility function to generate faulty messages to reproduce byzantine failure scenario.
9.	getMaxTorelentNumber()	Returns number of faulty replicas by (n-1)/3
10.	calculatedigest(data)	Uses md5 hashing technique and return the hash of the message
11.	isprimary()	To check if the current replica is a Primary
13.	execute(input)	Performs operation of the client.

Client APIs

S.No	API	Functionality(Overview)

.		
1.	sendrequest()	A client C requests the execution of state machine operation O by sending a <REQUEST,O,T,C> message to the primary.
2.	receive(msg= ('REPLY', v, t, i, command_id, r))	A replica sends the reply to the request directly to the client. The reply has the form <REPLY, v, t, c, i, r> where v is the current view number, t is the timestamp of the corresponding request, i is the replica number, r and is the result of executing the requested operation
3.	send_liveness_result(pv,nv)	Sends the Liveness state to Test.da

Driver APIs

S.No.	API	Functionality(Overview)
1.	receive(msg=('Test-done',is_syystem_safe,is_system_live ,totalmessagesinvolved,view_changed_happened))	Receives Test done from the testing framework and with all the values from testing framework.
2.	receive(msg= ("Stats",avgproctime,walltime))	Receives Stats from the comtroller.da and with all the values from it.
3.	run()	Main method which performs multiple operations like setting up each of the replicas, clients, testing and populating collected stats into a CSV

Test APIs

S.No.	API	Functionality(Overview)
1.	receive(msg= ('Liveness',pv,nv,clientID))	Receives Liveness message from the client with values = pv: previous view number; nv: new view number; ClientID: The client who has replied
2.	receive(msg= ("ReplicaInfo",executedlist,messageloglen,pid)):	Receives ReplicaInfo from each replica with parameters
3.	liveness_check():	Method which checks for liveness based on message received by it . Using above mentioned method
4.	safety_check()	Method which checks for safety by comparing the results generated are in order or not.

PART 3: IMPLEMENTATION

We have implemented PBFT in DistAlgo language. The code is present at [6]. We have used psutil[8] package to find the memory utilization value by processes and Controller.da file was taken from DistAlgo repo [7] for performance benchmarking with some changes done to it. We have 5 classes Driver, Client, Replica, Test, and Controller.

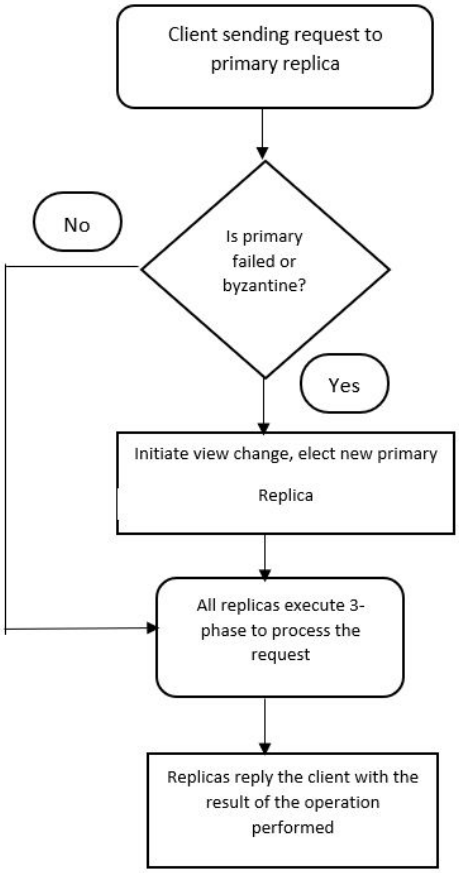
- **Language:** DistAlgo
- **Development Effort**: It took ~90 man hours to translate the algorithm given in paper[1] to working code. Extra efforts went into testing correctness part.
- **Lines of code:** The total lines of code are
    1. Client.da: LOC- 67 (Without Comments), 103 (With Comments)
    2. Driver.da: LOC- 113 (Without Comments)
    3. Replica.da: LOC- 200 (Without Comments), 345 (With Comments)
    4. Test.da: LOC- 45 (Without Comments), (With Comments)

Challenges Faced During Implementation

a. Faced a lot of difficulties designing a system that handles multiple primary replica failure scenarios: So according to the paper, there is no restriction as to how many times a primary replica might fail. Whenever a primary replica fails, the system should initiate a new-view where a new primary replica would get elected and the normal execution flow resumes. This cycle can get repeated for any number of times. But to simulate this environment, we had to implement two concepts termed checkpointing and garbage collection as depicted in the paper, which we thought was out of our scope after umpteen efforts from us. So for now, to the simulated view-change scenario, we tried to fail the primary replica only in the first instance, and later once new primary replica gets elected after the view change, we haven't again introduced a scenario where the newly elected primary replica fails again. In a nutshell, the following depicts what we have done and what more could have been done

WHAT WE HAVE DONE

(Primary replica failing only once)



WHAT MORE COULD HAVE BEEN DONE

(Primary replica failing multiple times)

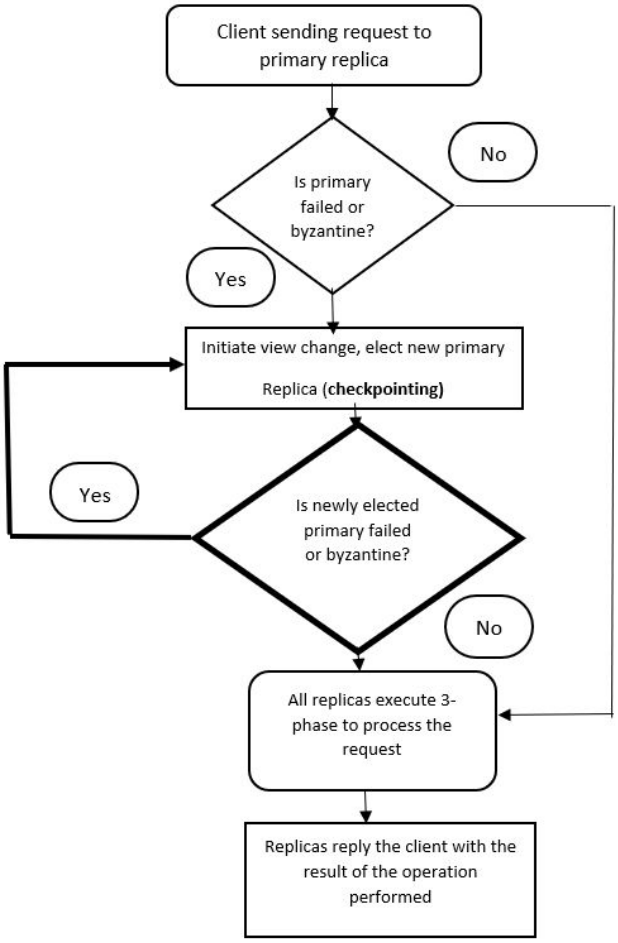


Figure 2: Flow Diagram representing what we did and what more could be done

b. Faced Issues while building a stable code for higher inputs like 50 replicas,70 clients with 2-10 operations each: This was the trickiest part. So we gradually started increasing the scale of the inputs and for some reason, our program was on the receiving end of a deadlock on many occasions. Our initial implementation encompassed many await statements inside the receive handlers and we suspected that this might be a potential issue for the problem. We tried to remove all the await statements inside the receive handler and it performed better. But, still, in some instances, the program was not terminating gracefully which we would ponder over.

