CSE 535 - Asynchronous Systems

PERFORMANCE COMPARISON FRAMEWORK FOR DISTRIBUTED CONSENSUS ALGORITHMS

December 11, 2018

Radhika Dhawan, SBU ID# 112074050
Jatin Sood, SBU ID# 112079000
Rohit Aich, SBU ID# 112126618

Contents

List of Figures

I. Introduction

Distributed consensus algorithms are among the most widely used algorithms in the fields using distributed databases in an unreliable environment, where there might be a number of faulty processes which might abruptly crash or abort. There are a plethora of consensus algorithms present out in the world, and they employ different techniques to achieve consensus between participating systems. Currently (as described in more detail in the sub-section "State of the art:"), to the best of our knowledge, there is no single comparison framework to compare the performances (in terms of memory and time consumption, and responses to abrupt process failures) of any two distributed consensus algorithms. In the course of this project, we have aimed to create a framework that achieves this task. We show that by considerably minimal effort, any python or python-based implementations (we have used DistAlgo implementations here) of consensus algorithms can be integrated into the system, and subjected to a detailed and insightful performance comparison. We have developed a central logging system to log the performance data of each of these implementations, and subsequently use it to generate graphs and charts depicting the performances visually.

Furthermore, we have used our prototype and integrated a vraPaxos and Raft implementation in DistAlgo, and compared their performances for a few metrics like latency, CPU time and elapsed time, memory consumption etc. We have discerned interesting comparison results from the framework, and show how the behaviours of these algorithms change in response to varying input parameters and random process failures. As an addition to the performance comparison framework, our code can also log inter-process communications in a specifically formatted log file, which can subsequently be parsed by ShiViz tool to generate visualizations for the implementations. This part would help the users understand every process flow in detail.

GitHub link: `https://github.com/unicomputing/eval-consensus-py`.

