Adaptive Request Modeling in Clusters during Adverse States

Ankit Gupta
ankit.gupta.iiith@gmail.com

24 September 2020

Abstract

A distributed system consists of multiple nodes that share incoming load among themselves. In a centrally coordinated environment, the responsibility falls on a gateway to manage request dispatch to this cluster of nodes. In this paper, we propose an adaptive and unified approach to request dispatch to enhance decision making capabilities of such a gateway in case of partial and total cluster shutdown states. Using the described algorithms as a base, we derive a formula for landing probability on a node and explore its implication on various cluster states. We also show, via graphs, the request dispatch pattern when using this approach with adverse cluster states.

Keywords - Load balancing, distributed systems, request dispatch, weighted random, probability, queueing

1 Introduction

The architecture of the internet has changed dramatically in the past few years. The software that we build, deploy and consume has evolved - from XML-based RPC services to Web APIs to Messaging Queues. As data and consumption grew, software started taking advantage of distributed methodology [1] [2] to manage itself. This resulted in scale, performance and fault tolerance [3]. Services are now organized into cluster and generally utilize an added layer of gateway that perform critical functions such as load balancing, service discovery, authorization, monitoring among many others.

An important objective of such a gateway is to make sure the client remains unperturbed from cluster state changes and is able to perform tasks as reliably as possible. Thus, the request dispatch pattern (from gateway to the cluster) plays an important role in the final fate of client requests. There are two types of load balancing algorithms used in gateways for request dispatch - static and dynamic [4]. Static algorithms work on a pre-programmed knowledge of the system, and make dispatching decisions using that limited information. This allows static algorithms to be fast and less resource intensive. Dynamic algorithms, on the other hand, tend to achieve an overall objective of the system such as improving response times, minimizing latency, minimizing CPU/memory utilization, and so on. This is achieved by localizing the cumulative state of key parameters of the system and performing relevant computations before tending to a request.

Implementations of these algorithms have limited mechanisms to handle adverse scenarios such as service connectivity issue, CPU/memory spikes, timeouts, corrupt disk/memory and deployment breaks and even less mechanisms to reliably recover from them. The performance parameters like CPU usage, memory usage, IOPS, disk fragmentation, http response error codes provide hints as to when a particular node is not working reliably, but a single algorithm focuses on optimizing for a single parameter only and ignores the rest. An increase or decrease in values for any of these parameters can contribute towards degraded cluster performance resulting in failing most client requests. Considering any such anomaly in these parameters as an *error*, we describe an adaptive approach which can deliver optimal request dispatch during adverse cluster states. We consider below two adverse cluster states for the rest of the paper -

- **Partial Cluster Shutdown State (PCS) -** This state is defined when one or more nodes in the cluster is exhibiting error and unable to respond satisfactorily as shown in figure (1). It assumes atleast one node in the cluster is healthy. We look at a probabilistic dispatching algorithm to move requests away from errored nodes towards other healthy nodes leading to increase in success probability in the first request trial.

- **Total Cluster Shutdown State (TCS) -** This state is defined when all nodes in the cluster are exhibiting error and unable to respond satisfactorily as shown in figure (2). We look at a queueing based approach with regular retries so client is freed up of re-sending requests.

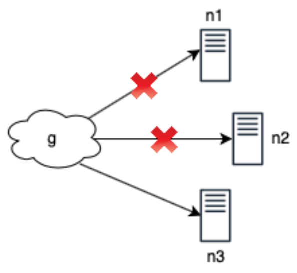


Figure 1: PCS

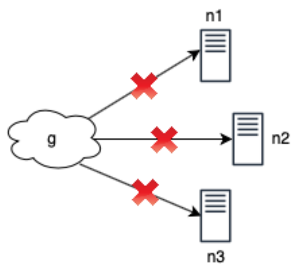


Figure 2: TCS

2 Background and Motivation

Load balancing algorithms in distributed systems have been studied extensively in the past. The goal of load balancing is largely two fold - first to improve performance and second to improve reliability [5]. From the perspective of a client request, performance refers to how quickly can a healthy node be selected to process while reliability refers whether or not some node is selected to process. In an ideal scenario (when all nodes in the cluster are up and running), all client requests are evenly balanced across all nodes and the desired performance and reliability metrics remain intact for the lifetime of the system. In a less than ideal scenario, there is always a trade off between performance and reliability. Optimizing one for a system may impair the other. Hence, we talk in terms of probability of load balancing success to determine by how much a distributed system is able to achieve its goals.