Interesting Findings

a. "maximum recursion depth exceeded "OS Runtime error: So here we tried to play around a bit with the inputs. So as we increased the number of replicas and number of clients along with the number of operations, the number of messages exchanged has grown exponentially, which has to lead to an OS “maximum recursion depth exceeded error”. So below was the workaround we had used.
Workaround: “import sys
sys.setrecursionlimit(100000)”

b. ”Message too long to send” error: So we tried to send a message log which encompassed about some thousands of messages and eventually we faced this error. Had to remove redundant information in the log so as to transport them.

PART 4: TESTING AND EVALUATION

Num.of Replicas	Num. of Clients	Num. of Operations	network delay	Replica Delay	Byzantine failures	has Primary Filed?	View - Changed?	Throughput	Total Run Time	Clock Time	time per process	is Safe?	Is Live?	replica messages	Memory Utilizatio
5	4	5	0	0	0	FALSE	FALSE	91.98922703	0.232376575	0.229491	0.9375	Yes	Yes	1094	0.243842
10	8	4	0	0	0	FALSE	FALSE	29.68002394	1.128023386	1.118418	7.515625	Yes	Yes	6684	0.24574
15	8	2	0	0	0	FALSE	FALSE	14.77229761	1.132915974	1.122948	7.71875	Yes	Yes	7382	0.247005
25	6	4	0	0	0	FALSE	FALSE	5.500395632	4.424161911	4.421721	32.203125	Yes	Yes	30559	0.250655
22	8	3	0	0	0	FALSE	FALSE	7.372302874	3.352163076	3.337262	24.671875	Yes	Yes	23718	0.249682

Fig 1:Statistics with 0 byzantine replicas and with no network delay

Num.of Replicas	Num. of Clients	Num. of Operations	network delay	Replica Delay	Num.of Byzantine failures	has Primary Filed?	View - Changed?	Throughput	Total Run Time	Wall Clock Time	Avg.Run time per process	is Safe?	Is Live?	Total replica messages exchanged	Process Memory Utilization percentage
5	4	5	0	0	1	TRUE	TRUE	0.985244	20.31434	20.31126	1.421875	Yes	Yes	971	0.243306
10	8	4	0	0	3	TRUE	TRUE	1.487017	21.70311	21.68106	12.07813	Yes	Yes	5088	0.246178
15	8	2	0	0	4	TRUE	TRUE	0.747437	21.7386	21.7231	13.51563	Yes	Yes	3957	0.245934
25	6	4	0	0	8	TRUE	TRUE	0.604961	41.21383	41.18673	157.9531	Yes	Yes	21592	0.25231
22	8	3	0	0	7	TRUE	TRUE	0.946775	26.60486	26.56688	50.125	Yes	Yes	14787	0.250315

Fig 2:Statistics with 1<=k<=f byzantine replicas including primary replica and with no network delay

Num.of Replicas	Num. of Clients	Num. of Operations	network delay	Replica Delay	Num.of Byzantine failures	has Primary Filed?	View - Changed?	Throughput	Total Run Time	Wall Clock Time	Avg.Run time per process	is Safe?	Is Live?	Total replica messages exchanged	Process Memory Utilization percentage
5	4	5	0	0	1	FALSE	FALSE	88.3414	0.237367	0.235076	0.984375	Yes	Yes	957	0.244036
10	8	4	0	0	3	FALSE	FALSE	34.95164	1.126973	1.115687	6.71875	Yes	Yes	22546	0.245156
15	8	2	0	0	3	FALSE	FALSE	1.530124	10.47861	10.48009	5.40625	Yes	Yes	4206	0.247784
25	6	4	0	0	7	FALSE	FALSE	6.22117	4.629724	4.602189	31.75	Yes	Yes	22651	0.249877
22	8	3	0	0	7	FALSE	FALSE	8.064418	2.616994	2.607136	17.67188	Yes	Yes	12604	0.248903

Fig 3:Statistics with 1<=k<=f byzantine replicas NOT including primary Replica and with no network delay