I. A schematic diagram of the proposed solution:

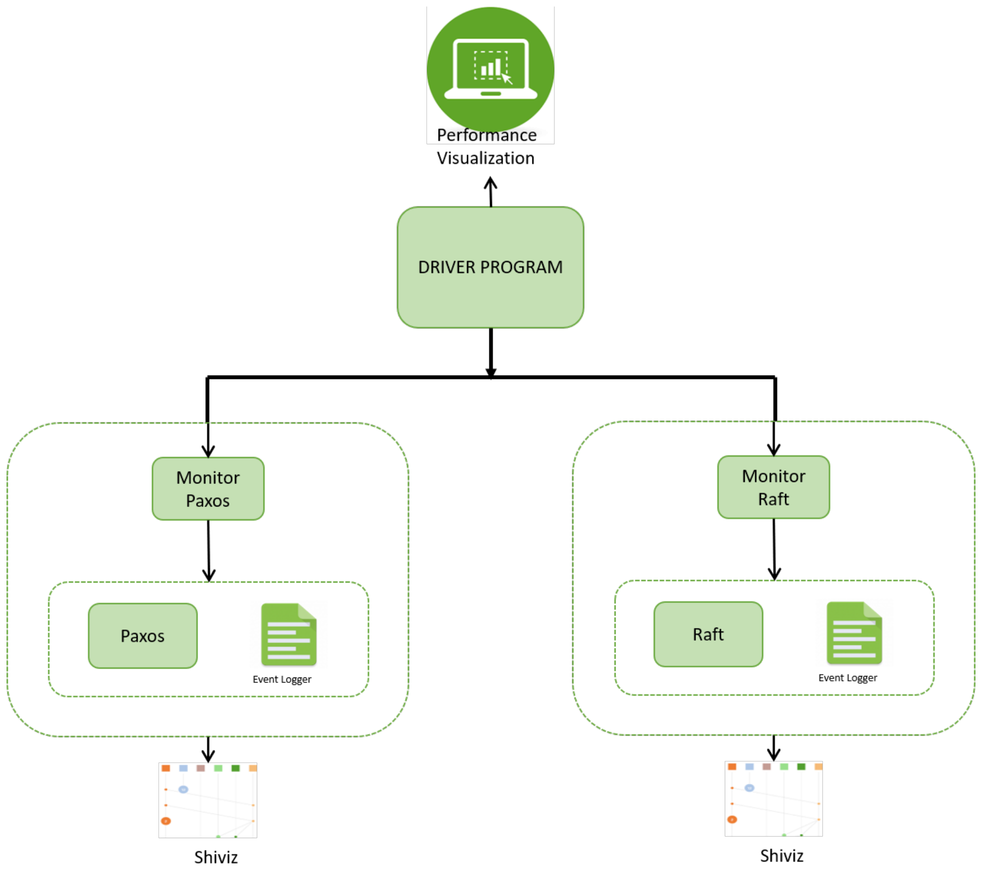


Figure 1: Architecture Diagram

II. State of the art:

We read a number of papers on performance comparison of distributed consensus algorithms to understand the available techniques and metrics for comparison. Urbán et. al. [1] introduces a comparison metric between two distributed consensus algorithms (one centralized and the other decentralized), and show how due to less contention, centralized algorithm performs better in some environments, in spite of the fact that the decentralized algorithm finishes in fewer communication steps. He compares the workload latency of CT and MR consensus algorithms. Also, in another paper on performance comparison between Hayashibar et al. [2] shows that how the two algorithms respond to failures and failure suspicions, and shows by comparing performances how the Paxos is more efficient after the first round of crashes. We have taken this idea of comparison, and have added a feature in our framework to check the performances after explicitly crashing one or more component systems in a distributed environment. Agrawal et. al. [3] compares the performance of the algorithms for byzantine agreement in distributed systems in terms of bit complexity, and show that for small networks ($n < 32$) and up to 10 percent faulty processes, the simple deterministic algorithm performs best, while for larger networks, pull-push is the best performing algorithm.

III. Problem statement:

The performance of consensus algorithms directly affect our daily lives. Our goal was to take a few famous and widely used consensus algorithms and compare their performances, and in the way develop a comparison framework, powered by a central logging system, that would be able to integrate many more such implementations and compare their efficiency for different metrics. Distributed consensus algorithms find a global usage in the fields mentioned below[7].

1. Synchronizing replicated state machines and making sure all replicas have the same (consistent) view of system state.
2. Electing a leader (e.g., for mutual exclusion) distributed, fault-tolerant logging with globally consistent sequencing.
3. Managing group membership.
4. Deciding to commit or abort for distributed transactions.

Of course, a detailed comparison scheme of such algorithms need a way to visualize the inter-process communications, especially when we are planning to abruptly crash a few processes and measure the fault tolerance capacity of the algorithms. With this goal, we have also enable our logger system to be able to log the inter-process communications in a specific format, which could be parsed by the ShiViz tool to generate an interactive visualization.