As mentioned earlier, dynamic load balancing algorithms rely on certain key parameters of the system based on which they make dispatching decisions. Even static algorithms can be converted to dynamic algorithms by bringing dynamic weight assignment into consideration, where weight of a node is derived from a key parameter of the system. In majority of situations, fine-grained knowledge of the problem itself is good enough to reveal these key parameters. However, there are a few formal ways to help determine them. [6] described methods to compute costs of key parameters in distributed system to achieve dynamic load balancing. Another method described in [7] uses Probabilistic Model (ProMo) for data flows which can be used to test the key parameters of a cluster that determine good load balancing and scheduling performance in the network. Once the key parameters are handy, appropriate discussions can happen on the load balancing strategies. Load balancing algorithms are designed to usually focus on a single parameter to optimize request dispatch. For example, *leastconn* selects node with least number of active connections, *roundrobin* selects nodes in order, *random* selects a random node, *ip-hash* uses client IP to generate hash key to map to nodes, *url-hash* is similar to *ip-hash* but uses request URL instead of IP for hashing, JSQ-d [8] relies on shortest queue parameter, JIQ [8] [9] relies on idle queue parameter, and some rely on average response time, cpu utilization, connectivity etc. Weighted minimal connection [14], weighted random [] and weighted round robin [14] [15] have also been explored and proven to be more efficient than non-weighted versions. For a large enough system, power-of-d-choices heuristic can be used, wherein only d/N randomly chosen nodes are probed for key parameter and request dispatched to the optimal node. A special case, power-of-two-choices [10] is utilized in a few modern load balancing systems (for example *Nginx* [11]). While these algorithms work fairly well in isolation, the need of a practical system dictates that *all* key parameter of the system must be considered for optimal request dispatch. Modern load balancing systems like Nginx [12] and Traefik [13] only support combination of one or two of these parameters and provide limited fallback options in case of erroneous dispatches. Any deviation from normal state of these parameters does contribute to some *error* in the system and we must consequently adapt our load balancing strategy to those modified states.

Herein lies the goal of this paper - to develop an adaptive load balancing strategy by utilizing error feedback from nodes irrespective of number of key parameters of the system. We explore different cluster states (PCS and TCS) in detail, and look at two approaches applicable to each state. In case of PCS, we look at *AdaptiveProbabilityDispatch* - a new dynamic weight assignment approach for weighted random algorithm and explore fallback options in case of erroneous dispatches. In case of TCS, we look at *DeferredQueue* - a queueing based approach with retrial capabilities. We will present how both approaches can work together to maximize the probability of load balancing success without compromising on performance and reliability metrics.

3 Intuition

A cluster can consist of multiple nodes (n) which stand by for processing client requests. The cluster employs a gateway that can dispatch a client request to one of the nodes from the cluster. In this paper, we will assume an open queueing network where nodes are computationally identical and requests are independent of each other and follow Poisson process of arrival.

3.1 Partial Cluster Shutdown State

Let's assume r_i is the count of requests landing on node i out of total R requests landing on the gateway. Let's also assume that the relative importance of node i is given by weight factor w_i. The relation between r_i, w_i and R can be modeled as shown in equation (1).

$$r_i = \frac{w_i}{n} R \tag{1}$$

The steady state of a cluster is defined when the total failed requests (fr) are zero across nodes. In steady state, all requests R reaching the gateway in time t are dispatched to any one of the nodes and get processed successfully in single attempt. Thus, $w_i = 1$ for all nodes in steady state. From equation (1) it can be deduced that if w_i is decreased, r_i must also decrease. Landing probability LP_i is closely associated with this definition. Lower weight will imply lower landing probability and thus lower number of requests. The need to dispatch lower number of requests arises when the previous requests sent to a particular node are failing. If at any point of time, fr_i is the number of failed requests on node i, then the error $e_i$on the node can be written as -

$$e_i = fr_i \tag{2}$$

The problem is that of decreasing total error on the cluster, while making sure that the request processes successfully on one of the nodes -