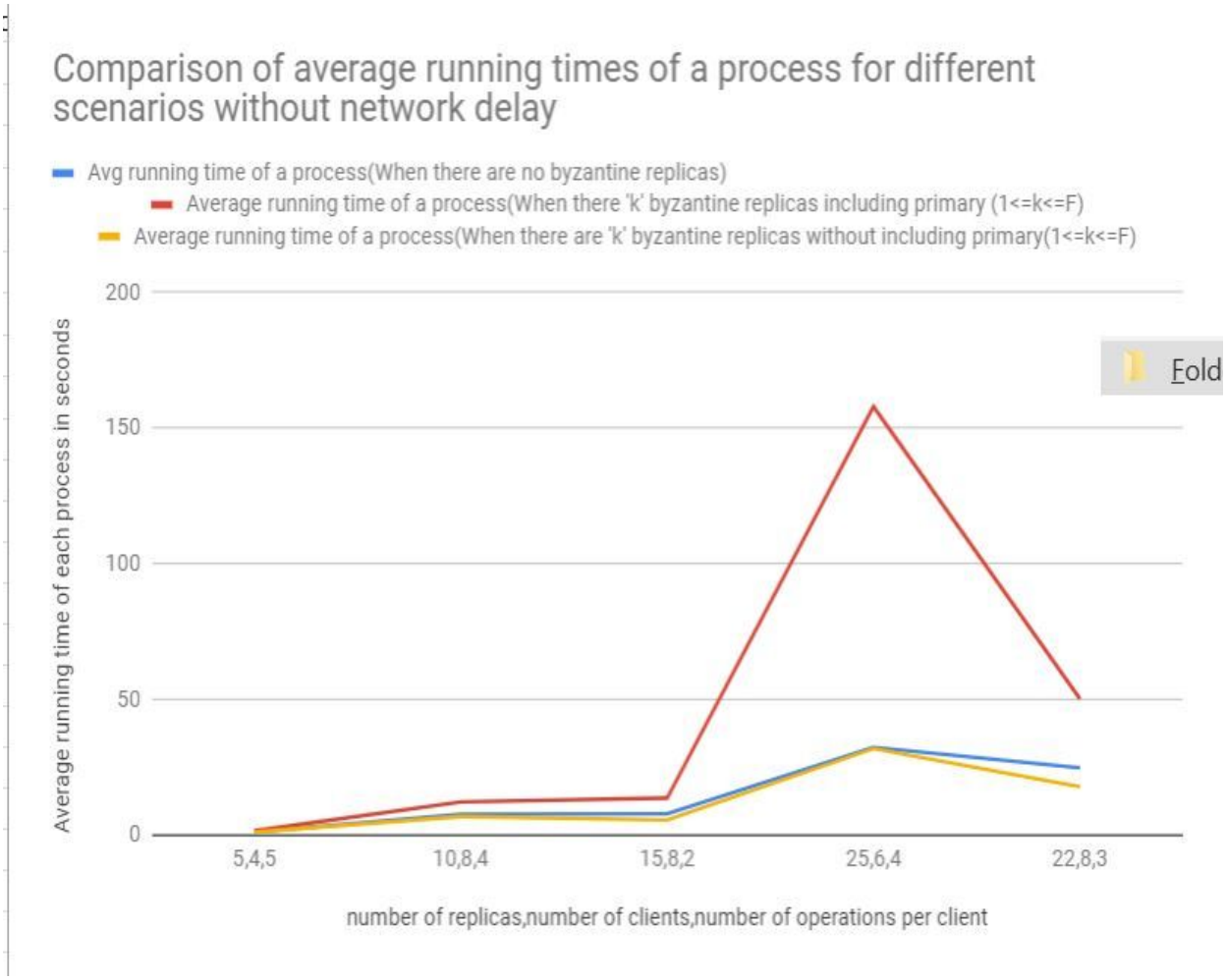


FIG 4:Comparison of average running times of a process for different scenarios without network delay

Observations: The most obvious conclusion from the above graph is that whenever view change occurs, it takes a fair bit of time to resume the normal case operation and hence the average process time is a bit on the higher side when primary replica fails(view-change occurs)

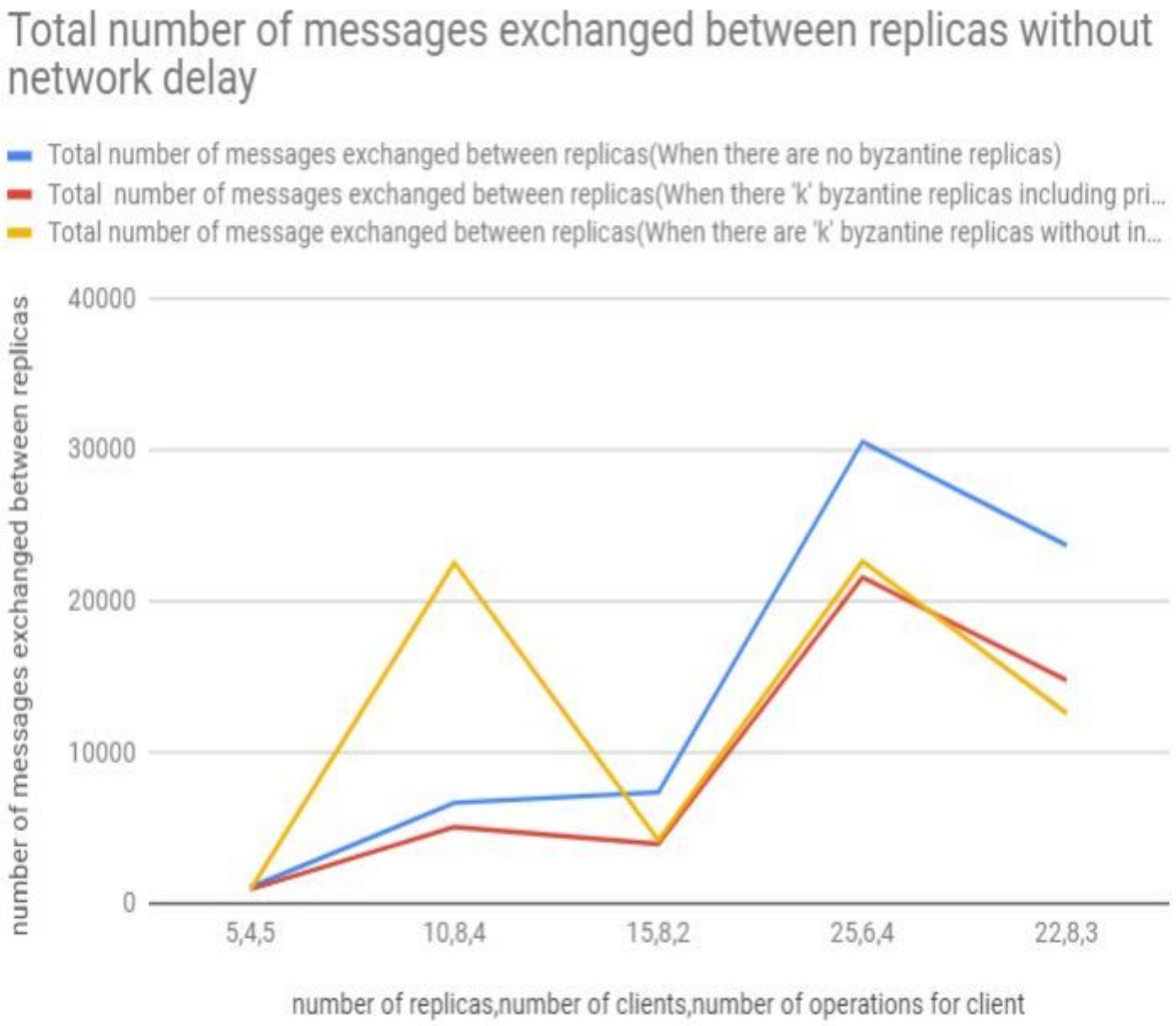


Fig 5:Total number of messages exchanged between replicas without network delay

Observation: As the number of clients and replicas increase, we can see that the number of messages exchanged among replicas will also increase.When view change occurs,since the system stops accepting messages so there is considerable drop in the number of messages exchanged when view-change occurs.But when there are no failures ,all messages become valid and there is a considerable increase in the number of messages stored in the logs.

