Dynamo Simulator

Karanbir Singh Chahal (ksc487)[1] and Udit Arora (ua388)[1]

[1]**New York University**

ABSTRACT

Reliably storing and accessing data in a distributed environment is important for modern web-scale applications. Dynamo showed how an eventually-consistent distributed key-value store can be built using different techniques while handling failures, which allows it to be used in production for demanding problems. In this project, we implement Dynamo from scratch and provide an easy to use web-app that lets users analyze different system and network configurations. We present an analysis of the performance of some of these configurations. Our code can be accessed at `https://github.com/karanchahal/dynamo-clone`.

INTRODUCTION

DeCandia et al. (2007) proposed Dynamo - an eventually consistent, highly available key-value store. Dynamo was implemented for use at Amazon by various internal services that required a high availability key-value store. They showed how different techniques can be combined to provide a single highly-available system and demonstrated that an eventually-consistent storage system can be used in production with demanding applications. They also provided insight into the tuning of these techniques to meet the requirements of production systems with very strict performance demands.

Bailis et al. (2014) introduced a way to quantify eventual consistency in a Dynamo-like weakly consistent key-value store. The presented a WARS system model of delays, and proposed the (Δ, p)-regular semantics to measure the degree of eventual-ness of the consistency. A system provides (Δ, p)-regular semantics if the probability of a read returning the value of the latest write is p.

Our work implements Dynamo in the Python language along with a simulation framework to carry out different kinds of analysis. We also develop a web-based front-end for the simulation. In this work, we analyze the (Δ, p)-regular semantics for different system configurations. Additionally, we analyze the performance of the GET and PUT requests under different system and load conditions. We also analyze the impact of failures on the system.

APPROACH

We use Python process as Dynamo nodes with GRPC-based communication between them. We also build a web-app based simulation framework using Python-Flask for experimentation with different network and system configurations.

We implement the following Dynamo protocols:

- Partitioning using consistent hashing
- Vector clocks for reconciliation of reads
- Handling temporary failures using hinted handoff and sloppy quorum
- Gossip protocol for failure detection

DESIGN AND IMPLEMENTATION

Dynamo

Our implementation of Dynamo primarily consists of:

- **DynamoNode**: This contains the logic corresponding to an individual node in Dynamo.

- **Launcher**: This contains the partitioning and spawning logic to spawn multiple Dynamo nodes based on configuration.
- **Client**: This contains methods for sending different type of requests to the nodes.
- **Web Simulator**: This is the web-based front-end for easy simulation and analysis.

We built each component in a modular manner, with unit-tests across modules to check the correctness. Figures 7 and 8 shows the simulator that lets users launch customized Dynamo instances and send different requests to it. Our implementation has the following features:

- Our implementation uses python multiprocessing to spawn n processes and each process communicates with each other using GRPCs. The objects transferred over the wire are mentioned in the dynamo.proto protobuf file.
- The class object that acts like a server for each node is defined in dynamo_node.py. This class object contains the following main public RPC functions:

    1. **Get()**: This function asks the server for the value for a particular key. The **Read()** function performs read to form a sloppy quorum.
    2. **Put()**: This function asks the server to add a key value pair to the Dynamo Hash Table instance. Note that the key will not necessarily be stored in the node for which the original Put() request is made, it may be rerouted according to the Dynamo protocol. A function **Replicate()** replicates a Put request.
    3. Other functions that allow for dynamic failing and gossiping between node are **Fail()** (to fail a certain node or bring it back up), **Transfer()** (for hinted hand-off recovery) and **Gossip()** to toggle the gossip protocol behavior.

- The dynamo instance can accessed by client functions mentioned in the file client_dynamo.py

Web simulator

Figures 7 and 8 show the web-based simulator application. The first setup page lets users launch a customized Dynamo environment with different parameters for the system and the network, as shown in Figure 7. Once the Dynamo environment is launched, the second client page (as shown in Figure 8) lets users send multiple GET and PUT requests and analyze the response time characteristics of these requests.