$$e^{R+\Delta R} < e^R,\ (r_i)_{success} > 0 \tag{3}$$

where e^R is the total error after request r is dispatched -

$$e^R = \sum_{i=1}^{n} (e_i^R)^2 \tag{4}$$

If we continue dispatching requests to errored node i, e_i will increase and equation (3) will not be satisfied. What we want is to adjust LP_i such that e_i reduces and gateway diverts less traffic than before to this errored node and more traffic to other healthy nodes. In other words, LP_i of a request on node i should reduce over time. Consequently, if longer time elapses, the rate of requests to node i should be negligible compared to other healthy nodes. Relation between LP_i and e_i can be written as -

$$LP_i = f\left(\frac{1}{e_i}\right) \tag{5}$$

Considering that a fraction of requests will still land on node i, and are bound to fail, they must be retried on the next node using *roundrobin* algorithm until one of the nodes responds success or all nodes are tried once, whichever occurs first. In section (5.1), we will look at an algorithm to reduce landing probability when error on a node increases.

3.2 Total Cluster Shutdown State

If all nodes start exhibiting the same behaviour as node i, eventually none of the nodes would be able to serve the request and we must queue the request in a FCFS queue, waiting to be executed later. Once one or more nodes are up, the queued requests must be re-dispatched to the cluster, and the error reset to zero on them. In section (5.2), we will look at an algorithm to implement a FCFS queue with queue and dequeue functions.

4 Approach

4.1 Adaptive Probability Dispatch

As described in section (4.1), landing probability is closely related to weights. We calculate weights of each node in cluster whenever any request dispatch is required. If total error is zero, all nodes are assigned equal weights and a node is chosen randomly for dispatch. If there is error found on any node, we go on to find weight on that node as Max Effective Error E_m in cluster divided by node error e_i. Once weights are assigned, the problem reduces to calculating weighted probabilities and the resulting node is chosen for dispatch.

All requests are run through *AdaptiveProbabilityDispatch* with incremental trial count until a trial succeeds or all trials are exhausted. Number of trials is equal to number of nodes.

Algorithm 1 AdaptiveProbabilityDispatch

```
Input: $request, trial, prev\_node$
Output: $result, node$
Ensure: $n \leftarrow GetNodeCount()$
  if $trial = 1$ then
    $each\, e_i \leftarrow GetErrorCount(i)$ {Get Error on each node}
    $each\, E_i \leftarrow \lfloor (1 + e_i)^{3/2} \rfloor$ {Compute Effective Error on each node}
    $E_m \leftarrow \max_i(E_i)$ {Compute Max Effective Error from each node}
    if $E_m = 1$ then
      $s = Randomize(1, n)$ {Use randomized algorithm to find target node}
      return $Send(request, s)$
    else
      $each\, w_i = \lceil E_m/(1 + e_i) \rceil$ {Compute weights for each node}
      $each\, pre_i = pre_{i-1} + w_i$ {Compute prefix array}
      $r = rand(1, pre_n)$
      $s = RangeBinarySearch(r, pre)$ {Use range binary search to find target node in pre such that r is in
      range between pre[i-1] and pre[i]}
      return $Send(request, s)$
    end if
  else if $trial = 2 \rightarrow n$ then
    return $Send(request, prev\_node + 1)$ {$prev\_node$ = node selected in previous trial)}
  end if
```

The *Send* function above tries a request on a node s and returns SUCCESS in case of successful processing and FAIL in case of failed processing. The routine *HandleRequest* is responsible for request trials and subsequent actions.

Algorithm 2 HandleRequest

```
Input: $request$
Ensure: $n \leftarrow GetNodeCount()$
  $prev\_node = -1$
  for $trial$ ←1 to $n$ do
    $(result_{trial}, i) = AdaptiveProbabilityDispatch(request, trial, prev\_node)$
    if $result_{trial} = FAIL$ then
      $IncrementErrorCount(i)$
      $prev\_node = i$
    else if $result_{trial} = SUCCESS$ then
      $ResetErrorCount(i)$
      $break$
    end if
  end for
  if $all\, result_{trial} = FAIL$ then
    $PushToDeferredQueue(request)$
  end if
```

4.2 Deferred Queue

The routine *HandleRequest* pushes a request to deferred queue if all trials of *AdaptiveProbabilityDispatch* return FAIL. This deferred queue *DQ* is a FCFS queue which holds all failed requests waiting for any node in the cluster to be available. The routine *RetryFromDeferredQueue* must dispatch the requests periodically after re-applying *AdaptiveProbabilityDispatch* described in section (5.1). The routine can be considered as a new client sending requests and follows the same treatment as real clients.