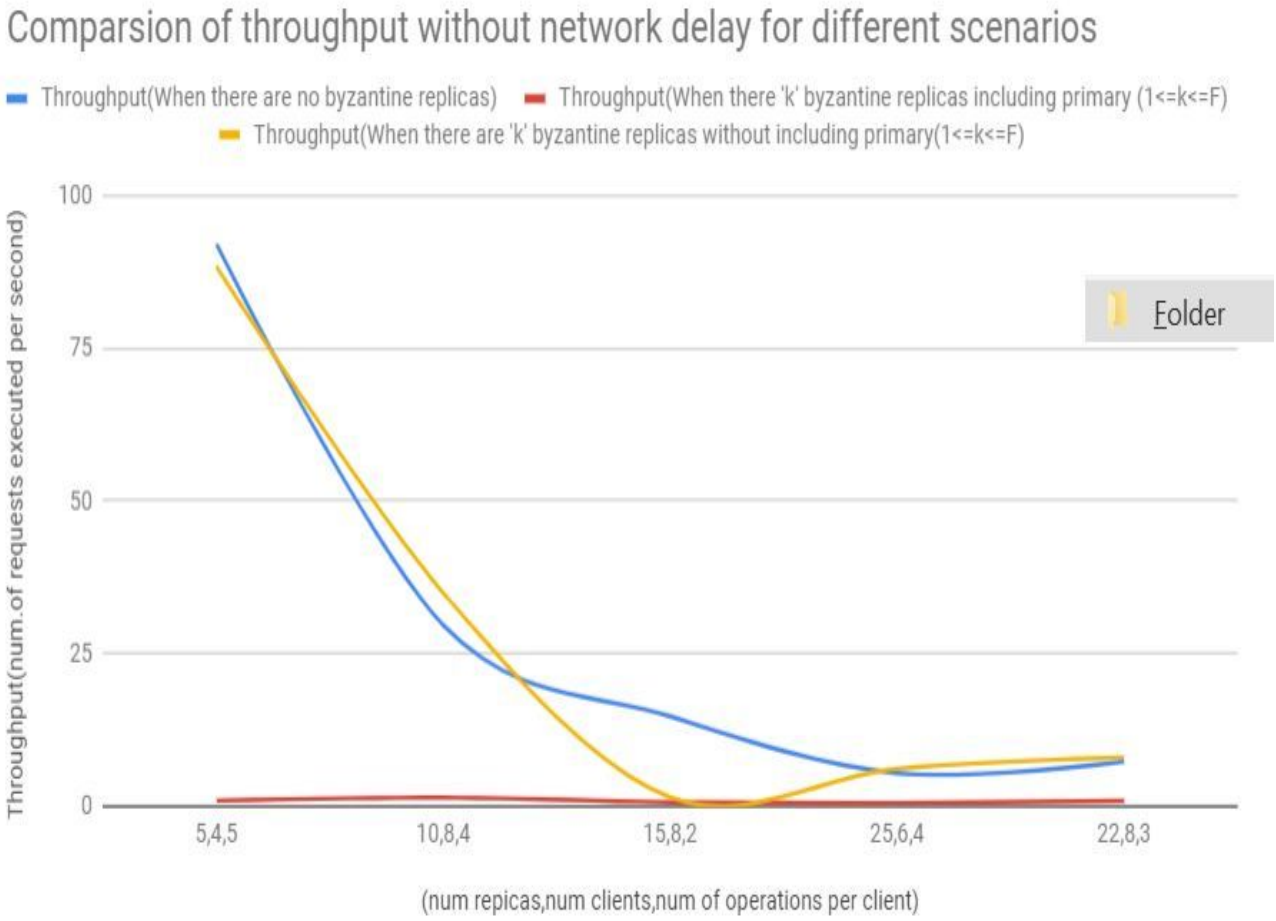


Fig 6:Comparison of throughput for different scenarios without network delay

Observation: A general trend follows over here where as the the number of operations,clients and replicas increase,it takes that extra bit of time to reach consensus and process the request and hence the throughput(number of requests executed per second)decreases.

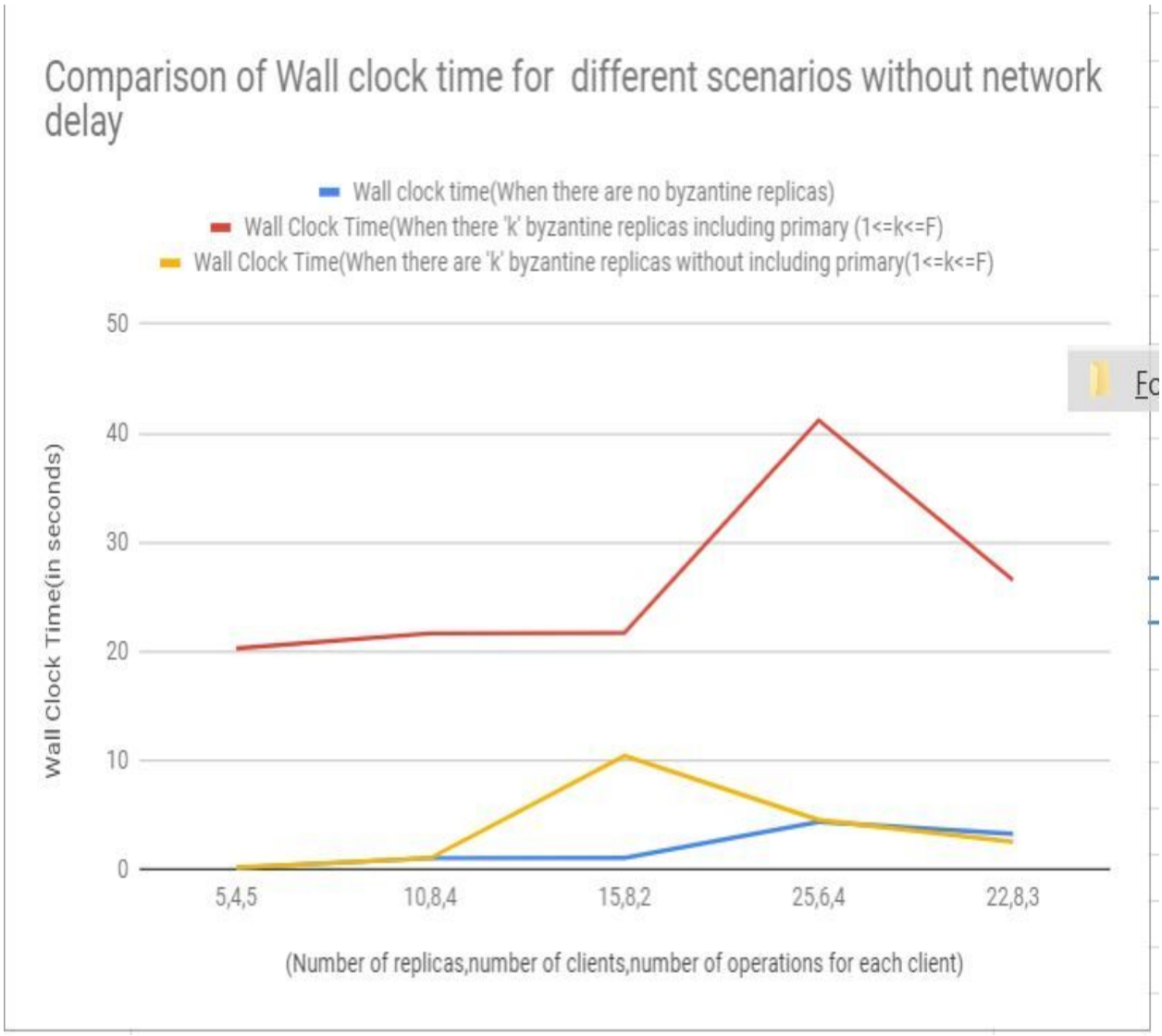


Fig 7: Comparison of Wall Clock time for different scenarios without network delay

Observation: The process of changing the view delays the process of executing requests and reaching consensus and hence its wall clock time is on the higher side when compared to other scenarios