EXPERIMENTS AND RESULTS

Probabilistic Bounded Staleness Analysis

Figures 1 and 2 show the (Δ, p)-regular semantics under normal and uniform latency distributions respectively with the same mean of 50 ms. We observe the following:

- The consistency is higher for all configurations when we have uniform latency distributions as compared to normal distributions, despite having the same mean.
- W, which is the number of successful write replications before a PUT request completes, has the highest impact on consistency.
- When we increase R and W such that R + W $>$ N, we get a highly consistent data store.

Additional analysis

Distribution of Get/Put Request Response Times

In Figure 3 we plot out the distribution of the time that GET/PUT requests take under a varying number of failing nodes. The network has a latency of 10 ms, and we use 8 processes with two different configurations of N,R,W:

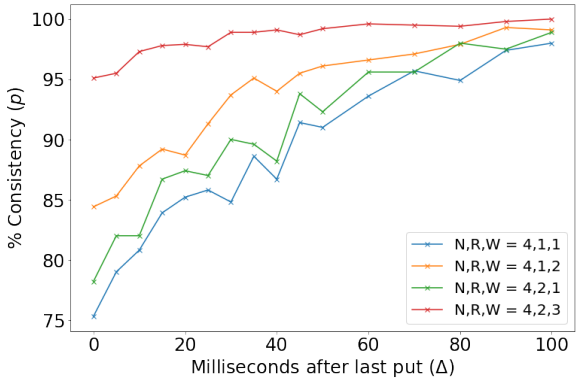


Figure 1. (Δ, p)-regular semantics under normal latency distribution with mean = variance = 50 ms. Each line represents a different system configuration.

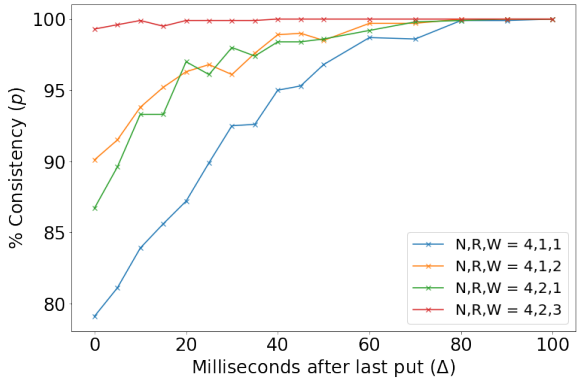


Figure 2. (Δ, p)-regular semantics under uniform latency distribution with a mean of 50 ms. Each line represents a different system configuration.

1. N,R,W = 4,1,3: This configuration favors faster read times because of a lower 'R' value.
2. N,R,W = 4,3,1: This configuration favors faster write times because of a lower 'W' value.

We send 1000 GET and PUT requests each in batches of 10 parallel requests. The requests are either sent to a random key from the key space, or a selected group of keys to simulate heavy load on those keys. We observe that Dyanmo can easily be configured to favor reads or writes, and that majority of the processing is I/O bound which leads to insignificant impact due to heavy load on some keys.

Configuration	Get (mean)	Get (99.9th %ile)	Put (mean)	Put (99.9th %ile)
N: 4, R: 1, W: 3	29.22 ms	36.68 ms	44.84 ms	66.28 ms
N: 4, R: 3, W: 1	42.78 ms	77.67 ms	30.13 ms	38.91 ms

Table 1. GET and PUT response times for different system configurations.

Table 1 shows the mean and 99.9th %ile response times for a fresh batch of 1000 GET/PUT requests. We can see that the two configurations behave as expected.

Distribution of Get/Put Request Response Times under Random Failure

In Figure 5 we plot out the distribution of the time that GET/PUT requests take under a varying number of failing nodes. The network has a latency of 10 ms and we use 8 processes with R=1 and W=3 and where the total number of replicas for the data is equal to 4. Hence, we observe a longer tail in the distribution of the PUT requests when the number of failures increase. The gossip protocol is running as a background task which help nodes maintain a consistent view of the health of the system. Predictably we observe that with lots of nodes failing, some requests take longer than others as they need to find suitable replica nodes to perform hinted hand-off.

