SOCIETY FOR SCIENCE AND EDUCATION
UNITED KINGDOM

TNC TRANSACTIONS ON NETWORKS AND COMMUNICATIONS

VOLUME 8, NO. 6
ISSN: 2054 -7420

A Comparison of Fair Sharing Algorithms for Regulating Search as a Service API

Evorell L. Fridge, Sikha Bagui
Department of Computer Science, University of West Florida, Pensacola, FL
efridge@uwf.edu; bagui@uwf.edu

ABSTRACT

Providers of a Search as a Service (SaaS) environment must ensure that their users will not monopolize the service or use more than their fair share of resources. Fair sharing algorithms have long been used in computer networking to balance access to a router or switch, and some of these algorithms have also been applied to the control of queries submitted to search engine APIs. If a search query's execution cost can be reliably estimated, fair sharing algorithms can be applied to the input of a SaaS API to ensure everyone has equitable access to the search engine.

The novelty of this paper lies in presenting a Single-Server Max-Min Fair Deficit Round Robin algorithm, a modified version of the Multi-Server Max-Min Fair Deficit Round Robin algorithm. The Single-Server Max-Min Fair Deficit Round Robin algorithm is compared to three other fair sharing algorithms, token-bucket, Deficit Round Robin (DRR), and Peng and Plale's [1] Modified Deficit Round Robin (MDRR) in terms of three different usage scenarios, balanced usage, unbalanced usage as well as an idle client usage, to determine which is the most suitable fair sharing algorithm for use in regulating traffic to a SaaS API. This research demonstrated that the Single-Server Max-Min Fair DRR algorithm provided the highest throughput of traffic to the search engine while also maintaining a fair balance of resources among clients by re-allocating unused throughput to clients with saturated queues so a max-min allocation was achieved.

Keywords: Fair Sharing, Deficit Round-Robin, Max-Min Fairness, Token-bucket.

1 Introduction

Full-text search engines like Solr or Elasticsearch are flexible resources that can be used in a variety of applications within an organization. No longer just for performing search services in web applications, these tools are growing in popularity among infrastructure teams who use them to store server logs and monitor critical network traffic. Due to licensing or maintenance restrictions, an organization may ask users from different teams to share the same search engine instance to accomplish all of these tasks. Similarly, Search as a Service (SaaS) providers may host the content for several users on the same search engine instance. Regardless of the application, users of a shared search engine resource may suffer

DOI: 10.14738/tnc.86.9633
Publication Date: 24th January 2021
URL: http://dx.doi.org/10.14738/tnc.86.9633

degraded performance if their neighbors are using more than their fair share of resources. It is important for everyone to share their resources fairly.

Fair sharing is a concept that has been heavily researched in the area of computer networking. Similar to a search engine instance with a limited number of users and queries it can support, a network link has a well-defined maximum amount of throughput that can be used within a given period of time. When multiple users are sharing a network connection, it is important to ensure that they are given equal access to that resource. Various algorithms have been developed to fairly regulate the flow of traffic from multiple users over a shared network link, and these algorithms have also been applied to other load balancing applications such as software APIs [2].

An API can be used as a proxy in front of a full-text search engine in order to verify user access credentials and control user request rates. Although APIs can be configured to regulate traffic to ensure an equal number of requests per second, this is not guaranteed to ensure a fair distribution of resources when dealing with full-text search queries. A user may submit the same number of requests as other users, but their queries may still disproportionately impact the system if they take more time to execute. This can be addressed by scoring each query according to the estimated impact it will have on the search engine. These scores can be used by a fair sharing algorithm like deficit round-robin [3] to ensure that the load on a search engine is balanced. Several variations of deficit round-robin and other related algorithms exist which may be suitable for this sort of query-aware load balancing: token-bucket, Deficit Round Robin (DRR), Peng and Pale's [1] Modified Deficit Round Robin (MDRR), and Khamse-Ashari et al.'s [4] Multi-Server Max-Min Fair DRR.