II. Design

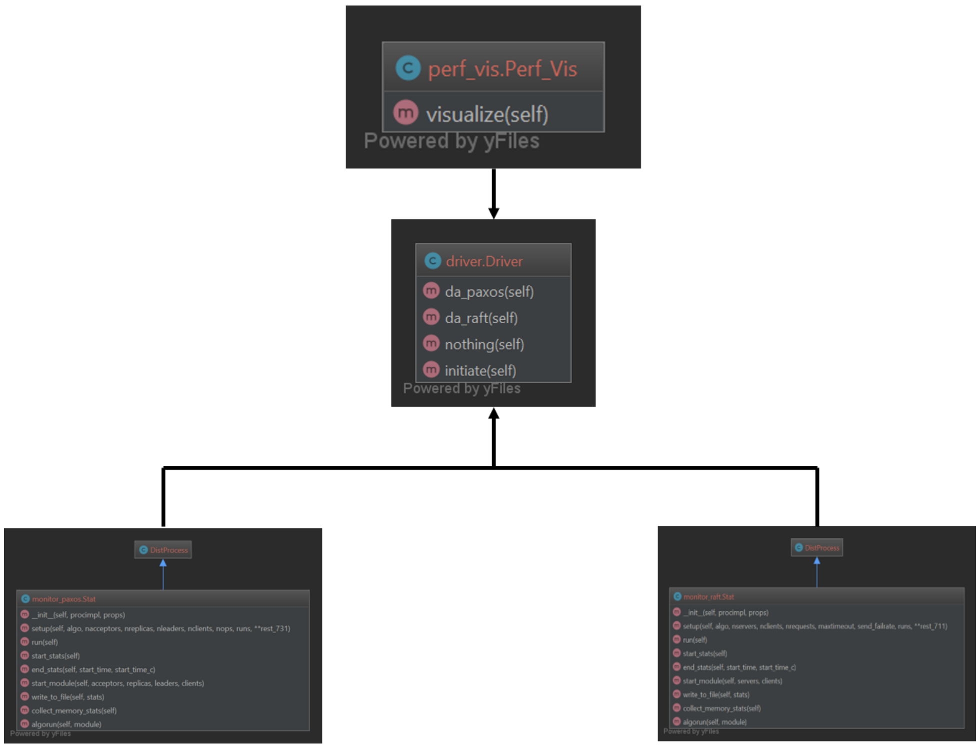

perf_vis.Perf_Vis
visualize(self)
Powered by yFiles
driver.Driver
da_paxos(self)
da_raft(self)
nothing(self)
initiate(self)
Powered by yFiles
DistProcess
monitor_paxos.Stat
__init__(self, procimpl, props)
setup(self, algo, nacceptors, nreplicas, nleaders, nclients, nops, runs, **rest_731)
run(self)
start_stats(self)
end_stats(self, start_time, start_time_c)
start_module(self, acceptors, replicas, leaders, clients)
write_to_file(self, stats)
collect_memory_stats(self)
algorun(self, module)
Powered by yFiles
DistProcess
monitor_raft.Stat
__init__(self, procimpl, props)
setup(self, algo, nservers, nclients, nrequests, maxtimeout, send_failrate, runs, **rest_711)
run(self)
start_stats(self)
end_stats(self, start_time, start_time_c)
start_module(self, servers, clients)
write_to_file(self, stats)
collect_memory_stats(self)
algorun(self, module)
Powered by yFiles

Figure 2: API Design Chart

Figure 1 shows a schematic architecture diagram of our system design. It has three primary parts, namely the monitor program, consensus algorithms implementations and their respective driver programs. The chain of execution starts at monitor program which is primarily responsible for invoking the drivers of the consensus algorithm implementations. We include the best state-of-the-art DistAlgo implementations of vrPaxos [5] and Raft [5] identified in previous section in our study of comparing performance. We implement driver programs to test these implementations as well as measure their performance metrics. Once we have the performance data of multiple runs from both the implementations, we use it to generate performance plots and observe interesting comparisons between the two implementations. We also modify the implementations to support logging in the format which is supported by ShiViz in order to generate flow sequence of the consensus algorithms.

I. Driver Programs

- We use separate drivers for each of our implementations. In our $figure1$, implementations A and B represent Paxos and Raft implementations in DistAlgo respectively. The corresponding drivers (A..B), again implemented in DistAlgo, control these implementations and capture CPU time and memory usage details of each implementation by varying the input parameters like number of runs, number of participating processes, number of requests per client, etc.
- The driver programs log these performance numbers which can be read by the monitor program for generating performance comparison and trends.

II. Monitor Program

- We use a single monitor program that is responsible for starting all the drivers programs in the system. Also, this monitor program is responsible to read all the performance logs that the drivers generate as outputs.
- From those captured logs, we use the monitor program to read performance statistics to create trends and plots that help us to compare the time and space complexities of both the implementations.
- This decoupled architecture helps us to maintain an abstraction between the main monitor program and the end implementations. The modular nature would ensure that we could insert any other algorithms or implementations in our framework. We would just have to write a DistAlgo driver for the program, and our monitor would be able to compare performances their performances.

III. Consensus Implementations

- In order to gather the information on memory consumed by an implementation, we have to tweak the implementation. Although, this is not a change which we desire, but gaining real-time information on consumed memory is an action which could be triggered only from inside of the process.
- For each of the agent (for example, leader, acceptor, etc. in case of vrPaxos) in consensus implementation, we define a minimalistic function to fetch the current memory consumed by the process and report it to the driver program via a message.
- We trigger this function multiple times within the agent process to gain as much knowledge of the consumed memory as possible. These stats are then collected, averaged and aggregated in the driver program.
- For visualizing flow sequence of a consensus algorithm, we have to further tweak the implementations. We keep a vector clock associated with each agent process of the consensus algorithm.
- For visualizing each inter-process communication in the consensus algorithm, we log the vector clock values of sender and receiver process in format required by shiviz. This is again done in minimilastic way.

III. Implementation

I. Language Consideration

We started with Python, and found that due to the inbuilt send and receive functions, DistAlgo is far more apt for our implementation. In fact, we chose to use the DistAlgo implementations of vraPaxos [4] and

Raft [5] from GitHub, because of the easiness of understanding. It is easier to create multiple DistAlgo processes as the syntax is simplified. Second, the inter-process communication between the processes is simplified by the library utility functions send, receive, sent and received. Implementing similar functions in python would have required considerable effort.

II. Environment

Platform: Microsoft Windows 10, GNU Linux (Linux Mint 18)
Language: DistAlgo version 1.1.0b12 & Python 3.6.5
Dependencies: Seaborn 0.8.1
Man hours: 4 weeks for a team of three.

Please follow the instructions in README to understand how to run the performance framework (**Github link:** *Readme*). The instructions are also mentioned in the *Testing and Evaluation* section later in the report.