Num.of Replicas	Num. of Clients	Num. of Operations	network delay	Replica Delay	Num.of Byzantine failures	has Primary Filed?	View - Changed?	Throughput	Total Run Time	Wall Clock Time	Avg.Run time per process	is Safe?	Is Live?	Total replica messages exchanged	Process Memory Utilization percentage
5	4	5	0.05	0	0	FALSE	FALSE	52.63189	0.385985	0.393008	1.171875	Yes	Yes	1100	0.242576
10	8	4	0.05	0	0	FALSE	FALSE	2.999841	10.68618	10.67342	6.4375	Yes	Yes	5880	0.245934
15	8	2	0.05	0	0	FALSE	FALSE	1.549005	10.35718	10.35129	7.015625	Yes	Yes	6960	0.247541
25	6	4	0.05	0	0	FALSE	FALSE	2.107036	11.44027	11.43052	28.59375	Yes	Yes	28050	0.252067
22	4	3	0.05	0	0	FALSE	FALSE	7.202939	1.762727	1.755331	11.65625	Yes	Yes	11819	0.24608

Fig 8: Statistics with zero byzantine replicas and with a network delay of 0.05

Num.of Replicas	Num. of Clients	Num. of Operations	network delay	Replica Delay	Num.of Byzantine failures	has Primary Filed?	View - Changed?	Throughput	Total Run Time	Wall Clock Time	Avg.Run time per process	is Safe?	Is Live?	Total replica messages exchanged	Process Memory Utilization percentage
5	4	5	0.05	0	1	TRUE	TRUE	0.973927	24.18778	24.18135	1.3125	Yes	Yes	971	0.242479
10	8	4	0.05	0	3	TRUE	TRUE	1.491498	29.48031	29.47148	12.07813	Yes	Yes	5088	0.245886
15	8	2	0.05	0	4	TRUE	TRUE	0.691968	30.50856	0	0	Yes	Yes	5711	0.246129
25	6	4	0.05	0	8	TRUE	TRUE	0.614683	47.04216	46.59766	165.8281	Yes	Yes	21575	0.250461
22	4	3	0.05	0	7	TRUE	TRUE	0.463873	31.09139	31.0697	51.9375	Yes	Yes	8463	0.248125

Fig 9:Statistics with 1<=k<=f byzantine replicas including primary replica and network delay of 0.05

Num.of Replicas	Num. of Clients	Num. of Operations	network delay	Replica Delay	Num.of Byzantine failures	has Primary Filed?	View - Changed?	Throughput	Total Run Time	Wall Clock Time	Avg.Run time per process	is Safe?	Is Live?	Total replica messages exchanged	Process Memory Utilization percentage
5	4	5	0.05	0	1	FALSE	FALSE	49.1218	3.240801	3.237912	0.875	Yes	Yes	957	0.243452
10	8	4	0.05	0	3	FALSE	FALSE	39.12579	3.445612	3.436883	3.578125	Yes	Yes	24654	0.246129
15	8	2	0.05	0	4	FALSE	FALSE	19.5587	3.093285	3.074432	6.703125	Yes	Yes	5662	0.24647
25	6	4	0.05	0	8	FALSE	FALSE	7.265325	6.426255	6.398948	28.73438	Yes	Yes	21536	0.25085
22	4	3	0.05	0	7	FALSE	FALSE	9.839427	2.930763	2.906	10.75	Yes	Yes	8415	0.248952

Fig 10:Statistics with 1<=k<=f byzantine replicas NOT including primary replica and network delay of 0.05

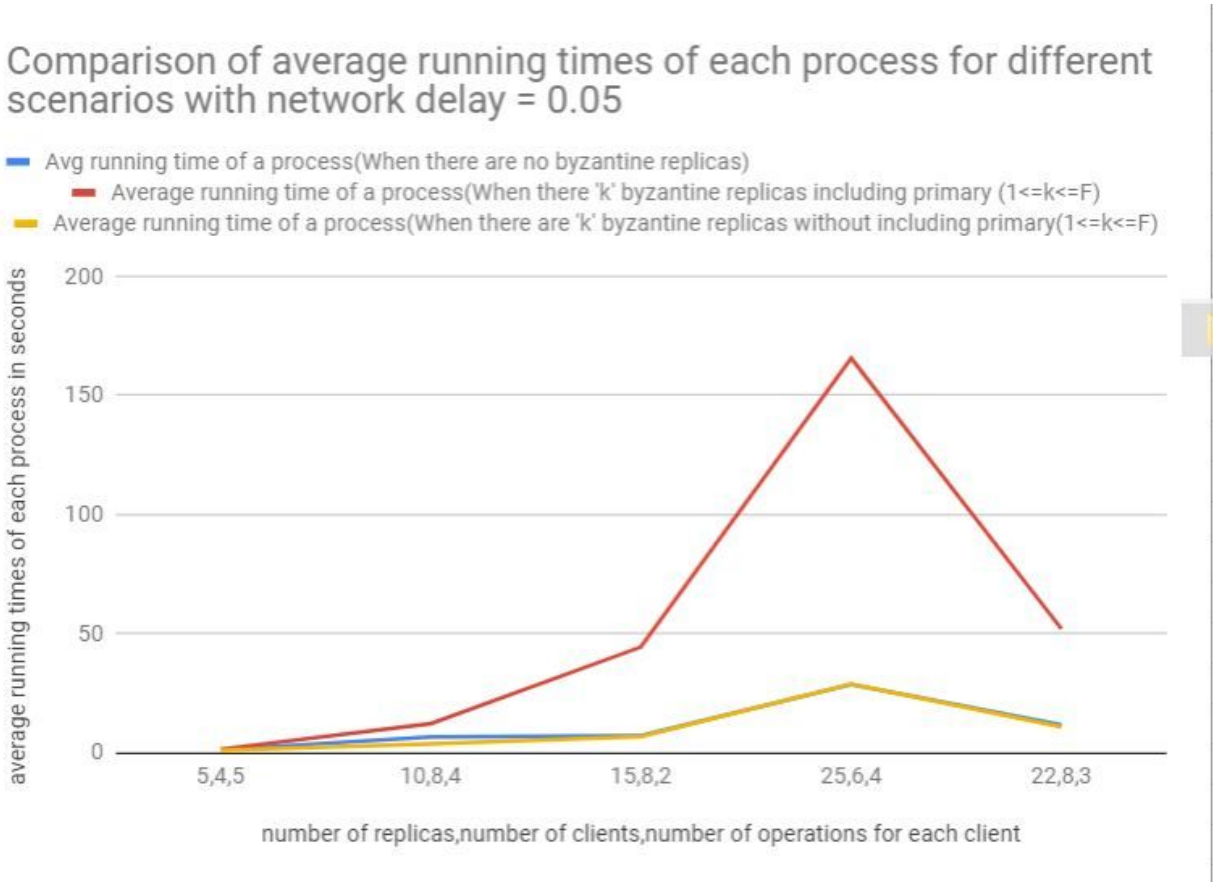


Fig 11:Comparison of average running times of each process for different scenarios with network delay of 0.05

Observations: Running times increase a lit bit for a process in all scenarios when network delay is introduced but it's significant when it comes to view-change scenario because of the additional view change process overhead