Algorithm 3 PushToDeferredQueue

```
Input: $request$
  $Queue(DQ, request)$ {Queue request to deferred queue DQ}
```

Algorithm 4 RetryFromDeferredQueue

```
$request \leftarrow Dequeue(DQ)$
$HandleRequest(request)$
```

Deferred queueing is recommended for DML-style operations (*Create*, *Update*, *Delete*) which can affect the state of the system. For example - a database batch update/delete query, insertion to message queue, logging are few candidates for such a scenario where deferred execution is desired. For *Read* operations, where immediate response is a requirement, queueing will not help. The client must be informed about the queueing behaviour so it can proceed with subsequent requests. The order of execution is also to be maintained at all times in the queue.

5 Analysis

5.1 Probability of a request landing on node i

From the approach in Algorithm 1, it can be deduced that the probability of a request landing on node i when trial=1 is given by -

$$LP_i = \frac{\left\lceil \dfrac{E_m}{(1+e_i)} \right\rceil}{\sum_{s=1}^{n} \left\lceil \dfrac{E_m}{(1+e_s)} \right\rceil} \tag{6}$$

where E_m is the Max Effective Error. As E_m and denominator (Total Effective Error) is constant for each LP_i at any given cluster state, we deduce that -

$$LP_i \propto \frac{1}{e_i}$$

The role of Effective Error E_i on node i here is to make sure there is a substantial difference between the weights of nearby nodes so even the errored nodes keep receiving requests, albeit diminished, but for a substantial period of time.

For the sake of simplicity, we are going to do an approximations to equation (6). We will replace the term $\left\lceil \dfrac{E_m}{1+e_i} \right\rceil$ with $\dfrac{E_m}{1+e_i}$, as the maximum deviation between the two can be 1 and the approximation error in equation (6) is negligible for large e_i values. Thus, the probability of any request to land on node i until it is errored can be given as -

$$LP_i = \frac{\dfrac{1}{(1+e_i)}}{\sum_{s=1}^{n} \dfrac{1}{(1+e_s)}} \tag{7}$$

This can be further broken into -

$$LP_i = \frac{\dfrac{1}{(1+e_i)}}{\left(\sum_{s=1}^{n-1} \dfrac{1}{(1+e_s)}\right) + \left(\dfrac{1}{1+e_i}\right)} \tag{8}$$

As the summation term in above equation consists of remaining healthy nodes, we can treat it as constant h, hence for $e_i = 0$, we get-

$$LP_i^1 = \frac{1}{1h+1}$$

which is the probability that any request or 1st request after node i got errored will land. Once errors are detected, e_i increases linearly, while E_i and E_m increase exponentially. For r$^{\text{th}}$ request that land on node i, we can derive approximate probability as -

$$LP_i^r = \frac{1}{rh+1} \tag{9}$$

We can derive the formula for h as well -

$$h = \sum_{s=1}^{n-1} \frac{1}{1+e_s} = (n-1) \quad for\, e_s = 0 \tag{10}$$

From above analysis, it can be seen that if n_e is the count of errored nodes, the generic formula for h will be $(n - n_e)$ and generic formula for LP_i will then become -

$$LP_i = \frac{\dfrac{1}{(1+e_i)}}{(n-n_e) + \left(\sum_{s=1}^{n_e} \dfrac{1}{(1+e_s)}\right)} \tag{11}$$

If we plot equation (11) for a 3-node cluster with 1 node down, we get the plot shown in figure (3). The probability starts out as 1/3 but drastically reduces in case of error. The probability reduces even more drastically for a 8-node cluster as shown in figure (4). As evident, the probability plots for errored nodes follow a harmonic progression with $a = 1$ and $d = 1$.