III. Multi-Paxos and Raft Implementation

We have picked the DistAlgo versions for both the consensus algorithm implementations which are using in-built process communication utilities. Each process in the implementations is a DistAlgo process. Send and receive in-built DistAlgo functions are used for inter-process communication.

The paxos DistAlgo implementation contains clients processes which send commands to be executed to the replica processes. (**GitHub link:** *vrPaxos*) The replica process places the command in slot. Each replica proposes command for the slot to all the leader processes. Each leader process creates a scout process and commander process which are used for phase 1 and phase 2 of the two-phase commit respectively. The commander on behalf of leader, then sends the decision to all the replicas for that slot, after which all replicas execute same command for that slot and consensus is reached.

The Raft[6], (**GitHub link:** *Raft*) is a consensus algorithm that is easier to understand and equivalent to Paxos in fault-tolerance and performance, but its structure is different from Paxos. The difference is that it's decomposed into relatively independent subproblems, and it cleanly addresses all major pieces needed for practical systems. Raft is similar in many ways to existing consensus algorithms (most notably, Oki and Liskov's Viewstamped Replication), but it has several novel features:[6]

- Strong Leader
- Leader Election
- Membership Changes

Both the implementations have a separate versions with crashing one leader.
GitHub links: *Paxos with Leader crash* and *Raft with Leader crash*.

IV. Drivers & Monitors

The driver is written in DistAlgo. (**Github link:** *Driver*) The main job of the driver program is to invoke monitor programs which run consensus implementations. The driver program is designed like a flexible switch which takes input from user for the specific implementation that user wants to test. After selection, the user can enter the number of number of various agent processes of distributed consensus, like servers, clients, replicas, etc. These parameters are passed to the driver programs for invoking the implementations with required configurations.

The monitor programs are written in DistAlgo as well. (**Github links**: *vrPaxos Monitor*, *Raft Monitor*) The monitor DistAlgo process runs multiple runs of its respective consensus algorithm, in which it creates all the DistAlgo processes from the consensus implementation which act as an agent in achieving consensus. These are namely clients, replicas, leaders and acceptors in case of vrPaxos. In case of raft, the monitor invokes clients and servers DistAlgo processes.

Monitor waits for the clients to terminate and sends the terminating signal to all other DistAlgo processes created by it. The monitor processes also keep track of latency and cpu time consumed by the implementation runs. It also receives 'memory consumption' statistics sent in the form of messages by the child processes and aggregates them to judge the total virtual memory consumed by the implementation. Monitor programs then log these details into a separate csv file which is later read by performance visualizer to generate interesting plots.