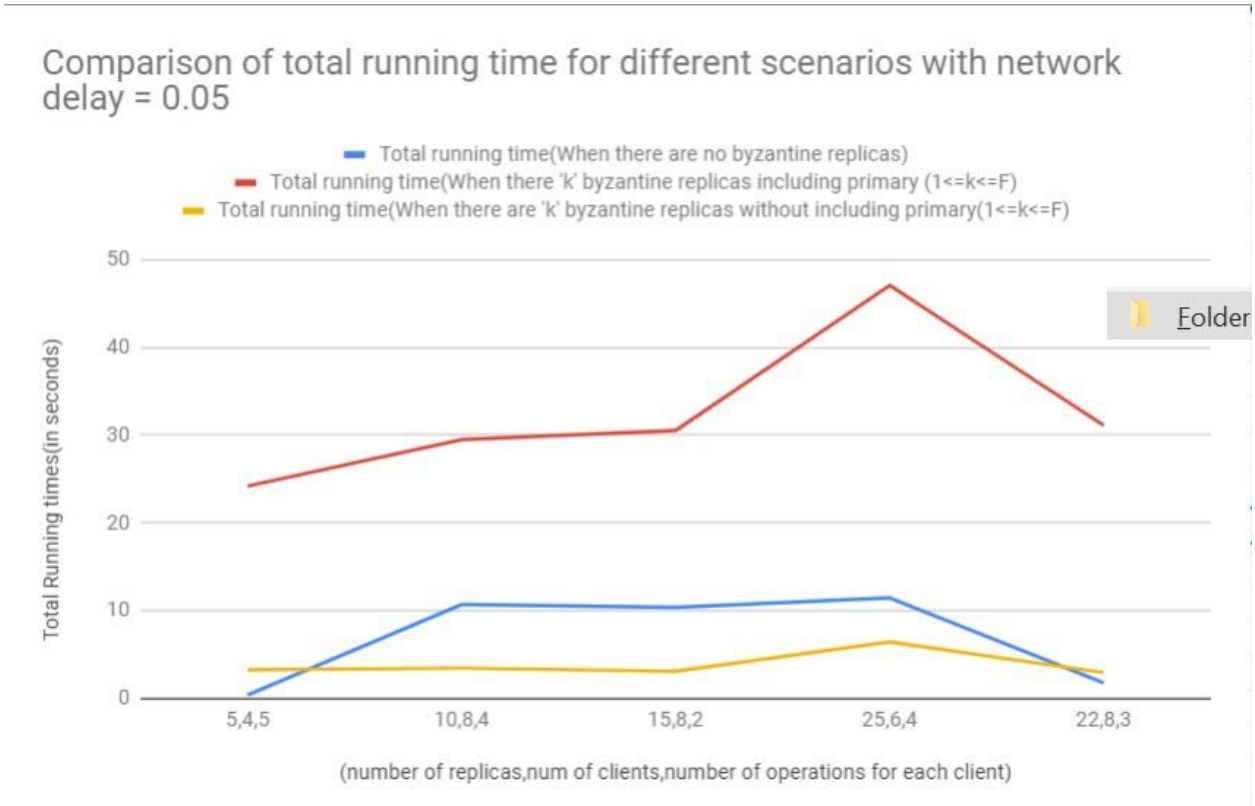


Fig 12:Comparison of total running times of each process for different scenarios with network delay of 0.05

Observations: Same trend follows as that of average running times for process.

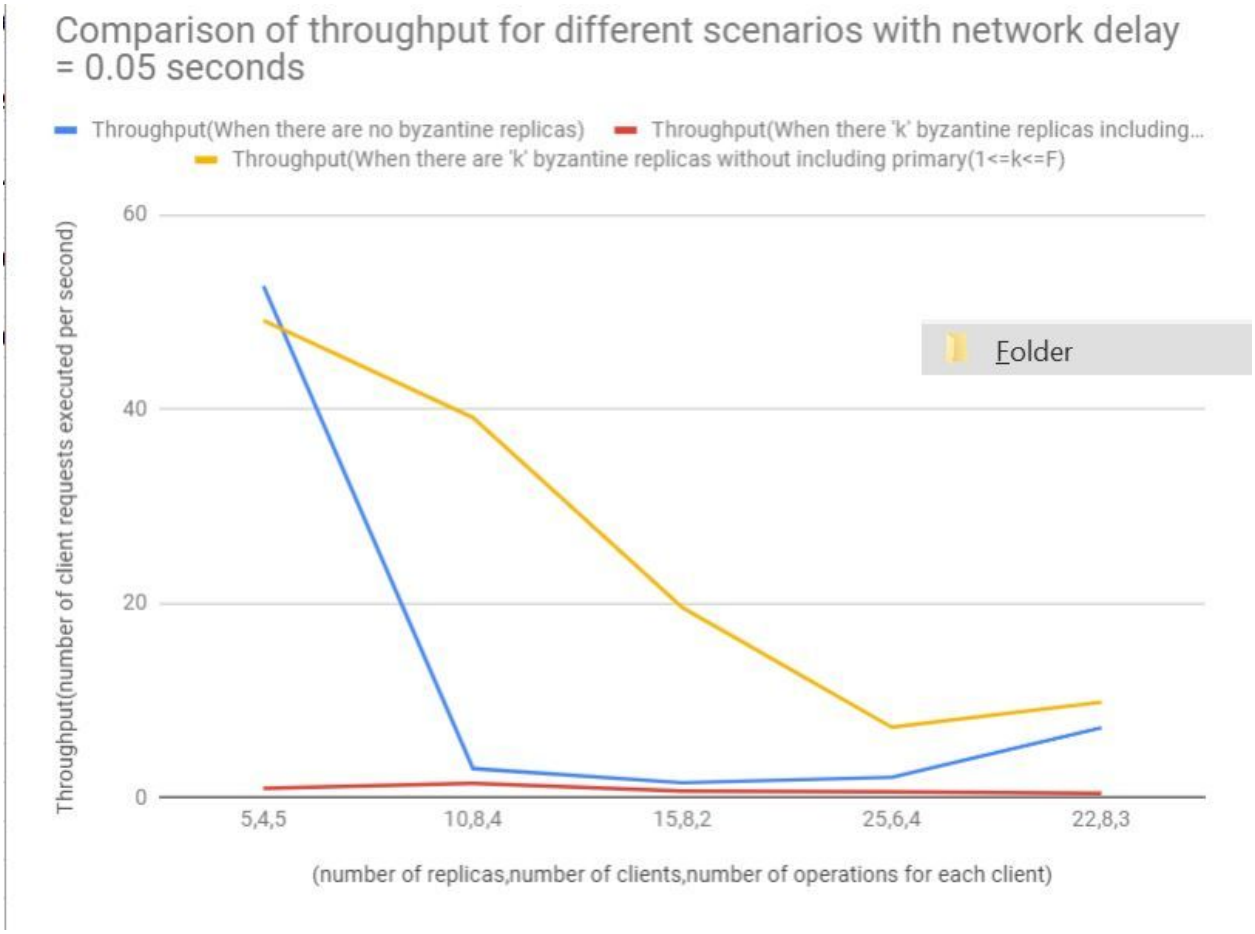


Fig 13:Comparison of throughput of each process for different scenarios with network delay of 0.05