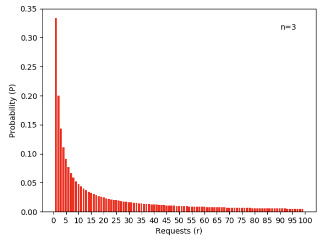


Figure 3: P vs r for 3-node

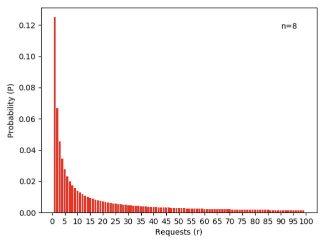


Figure 4: P vs r for 8-node

5.2 How quickly can requests be moved away from an errored node i

The landing probability of exactly r requests on node i can be given as -

$$CLP_i^r = \prod_{j=1}^{r} LP_i^j \tag{12}$$

Hence, if the rate of requests to a cluster is $\frac{R}{t}$, then rate of request on the node i will be -

$$\frac{r}{t} = CLP_i^r \times \frac{R}{t} \tag{13}$$

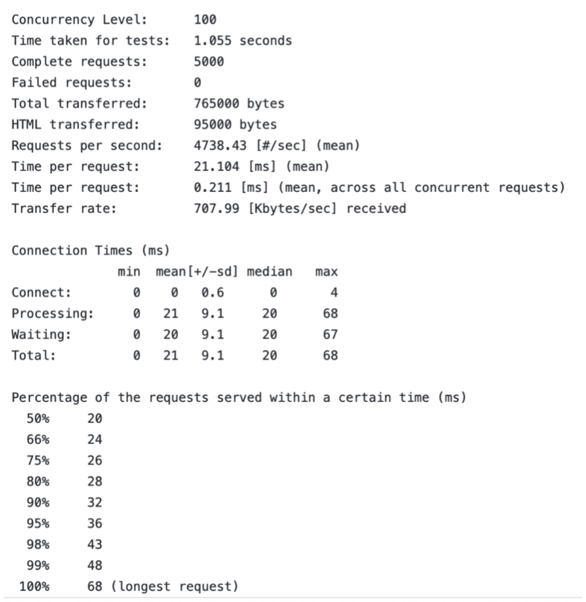
This is an important result as it shows that rate of requests to node i decreases proportional to $1/r$ every time a request lands on node errored node i. Consequently, the rate of request to healthy nodes increases which can also be derived from equation (11). These results will be much more clear when we look at request pattern graphs for a few cluster states in section (7).

6 Results

We set up a program - *serviceq* [19] for demonstrating our approach in section (5). *serviceq* is a gateway with configurable service paths, queue length, retry gaps, and timeouts. It is written in *Go 1.13* [20] for handling http (HTTP/1.1) requests with builds available for linux and darwin environments.

For testing purposes, a simple http web server was created with support for GET and POST calls. This web server was also written in Golang and three instances were deployed on three separate machines on the same port. *serviceq* was configured with *endpoints* of these instances. Other configurations added to *serviceq* were *retry_gap* = 0 and *queue_length* = 2048. The tests were performed using Apache Bench (ab) for different rates of request and concurrency levels. For figure (5) and (6), we have considered 5k GET requests at a concurrency level of 100.

```
$ ab -c 100 -n 5000 -m  GET http://serviceq.net:5252/
```

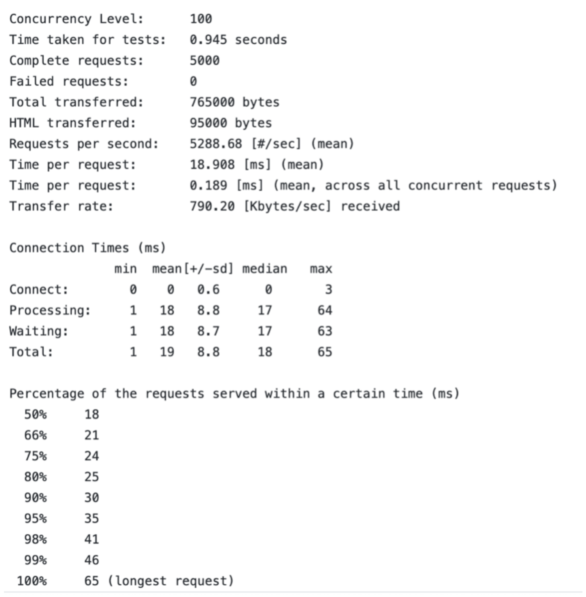


```
Concurrency Level:      100
Time taken for tests:   1.055 seconds
Complete requests:      5000
Failed requests:        0
Total transferred:      765000 bytes
HTML transferred:       95000 bytes
Requests per second:    4738.43 [#/sec] (mean)
Time per request:       21.104 [ms] (mean)
Time per request:       0.211 [ms] (mean, across all concurrent requests)
Transfer rate:          707.99 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.6      0       4
Processing:     0   21   9.1     20      68
Waiting:        0   20   9.1     20      67
Total:          0   21   9.1     20      68

Percentage of the requests served within a certain time (ms)
  50%     20
  66%     24
  75%     26
  80%     28
  90%     32
  95%     36
  98%     43
  99%     48
 100%     68 (longest request)
```