Mean and 99 percentile durations under different rates of failures

In Figure 4 we test out the mean and 99th %ile durations of the PUT requests of a Dynamo system with W=3, N=4, R=1 at varying rates of failures. The network has a latency of 10 ms. After every x milliseconds, we fail a random node and bring back some other failed node at a gossip refresh rate of 0.5-0.8 seconds. We run this experiment to validate that the duration of the PUT requests should increase with a higher failure rate and we indeed verify that this is the case. The fact that the mean duration performance isn't affected much by failures reinforces the strengths of the availability guarantees of Dynamo.

Degree of consistency across replicas

In Figure 6 we test the proportion of 'R' read requests (that are performed during a GET) which have the latest value among them. This is done under a normal latency distribution with mean and variance of 10 ms. This analysis allows us to check the degree of consistency of replicas. From the figure, we observe that increasing 'W' leads to a much higher ratio of read requests having the latest value.

CONCLUSIONS

We implemented Dynamo from scratch and presented a web app to simulate and analyze different loads under different configurations. We analyzed the system under varying loads, failures, and looked at some internal consistency metrics as well.

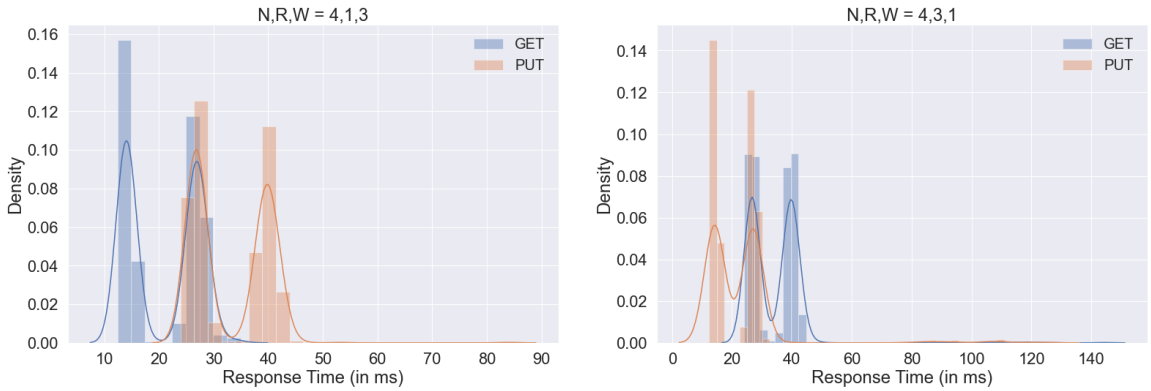
Future Work

- We would like to understand the correlation between the gossip refresh rate and the rate at which requests come into the system. A method that dynamically adjusts the gossip refresh rate based on the rate at which failures are increasing can significantly help build a fresher view of the global health of the system. This is helpful as during the presence of failures and a slow gossip protocol, the nodes can build a global view of a system that thinks that most of the nodes are failing hence leading it to fail requests. An adaptive gossip protocol would ease the communication burden on the network and at the same time be responsive to transient failures.
- A method for failure analysis that can analyse logs of a distributed system and predict potential failures based on the distribution of requests coming in would be immensely useful. During the lifetime of a distributed system a lot of exceptions can happen and some of them are very rare to replicate. A machine learning algorithm which can derive semantic meaning from a distributed trace would be super interesting. Perhaps a network taking a trace as input and spitting out the exact memory of each node at any state. This would be possible if the logs are structured well and serve as a useful pre-training objective to the model. Methodologies that further train this model to classify certain errors and enable zero shot generalization to new errors would be useful work in our opinion.
- To improve load-balancing, we can use multiple techniques like auto-scaling and routing of requests to other top-N nodes in the ring other than the first. We are also interested in building out a Kubernetes-like orchestrator that adds new nodes to the system to gain further insights into distributed systems.
- Improve simulation front-end to support more types of analysis.