V. Performance Visualization

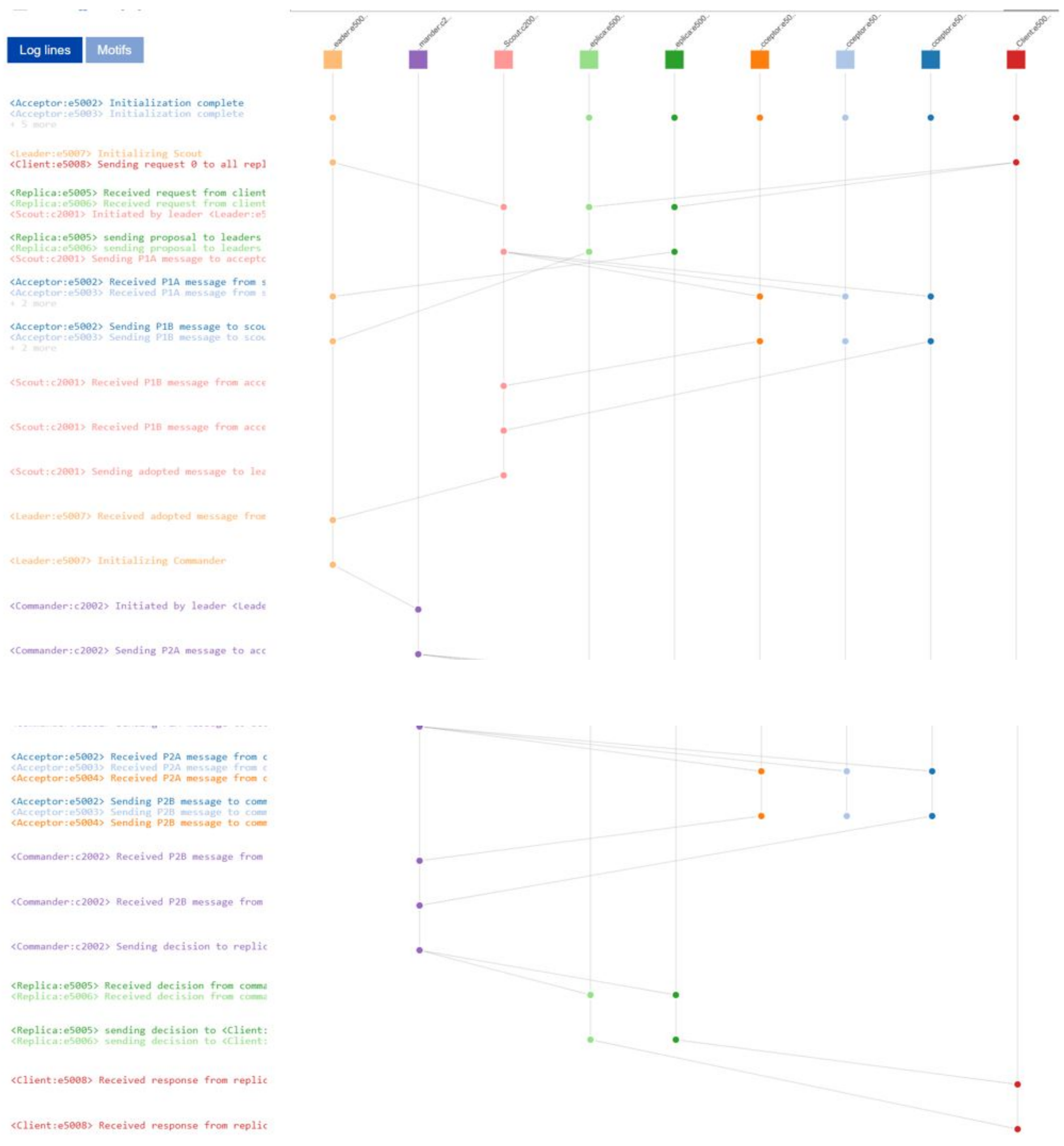
We are logging and visualizing various performance metrics as mentioned in subsection IV (Drivers & Monitors) and analyzing trends for both Paxos and Raft implementation for comparison. We also did the same comparisons after deliberately failing a leader for both implementations as shown below.

- **Without Leader Crash**
  - We ran the implementations for multiple parametric values ranging from 3 processes to 11 processes(acceptors in case of Paxos and servers in case of Raft)
  - Also, we ran the implementations for varying range of number of requests from 1 to 9.
  - Each of the varied parametric values were run 8 times each and the average parametric values were plotted.
  - We used Seaborn to make factorplots for comparing the implementations.
- **With Leader Crash**
  - We crashed the elected leader in case of Raft and any one leader in Paxos after a proposal is sent.
  - After making the above change, Seaborn factorplots were made similar to "without leader crash".

VI. Logging system & ShiViz

- We have created a central logging system with five distinct logging levels, viz. INFO, DEBUG, WARNING, ERROR & CRITICAL. We primarily have two kinds of logging systems in this project.
- In the first logging system, we extract performance reports from our driver programs (monitor.raft.da & monitor.paxos.da). and dump them in separate log files. We use these logs to plot performance curves for the different algorithms.
- In the next system, we directly manipulated the implementations to capture the inter-process communication between the constituent processes/ servers, accordingly with the timestamps on their individual vector-clocks. We stored the information incrementally in a buffer and had the information dumped in a log-file in a specific format that could be parsed using simple regex patterns, and could be used directly by the ShiViz tool to generate visualizations.

We are using the VRA Paxos implementation in DistAlgo as our first example. We have created a driver around it to run the code and log the results in a file. Further, we are attempting to log the movement of messages between the Acceptors, Learners and Proposers, and log them in specific formats, so they could be parsed by visualization tools like ShiViz. Further, We are using the python library 'seaborn' to create graphs and charts based on the logged performance data by our driver. We are using the 'psutil' library of Python to capture the performance details.



```
<Acceptor:e5002> Initialization complete
<Acceptor:e5003> Initialization complete
+ 5 more

<Leader:e5007> Initializing Scout
<Client:e5008> Sending request 0 to all repl

<Replica:e5005> Received request from client
<Replica:e5006> Received request from client
<Scout:c2001> Initiated by leader <Leader:e5

<Replica:e5005> sending proposal to leaders
<Replica:e5006> sending proposal to leaders
<Scout:c2001> Sending P1A message to accepto

<Acceptor:e5002> Received P1A message from s
<Acceptor:e5003> Received P1A message from s
+ 2 more

<Acceptor:e5002> Sending P1B message to scou
<Acceptor:e5003> Sending P1B message to scou
+ 2 more

<Scout:c2001> Received P1B message from acce

<Scout:c2001> Received P1B message from acce

<Scout:c2001> Sending adopted message to lea

<Leader:e5007> Received adopted message from

<Leader:e5007> Initializing Commander

<Commander:c2002> Initiated by leader <Leade

<Commander:c2002> Sending P2A message to acc

<Acceptor:e5002> Received P2A message from c
<Acceptor:e5003> Received P2A message from c
<Acceptor:e5004> Received P2A message from c

<Acceptor:e5002> Sending P2B message to comm
<Acceptor:e5003> Sending P2B message to comm
<Acceptor:e5004> Sending P2B message to comm

<Commander:c2002> Received P2B message from

<Commander:c2002> Received P2B message from

<Commander:c2002> Sending decision to replic

<Replica:e5005> Received decision from comma
<Replica:e5006> Received decision from comma

<Replica:e5005> sending decision to <Client:
<Replica:e5006> sending decision to <Client:

<Client:e5008> Received response from replic

<Client:e5008> Received response from replic
```