This paper modifies Khamse-Ashari et al.'s [4] multi-server algorithm for use in a single server environment, hence it is referred to as Singler-Server Max-Min Fair DRR in this paper. This paper also determines which of the four algorithms, token-bucket, Deficit Round Robin (DRR), Peng and Pale's [1] Modified Deficit Round Robin (MDRR), and Single-Server Max-Min Fair DRR, is most effective for use in a SaaS API to balance requests from multiple users in a fair way. The performance of each of these variations on DRR are compared in terms of a balanced usage scenario, unbalanced usage scenario as well as an idle client usage scenario.

The rest of this paper is organized as follows: section two provides more background on fair queueing and how the fairness of an implementation can be measured. Section three reviews previous work on fair queuing that has been done in computer networking and how this has been applied to fair sharing of databases and search engines. Section four discusses the specifics of each of our fair sharing algorithm implementations. Section five describes the experimental setup and methods used to evaluate the performance of each fair sharing algorithm. Section six presents the results and section seven discusses the results of the experiments. Finally, section eight presents the conclusions that can be derived from this work and approaches for future work.

2 Background

The field of computer networking has long had to deal with the problem of multiple users contending for limited resources. There are two similar concepts in networking that are related to fairness: rate-limiting and fair queuing. The algorithms that implement each of these concepts have much in common, but the impact on the customers using the resource can be very different. Rate limiting effectively enforces a bandwidth cap on a customer, keeping their overall usage of the resource under a fixed limit [5]. A rate

Copyright © Society for Science and Education, United Kingdom

limiting algorithm may allow periodic bursts of data that exceed a specific limit, but the overall effect of this algorithm over time is to keep bandwidth usage below a specific level.

Fair queuing "provides a straightforward strategy for dividing bandwidth among multiple senders according to preset percentages" [5]. Most fair queueing algorithms maintain queues in memory for each user or class of users to measure and regulate the data that they are attempting to send. Some transmissions may have to wait in a queue until the user is allowed to transmit again. Queues have a limited amount of storage, so if the data being sent to a queue exceeds its capacity it must do something with the overflow data. A drop-tail approach simply discards any packets as they arrive if the queue does not have room. A switch or router may choose to take a more proactive approach to managing an overflow situation by using active queue management algorithms to notify the sender that an overflow is about to happen.

Unlike rate limiting, fair queuing algorithms are not intended to keep users from using more than a specific amount of bandwidth. They are built to evenly distribute the available bandwidth among all the active users attempting to use a data link. If only one user is using the data link and the rest of the users are inactive, a fair sharing algorithm should allow that user to receive the whole allocation of bandwidth. If a fair sharing algorithm provides max-min fairness, then it "first fulfils the demand of users who need less than they are entitled to, and then evenly distributes unused resources among the 'big' users" [6].

The concept of fairness can be expressed in many different ways, but Jain's fairness index [7] is often used as a way to quantify the level of fairness experienced in a situation over a period of time. This index has a scale that ranges from 0.0 to 1.0, where 1.0 is perfectly fair and all users receive the same allocation. A value can be calculated for a given set of allocations by using the formula,

$$f(x) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \cdot \sum_{i=1}^{n} x_i^2} \quad (1)$$

where x_i is the allocation for a given user and n represents the number of users. Jain's work gives us the example of distributing 20 dollars among 100 people. If every person were given 20 cents the scheme would have a fairness index of 1.0 or 100% fair, but giving 10 people 2 dollars and no money to 90 people would result in a scheme with a fairness value of 0.10 or only 10% fair.

3 Related Works

3.1 Deficit Round-Robin (DRR)

In the scenario of a data switch with multiple high-speed inputs and one slower output, a service is considered fair if every stream of data has equal representation in the outgoing stream. Nagle [8] was one of the first to write about the need for fairness in networking and proposed a round-robin approach that took turns sending data. Although Nagle's ideas were on the right track, other researchers later realized a flaw in his approach: not all packets were of equal size. A client with consistently large packets could send more data over time although his packet count remained the same as that of his peers. One proposed solution to this problem was bit-level round-robin [9], which was guaranteed to be fair but required sorting an outbound queue and therefore had $O(\log n)$ time complexity for n queues. Although this algorithm was completely fair to all users, it did not perform well under a heavy load so its practical applications were limited.