Observations: Throughput remains same when view-change phenomena occurs where as it increases with smaller number of requests and decreases with higher number of requests for non-view change scenarios

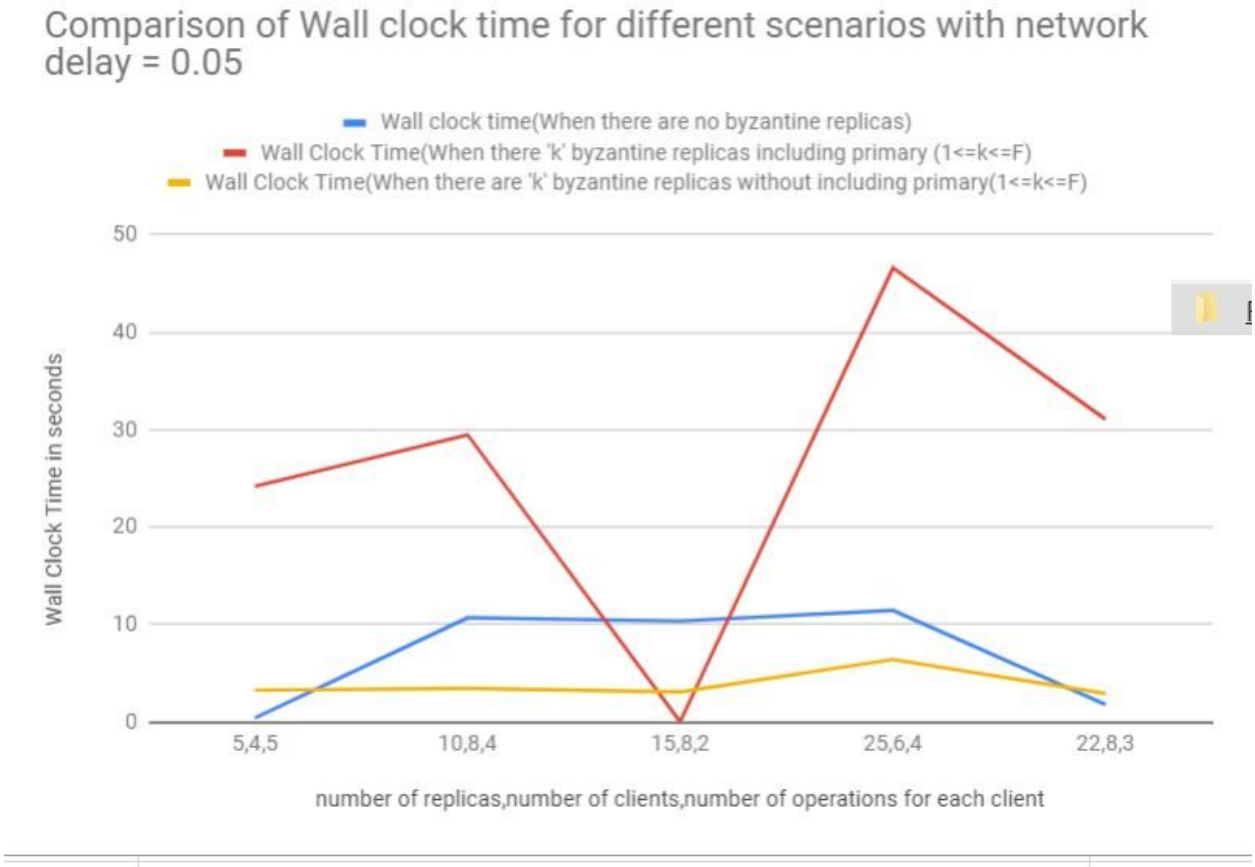


Fig 14:Comparison of Wall Clock time of each process for different scenarios with network delay of 0.05

Observations: Running times increase a lit bit for a process in all scenarios when network delay is introduced but it's significant when it comes to view-change scenario because of the additional view change process overhead.When view change doesn't occur properly,the program did not terminate gracefully so ,a wall clock time of 0 is an aberration

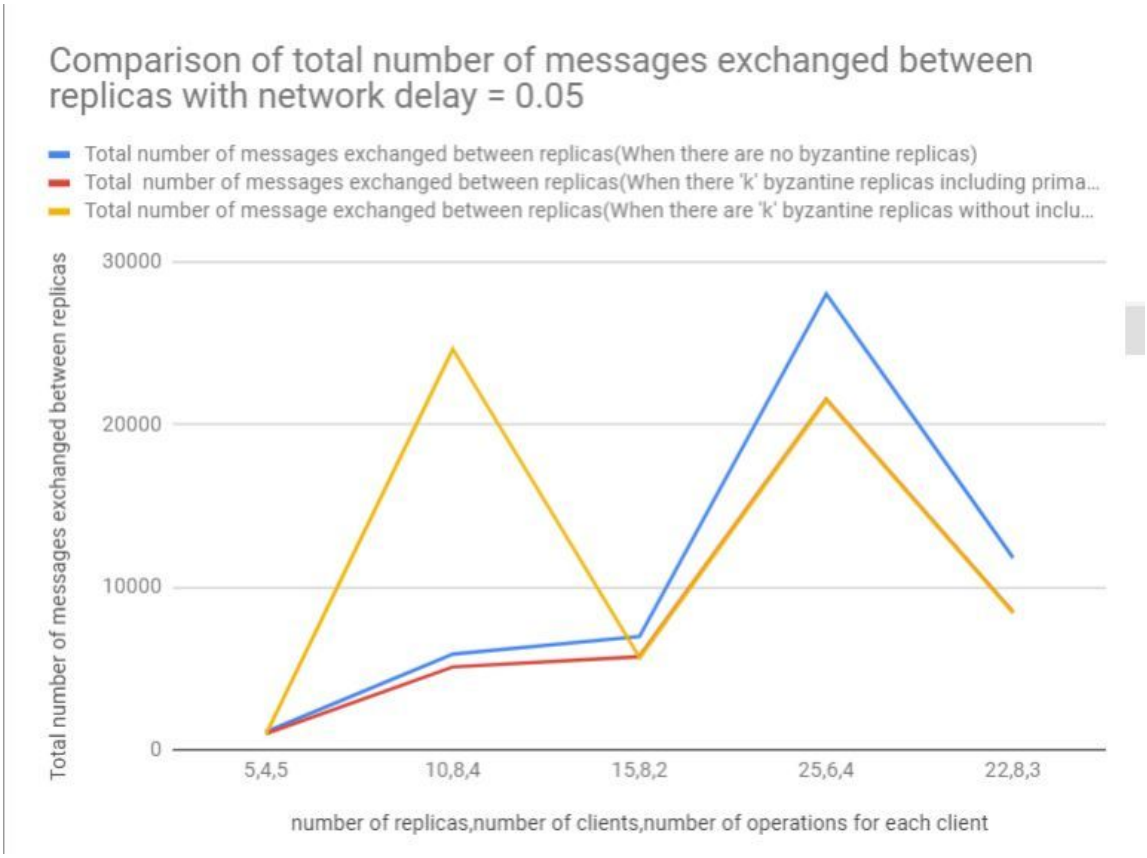


Fig 15:Total number of messages exchanged among replicas with a network delay of 0.05

Observation: Same trend followed as that of Total number of messages exchanged among replicas without a network delay.But the magnitude of the number of messages is less when compared to total number of messages exchanged among replicas without a network delay = 0.A bit of redundancy is eliminated.