Figure 3: Sample ShiViz Log for Multi Paxos

Figure 3 shows a screenshot from ShiViz log we generated from our implementation of Multi-Paxos with 1 leaders, 2 replicas, 1 clients and 3 acceptors

IV. Testing & Evaluation

I. Steps to run

1. **For Performance Comparison**

- **driver_test_perf.da** Run driver_perf.da using "python -m da driver_test_perf.da". It will give us 2 options to choose implementations to run - DistAlgo Paxos and DistAlgo Raft.
- On entering 1, it asks for DistAlgo Paxos to choose from varying number of acceptors and number of requests per client. On entering 'a', it runs DistAlgo Paxos for varying range of acceptors. On entering 'b', it runs DistAlgo Paxos for varying range of requests per client.

- On entering 2, it asks for DistAlgo Raft to choose from varying number of servers and number of requests per client. On entering 'a', it runs DistAlgo Raft for varying range of servers. On entering 'b', it runs DistAlgo Raft for varying range of requests per client.
- Enter '1' for performance evaluations with one leader crash and '0' for evaluations without leader crash.
- The default values for paxos are 5 acceptors, 4 replicas, 2 leaders, 3 clients, 3 requests and 8 runs. Default values for raft are 5 servers, 3 clients, 3 requests, 3000 timeout and 8 runs. All runs by default are without leader.
- Two log files named raftlogfile.csv and paxoslogfile.csv will be generated.
- Run Perf.Vis.da and 3 pdf files will be generated with names output1.pdf, output2.pdf and output3.pdf will be generated containing the performance comparison visualizations.

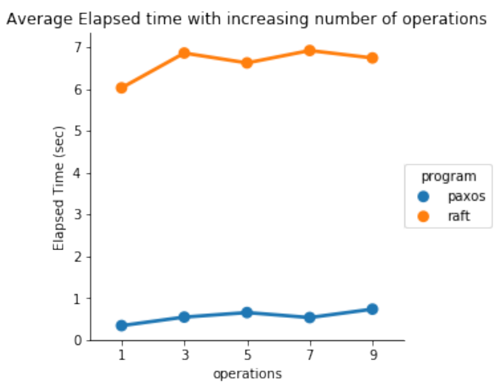
2. **For ShiViz Logs**

   - **driver.da** Run driver.da using "python -m da driver.da". It will give us 2 options to choose implementations to run - DistAlgo Paxos and DistAlgo Raft.
   - Currently, shiviz logs have been added to only DistAlgo Paxos. Shiviz logs for raft will be generated in future work as we are planning to automate the shiviz logging part without having to change the implementation.
   - Choose 1 for running DistAlgo Paxos for generating logs for Shiviz. Enter 3 2 1 1 1 1 for count of accpetors, replicas, leaders, clients, operations and runs respectively.
   - A file named vrpaxos-buffer.log file is generated.
   - Open "https://bestchai.bitbucket.io/shiviz/" and click on 'Try out on ShiViz', upload the log file "vrpaxos-buffer.log" and click on Visualize.