Figure 5: Steady State, n nodes up



```
Concurrency Level:      100
Time taken for tests:   0.945 seconds
Complete requests:      5000
Failed requests:        0
Total transferred:      765000 bytes
HTML transferred:       95000 bytes
Requests per second:    5288.68 [#/sec] (mean)
Time per request:       18.908 [ms] (mean)
Time per request:       0.189 [ms] (mean, across all concurrent requests)
Transfer rate:          790.20 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.6      0       3
Processing:     1   18   8.8     17      64
Waiting:        1   18   8.7     17      63
Total:          1   19   8.8     18      65

Percentage of the requests served within a certain time (ms)
  50%     18
  66%     21
  75%     24
  80%     25
  90%     30
  95%     35
  98%     41
  99%     46
 100%     65 (longest request)
```

Figure 6: Errored State, n2 node down

Figure (5), with n nodes up, shows the average processing rate of 4738 req/s and 99% requests completing within 48ms. The load distribution determined is n_1: 1654, n_2: 1653, n_3: 1693. Due to the error count being zero on all nodes, program decided to distribute the load equally between the nodes (well, pseudo-equally)

Figure (6), with n_2 down, shows the average processing rate of 5288 req/s and 99% requests still completing within 46ms. The load distribution determined by the this time is n_1: 2529, n_2: 65, n_3: 2473. Due to the fact that program has now learnt from the service errors, it has reduced the probability of selecting errored node. Only 65 requests (1.3%) were dispatched to errored node n_2. The percentage reduces on further requests.

Another thing to note is that the trend between 50% and 99% times in both cases remain similar, which goes to show that the connectivity errors don't harm the throughput and response times. The results follow similar trends for POST requests and for total cluster shutdown state post recovery. Next, we will plot request pattern graphs for few common scenarios and look at how requests were dispatched to various nodes.