REFERENCES

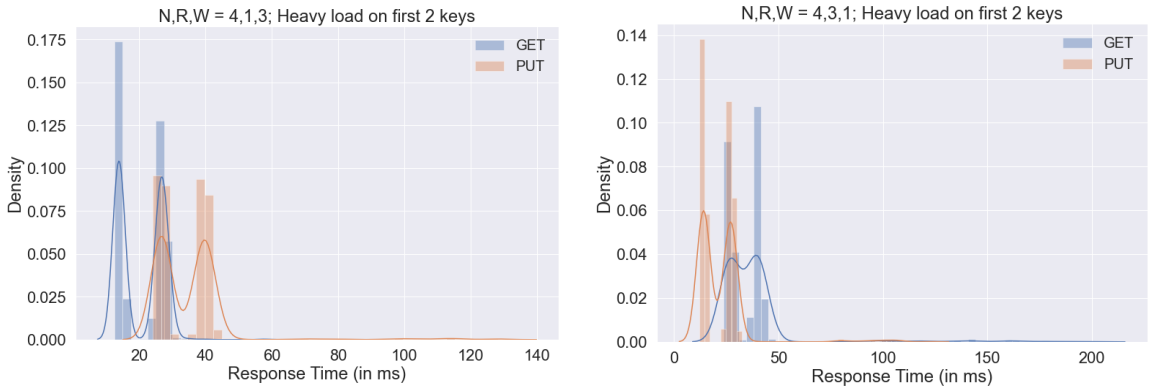
Bailis, P., Venkataraman, S., Franklin, M. J., Hellerstein, J. M., and Stoica, I. (2014). Quantifying eventual consistency with pbs. *Commun. ACM*, 57(8):93–102.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220.



(a) Read-favoring system configuration

(b) Write-favoring system configuration



(c) Read-favoring system configuration with heavy load on some keys

(d) Write-favoring system configuration with heavy load on some keys

Figure 3. Distribution of response times for different system configurations and uniform load across keys vs skewed load on certain keys.

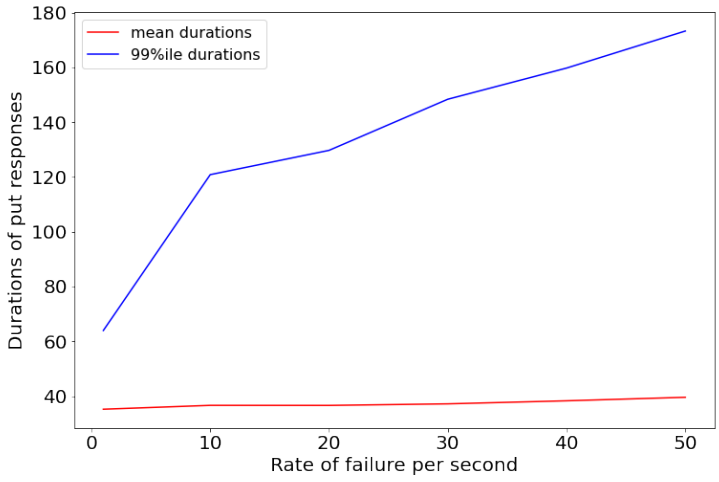


Figure 4. The mean and 99 percentile durations of put requests during random failure of nodes at various rates. We observe significant changes in the 99 percentile distribution whereas the mean response rate roughly stays the same. This experiment proves the robustness of hinted hand-off in Dynamo.