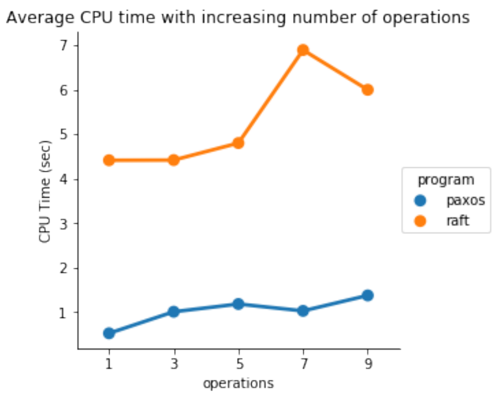
II. Results

1. Various number of Requests per client

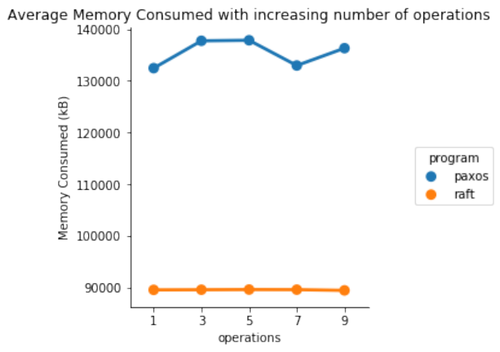
   - *Figure* 4*a* shows latency comparison of both Raft and Paxos implementations. It can be seen that latency for both Raft and Paxos increases on increasing number of requests per client. Another important observation is that elapsed time for Raft is much more in comparison to Paxos. This is attributed to the fact that leader election happens in the raft consensus implementation prior to the achieving consensus via log replication.
   - *Figure* 4*b* shows CPU time comparison of both Raft and Paxos implementations. We see that in general Paxos takes less CPU time than Raft. On increasing number of client requests, we observe that cpu time for paxos increases at a steeper rate than raft and will surpass it.
   - *Figure* 4*c* shows memory consumption comparison of both Raft and Paxos implementations. We have compared virtual memory consumption of both the algorithms. We show that Raft consumes constant amount of memory with the increase in the number of requests, whilst Paxos shows a general increasing trend.



(a) Latency



(b) CPU time

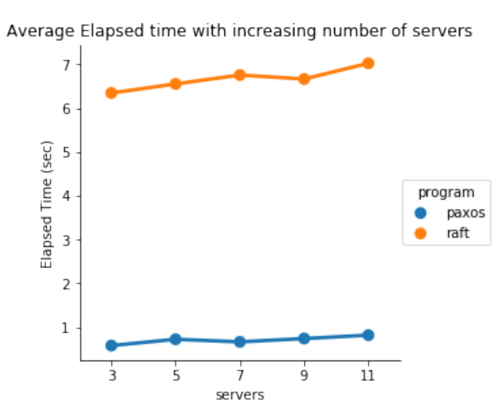


(c) Memory Consumption

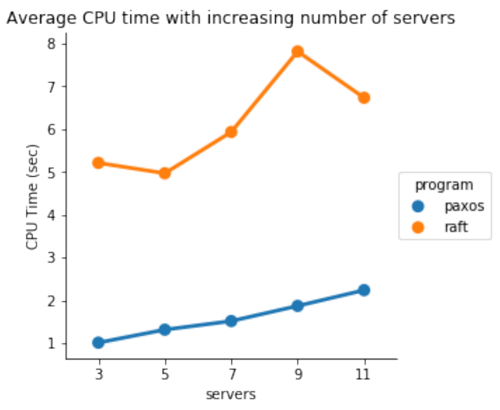
Figure 4: Comparison between Raft and Paxos varying number of requests

2. Various number of processes (servers):

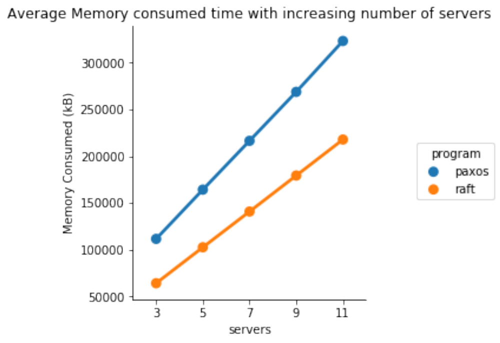
   - *Figure 5a* shows latency comparison of both Raft and Paxos implementations. With increasing the number of component processes, the rate of increase in the latency of Raft shows a slightly more increasing rate than that of Paxos. Also, Paxos has a much lower latency in general than Raft.
   - *Figure 5b* shows CPU time comparison of both Raft and Paxos implementations. The trend of CPU time consumption is almost same to the trend of CPU time consumption with increasing number of requests. Raft consumes much more time than Paxos.
   - *Figure 5c* shows memory consumption comparison of both Raft and Paxos implementations. Both the algorithms display steady increases, however, Paxos has a slightly higher value. The graphs 4c and 5c shows Paxos is much more time efficient, but much less memory efficient that Raft in general.