FAILURE SCENARIO (DEADLOCK):

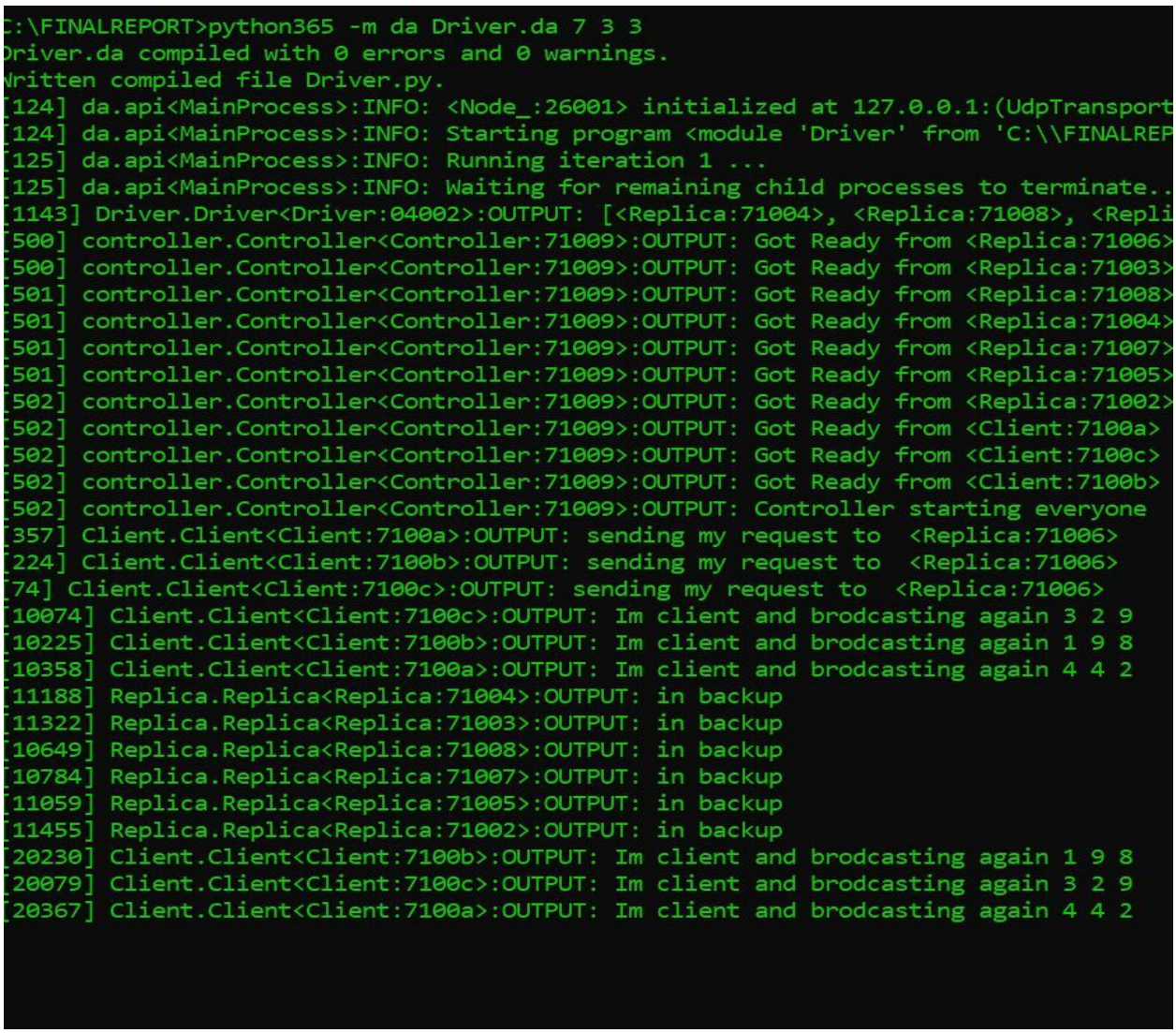


Fig: Here number of replicas 3*f + 1 = 7; but we have selected “4” replicas as byzantine and we can see that the program encountered a deadlock because the maximum number of faulty replicas can be “2” according to 3f+1 rule.

Instructions for running programs on test

Above tests and results were captured using the below command

python -m da Driver.da num_replicas num_clients num_requests timeout netwrok_delay replica_delay simulate_byzantine

1. **num_replicas:** Number of replicas (default set to 4)
2. **num_clients:** Number of Clients (default set to 1)
3. **num_requests:** Number of requests by Client (default set to 1)
4. **timeout :** Configurable Timeout (default set to 25)
5. **Netwrok_delay:** Network delay in the system (default set to 0)
6. **Replica_delay:** Delay within Replicas(default set to 0)
7. **Simulate_byzantine:** Explained in detail below (default set to 0)

To simulate the byzantine environment where either a Primary is byzantine or any Backup Replica is byzantine, while passing arguments during running of code, you can pass the 7th argument as either

0: Represents that none will be byzantine.

1: Represents that Primary will be byzantine and some other may or may not be byzantine. This case will initiate View Change Protocol and then result is achieved after a View Change has happened.

2: Represents that any Backup may be byzantine(not Primary). In this case result received by the client is correct as per PBFT Algorithms

Running the above code will generate the performance.csv which is used to generate the plots by Google Graphs. Performance.csv is generated for each type of run.

PART 5: REFERENCES

1. Castro, M., Liskov, B.: Practical Byzantine Fault Tolerance. In: Proceedings of OSDI 1999, New Orleans, LA (Feb. 1999) http://pmg.csail.mit.edu/papers/osdi99.pdf
2. WHAT IS PRACTICAL BYZANTINE FAULT TOLERANCE (PBFT) https://crushcrypto.com/what-is-practical-byzantine-fault-tolerance/
3. The Byzantine Generals' Problem https://medium.com/all-things-ledger/the-byzantine-generals-problem-168553f31480
4. Experiments with pBFT https://github.com/gdanezis/pybft
5. Sample implementation of PBFT consensus algorithm https://github.com/bigpicturelabs/consensusPBFT
6. Pbft in distalgo https://github.com/unicomputing/pbft-distalgo
7. This is the Python implementation of DistAlgo, a language for distributed algorithms https://github.com/DistAlgo/distalgo
8. Psutil package https://pypi.org/project/psutil/
9. From Viewstamped Replication to Byzantine Fault Tolerance http://pmg.csail.mit.edu/papers/vr-to-bft.pdf