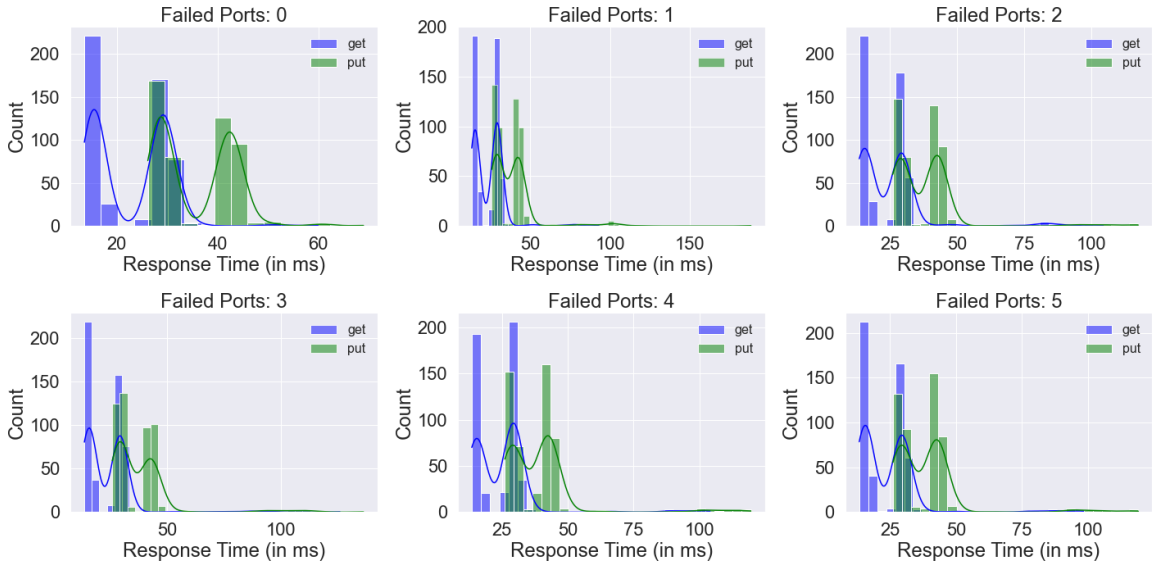


Figure 5. The distribution of get and put requests in a varying number of failing nodes. This configuration has 8 nodes out of which we progressively fail 5. We take care to allow the gossip protocol to catch up until we decide to test a new failing configuration. This experiment validates that under failures the distribution grows a longer tail as it is expected that a node contact various other nodes to fulfill it's request. We tested this under a standard gossip refresh time of anywhere between 0.5 and 0.8 seconds.

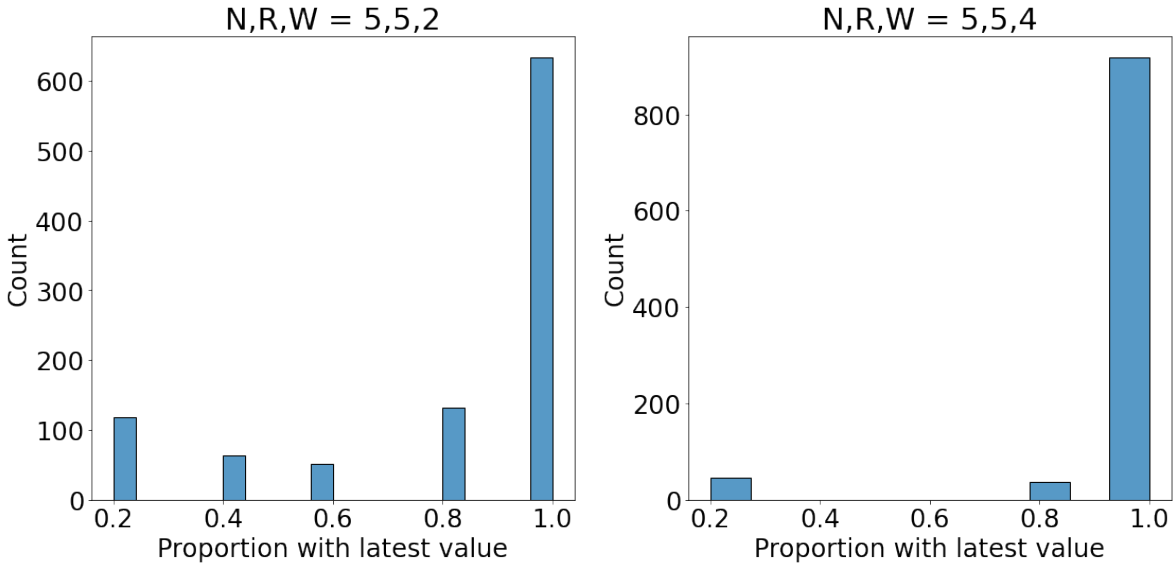


Figure 6. The proportion of 'R' read requests that have the latest value to measure the degree of consistency across replicas. We observe that increasing 'W' improves the proportion because Dynamo waits for a higher number of write responses before returning the PUT response.