(a) Latency



(b) CPU time

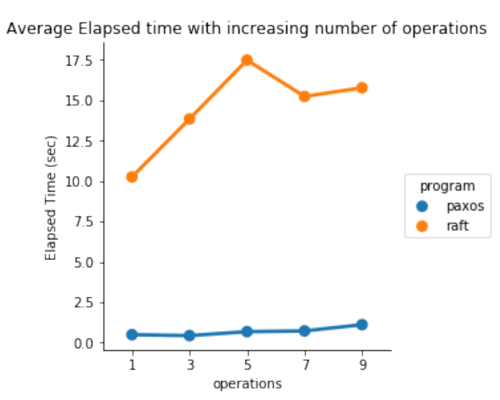


(c) Memory Consumption

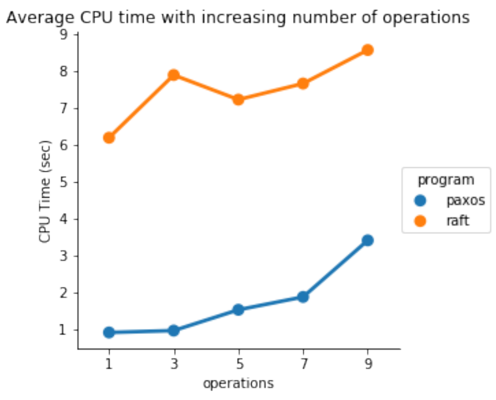
Figure 5: Comparison between Raft and Paxos varying number of processes

3. Varying number of requests per client on leader crash

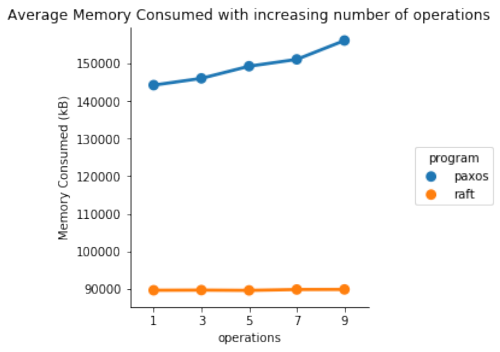
   - *Figure 6a* shows latency comparison of both Raft and Paxos implementations. Since, Raft has only one leader that is chosen after leader election, on leader crash, leader election happens again which increases the elapsed time. It can be seen that latency in Raft has increased significantly on increasing number requests per client on leader crash as compared to latency without leader crash.
   - *Figure 6b* shows CPU time comparison of both Raft and Paxos implementations. CPU time also increases for Raft for same reason as elapsed time as in 6a.
   - *Figure 6c* shows memory consumption comparison of both Raft and Paxos implementations. Memory consumption for Paxos is same as Figure 4c (without leader crash) and for Raft, it has increased but not very significantly.



(a) Latency



(b) CPU time



(c) Memory Consumption

Figure 6: Comparison between Raft and Paxos varying number of requests on leader crash

V. Future Work

- Currently, for showing ShiViz visualizations, we are dumping logs in the format required by ShiViz while sending and receiving messages in both vrpaxos and raft consensus implementations. We plan to automate this logging. To achieve this, we have to inherit the DistAlgo process class and modify send and receive functions to handle additional logging part.
- We are currently comparing performances of DistAlgo implementations of paxos and raft only as the DistAlgo code resembled the pseudo-code of the algorithm to the most extent. We plan to extend the framework to support consensus implementations in other languages such as python as well. This can be easily done in a minimal way.

References

[1] Urbán, P 2004, 'Comparing Distributed Consensus Algorithms', Proc. IASTED Int'l Conf. on Applied Simulation and Modelling (ASM), To Appear.

[2] Agrawal, S 2016, 'A Performance Comparison of Algorithms for Byzantine Agreement in Distributed Systems', 2016 12th European Dependable Computing Conference (EDCC), 249-260.

[3] Hayashibara, N 2002, 'Performance Comparison Between the Paxos and Chandra-Toueg Consensus Algorithms', 2002 Infoscience, hrefhttp://infoscience.epfl.ch/record/52482.

[4] Lin, B 2017, 'vrpaxos', GitHub Repository: 'https://github.com/DistAlgo/distalgo/tree/master/da/examples/vrpaxos'.

[5] Lin, B 2017, 'raft', Github Repository:
'https://github.com/DistAlgo/distalgo/tree/master/da/examples/raft'

[6] Ongaro, D 2014, ' In search of an understandable consensus algorithm', USENIX ATC'14 Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference, 305-320.

[7] Rutgers University Consensus Notes: 'https://www.cs.rutgers.edu/ pxk/417/notes/content /consensus.html'.