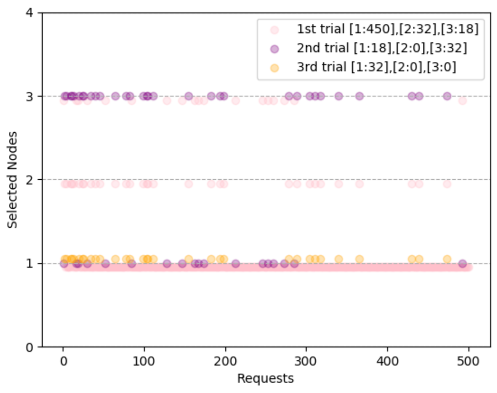


Figure 7: 1 up, 2 down

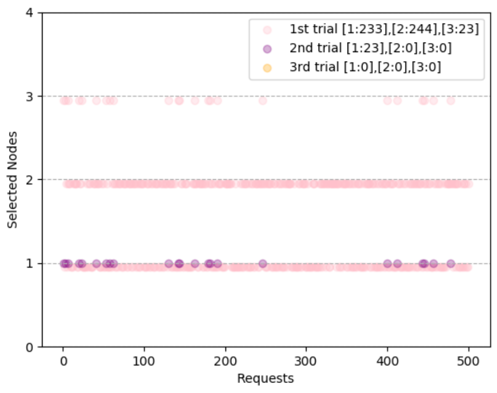


Figure 8: 2 up, 1 down

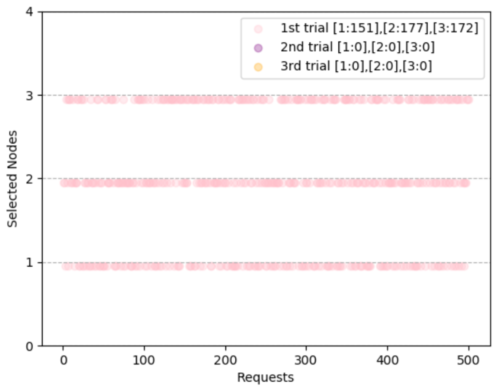


Figure 9: All up

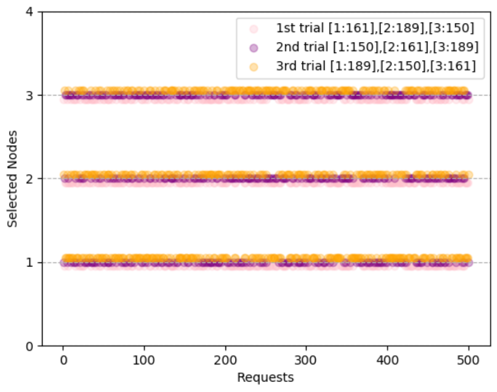


Figure 10: All down

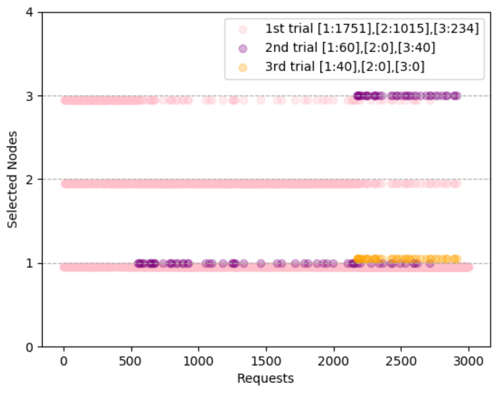


Figure 11: 1 down after 1k, 2 down after 2k

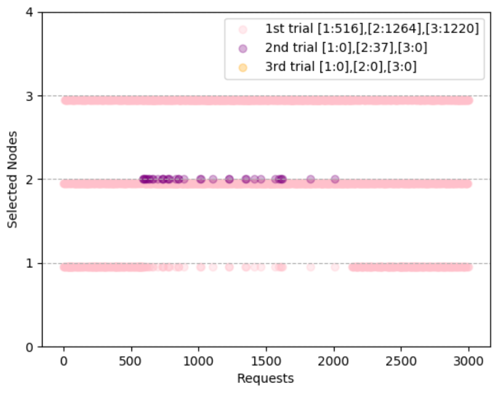


Figure 12: 1 down after 1k, all up after 2k

Figure (7) shows that initially the landing probability was same for all nodes. But when n_2 and n_3 were found errored, most requests were diverted away as depicted by less and less pink dots during the course of requests. Also, n_1 which received some of the requests from trial 2 and 3, eventually became the primary selection for trial 1 itself with landing probability $\tilde{1}$.

Figure (8) shows that initially the landing probability was same for all nodes. But when n_3 was found errored, most requests were diverted away as depicted by less and less pink dots during the course of requests. Also, n_1 and n_2 eventually became primary selections for trial 1 itself with equal landing probability.

Figure (9) shows that the landing probability was equal throughout for all nodes and all requests succeeded on trial 1 itself.

Figure (10) shows that the landing probability was equal throughout for all nodes (due to equal error) and failed on trials 1,2,3.

Figure (11) shows that initially the landing probability was same for all nodes upto 1k requests. But when n_3 was found errored, most requests were diverted away as depicted by less and less pink dots during the course of requests. n_1 became selection for trial 2 which gradually reduced as well when n_1 and n_2 became primary selection for trial 1 with equal landing probability. Furthermore, when n_2 was errored after 2k requests, n_3 started receiving few requests due to *roundrobin*, which failed as well, and n_1 was finally selected in trial 3 and succeeded.

Figure (12) shows that initially the landing probability was same for all nodes upto 1k requests. But when n_1 was found errored, most requests were diverted away as depicted by less and less pink dots during the course of requests. n_2 became selection for trial 2 which gradually reduced as well when n_2 and n_3 became primary selection for trial 1 with equal landing probability. Furthermore, when n_1 was up again after 2k requests, all nodes eventually got to equal landing probability.