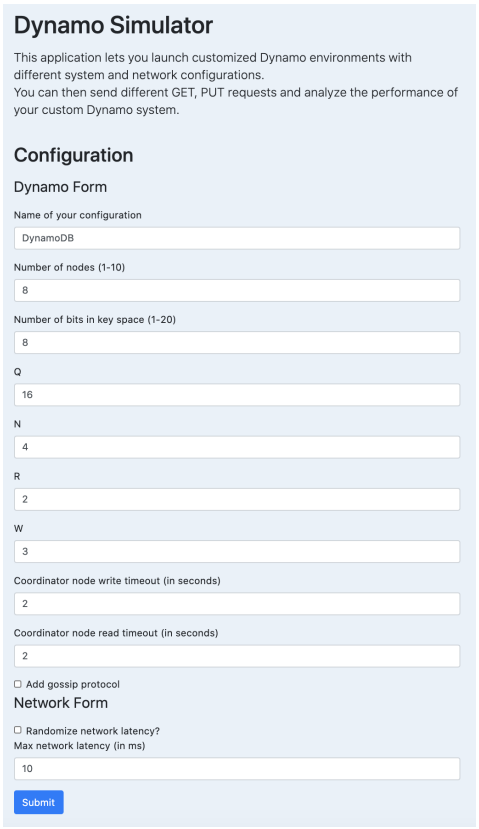


Figure 7. The setup page of the Dyanmo Web Simulator that lets users spawn Dynamo instances with different system and network configurations.

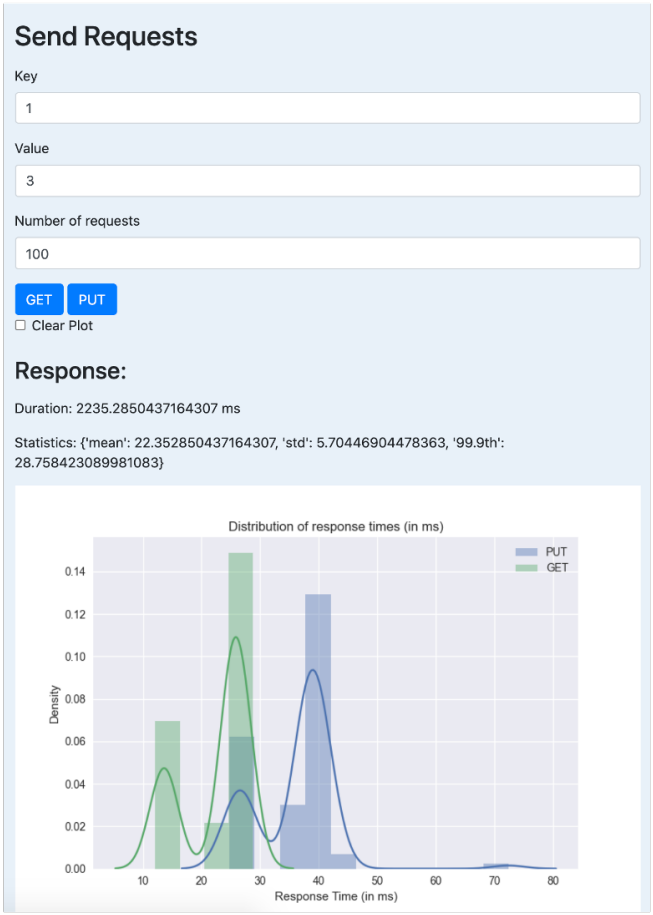


Figure 8. The client page that lets users send requests and analyze response times.