McKenney [10] observed that the requirement to maintain a queue for each data stream was another shortcoming of these round-robin algorithms. When many users attempted to use a data switch the number of queues that needed to be maintained was unmanageable. McKenney introduced the concept of stochastic fair queuing, which used a hash-based algorithm to balance inputs from many sources into a manageable number of queues. His algorithm had $O(1)$constant time and was very fast but was not any fairer than Nagle's approach because it didn't address the problem of packets with variable sizes.

3.2 Deficit Round-Robin (DRR)

The deficit round-robin (DRR) approach [3] built on these previous ideas to deliver an algorithm that was very fast and was approximately very fair. Their DRR algorithm used the hash function from stochastic fair queuing to group traffic from many sources into queues. It then visited the active queues in a round-robin fashion, allocating a "quantum" of credits each time a queue was visited. During a queue's visit, it was allowed to transmit all packets it had enough credits to process. Additional packets that were unable to be sent were retained in the queue, and the remaining balance of that queue's credits were retained in a variable and could be used during that queue's next turn. This algorithm benefited from the constant time efficiency of the stochastic fair queueing model while also accounting for variable packet sizes by allocating each queue a budget of total data that could be sent rather than a specific number of packets. It ensured data rates that were approximately fair within the size of one quantum, which meant that the most out of balance it could be among all active clients was one "round" of traffic sent as a queue is serviced.

Several researchers have attempted to improve on the DRR algorithm since it was first published. Many of the improvements have been made to allow time-sensitive data (such as video conferencing or phone calls) to be transferred on the network without encountering the normal delays associated with waiting in a queue. Cisco networking equipment has implemented a modified form of DRR that introduces a priority queue solely reserved for packets that contain time-sensitive traffic [11]. Tong and Zhao [12] developed a variation of the DRR algorithm that allowed the arrival of higher-priority packets to introduce a reduction of the quantum values assigned to lower priority queues so that the high-priority traffic can be serviced faster.

Another study that introduced a variable quantum value to the DRR algorithm was Khamse-Ashari et al.'s [4] effort to apply the DRR algorithm to regulate traffic in a heterogeneous, multi-server environment. At the beginning of each DRR round, their algorithm estimated the amount of service that could be provided by each server during that period. A quantum value for that server was then generated and divided among the active queues so that a max-min allocation was achieved. The DRR algorithm was then executed normally on each server with the computed quantum value. Their Multi-Server Max-Min Fair DRR algorithm maintained the fairness level achieved by the original DRR algorithm while also realizing improved throughput on each server.

3.3 Token-Bucket

Although it is not technically a fair queuing algorithm, another credit-based algorithm known as token-bucket [13] can be used to limit traffic throughput on packet switches in situations where a bandwidth cap for a user is desirable. In this approach, customers each start with the same number of tokens. A round-robin approach is not used here, although queues are used for each input stream. Packets are scored based on byte size as they come in, and if they have a sufficient token balance they are allowed to

Copyright © Society for Science and Education, United Kingdom

transmit and the cost is deducted from their token balance. Once a customer exhausts their tokens, their packets must wait in a buffer until more tokens are allocated that can be used to pay for these queries. According to Dordal [5], "the main practical difference between fair queuing and token-bucket is that if one sender is idle, fair queuing distributes that sender's bandwidth among the other senders. Token-bucket does not: the bandwidth a sender is allocated is a bandwidth cap."