7 Conclusion

In this paper, we proposed an approach for assigning weights (using error feedback) for optimal request dispatch to the cluster. Landing probabilities are calculated for every node and a probabilistic dispatching algorithm is described. If request processing fails altogether, fallback option is to queue the request and retry at a later point of time. This approach tends to increase the success probability of a request in minimum trials as shown in the paper. The approach can also be used in conjunction with existing load balancing algorithms to reduce overall cluster errors.

8 Challenges and Follow-Up Work

The challenge lies in finding key parameters of the cluster along with safe threshold values and their feedback mechanism to the gateway. For example, the *serviceq* implementation of *HandleRequest* takes into account http response codes for incrementing error count. As not all performance indicators can fit into the same feedback mechanism, we will be working on building a unified mechanism to identify and share key parameters with the gateway and extend *serviceq* to handle it. We will also be exploring development of a client side process (similar to *serviceq*) in the form of layer 7 network protocol.

References

[1] Jim Waldo, Geoff Wyant, Ann Wollrath, Sam Kendall. A Note on Distributed Computing. In *IEEE Micro, 1994*

[2] L. M. Vaquero, L. Rodero-Merino, J. Caceres and M. Lindner. A break in the clouds: towards a cloud definition. In *SIGCOMM ACM Computer Communication Review,vol. 39, pp. 50–55, December, 2008*

[3] B. Clifford Neuman. Scale in Distributed Systems. In *Readings in Distributed Computing Systems. IEEE Computer Society Press, 1994*

[4] Shaoyi Song, Tingjie Lv, and Xia Chen. Load Balancing for Future Internet: An Approach Based on Game Theory. In *Journal of Applied Mathematics, Volume 2014*

[5] Reliability Engineering. https://w.wiki/dWG. Visited on *2020/09/24*

[6] Stavros I. Souravlas, Angelo Sifaleras. Network Load Balancing Using Modular Arithmetic Computations. In *Vlamos P. (eds) GeNeDis 2016. Advances in Experimental Medicine and Biology, vol 988. Springer, Cham*

[7] Stavros Souravlas ProMo: A Probabilistic Model for Dynamic Load-Balanced Scheduling of Data Flows in Cloud Systems. In *Electronics 2019, 8(9), 990*

[8] Mark van der Boor, Sem C. Borst, Johan S.H. van Leeuwaarden, Debankur Mukherjee. Scalable load balancing in networked systems: A survey of recent advances. In *arXiv, arXiv:1806.05444, 2018*

[9] Lu, Yi and Xie, Qiaomin and Kliot, Gabriel and Geller, Alan and Larus, James and Greenberg, Albert. Join-Idle-Queue: A Novel Load Balancing Algorithm for Dynamically Scalable Web Services. In *Perform. Eval. 68. 1056-1071. 10.1016/j.peva.2011.07.015., 2011*

[10] M. D. Mitzenmacher. The power of two choices in randomized load balancing. As *PhD thesis, University of California at Berkeley, Department of Computer Science, Berkeley, CA, 1996*

[11] Nginx and Power of Two Choices. https://www.nginx.com/blog/nginx-power-of-two-choices-load-balancing-algorithm. Visited on *2020/09/24*

[12] Nginx Load Balancing. https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/#method. Visited on *2020/09/24*

[13] Traefik Load Balancing. https://doc.traefik.io/traefik/v1.7/basics/#load-balancing. Visited on *2020/09/24*

[14] Pan, Z. , Jiangxing Z. Load Balancing Algorithm for Web Server Based on Weighted Minimal Connections. In *Journal of Web Systems and Applications (2017) 1: 1-8*

[15] W. Wang and G. Casale. Evaluating Weighted Round Robin Load Balancing for Cloud Web Services. In *16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Timisoara, 2014, pp. 393-400, doi: 10.1109/SYNASC.2014.59*

[16] R.M. Karp. An introduction to randomized algorithms. In *Discrete Appl. Math., 34 (1-3) (1991), pp. 165-201*

[17] Efraimidis, Pavlos. Weighted Random Sampling over Data Streams. In *10.1007978-3-319-24024-4_12, 2010*

[18] Chiang, M., Cheng, H., Liu, H. et al. SDN-based server clusters with dynamic load balancing and performance improvement. In *Cluster Comput, 2020*

[19] ServiceQ. https://github.com/gptankit/serviceq. Visited on *2020/09/24*

[20] Golang. https://golang.org/doc/go1.13. Visited on *2020/09/24*