Token-bucket systems are straightforward to implement in networking-related applications because it's easy to measure the impact of each packet. A user's data allocation is deducted by the number of bytes of data each packet contains. Measuring the impact of a search term on the query performance in a full-text search engine can be more difficult. Macdonald, Tonellotto, & Ounis [14] noted that the query term count and "posting" size of an inverted index is often a predictor of query time. They used a linear regression learning model to predict query performance times. They also suggested using out-of-order processing strategies to submit potentially fast queries before sending slower ones.

3.4 Fair Sharing in Full-text Search

Contention for resources is also a very real problem in full-text search. In a full-text search engine, those resources are primarily disk I/O and the CPU time to run a query. Data networks can use bytes sent as a metric to reliably measure the bandwidth used by a client. However, search queries may have a variety of different execution times based on the contents of the index and the complexity of the query. Peng and Plale [1] demonstrated that these variations in query execution time make queries per second (QPS) a poor estimate of user performance. Instead, they proposed using CPU time as a true measurement of a query's impact on a system. This time can be measured as the execution time of a query when no other load is on the system.

Peng and Plale [1] trained a linear regression model to predict execution time for a given query's term count and posting size. They used a large sample of real-world search terms and ran them against a large data set of searchable text. The result was a model that they could use to quickly compute a cost score for a user's attempted query. Fridge and Bagui [15] demonstrated that a similar model could be constructed to estimate the execution time of an Elasticsearch query. They used the term vectors feature of the Elasticsearch API to obtain a document frequency value which was similar to the "postings" value used by other researchers. When combined with the number of terms in a query to train a linear regression model, this was shown to be a reliable predictor of query execution time.

Peng and Plale [1] used a modified version of the deficit round-robin algorithm to enforce fair sharing on Solr full-text search clusters. Their modification was that instead of taking turns, all users were to be given a bucket of assigned tokens that were refilled only once all tokens had been spent. This is similar to Heinanen and Guerin's [13] token-bucket approach, though token-bucket would refill user tokens according to a predetermined rate instead of waiting for all users to exhaust tokens. Peng and Plale [1] said that in their algorithm "no request queue for each tenant needs to be maintained and polled in a round-robin fashion, which is especially useful for [a] servlet based service." This algorithm also made it easier for Peng and Plale [1] to use a round robin approach on a distributed Solr search engine, since there was no need for common queues for the nodes to coordinate and maintain.

3.5 Dealing with Asymmetric User Load

Peng and Plale [1] also mentioned that a periodic refill of tokens may be necessary in their algorithm. This is because their modifications to deficit round-robin introduce the possibility that a heavy user would run out of tokens and experience a service outage while waiting for a comparatively light user to also run out of tokens. Their tests were run with consistently high usage volumes for all users which is not consistent with real-world traffic which often comes in waves of high and low usage and could vary depending on the popularity of a customer's application.

Heinanen and Guerin's [13] token-bucket algorithm mitigates this problem by continually filling up a user's token balance, but because it is a rate-limiting algorithm rather than a fair queueing one, the result would not be max-min fair. Shreedhar and Varghese [3] discuss the possible scenario in their deficit round-robin algorithm where input queues are not always saturated. If a flow is very "bursty and [has] nothing arrive for a large period," then it would of course not require any service during that period. They also state that "if the queue for a flow is periodically empty and the flow misses round-robin opportunities, these are opportunities that the flow could not use anyway." However, the original DRR algorithm [3] does not take advantage of these missed opportunities by distributing an inactive queue's quantum value among the other active queues. Therefore, the original DRR algorithm is only max-min fair if all queues are saturated.

With respect to Shreedhar and Varghese's [3] DRR algorithm, Dordal's [5] assertion that "if one sender is idle, fair queuing distributes that sender's bandwidth among the other senders" is not accurate. However, Khamse-Ashari et al.'s [4] modification of the DRR algorithm demonstrated that such behavior was possible. Applying a max-min calculation to every round of DRR and redistributing the surplus quantum among the active clients would ensure max-min fairness at the expense of slightly increased computational complexity.

3.6 Addressing the Research Gaps

This work addresses the following gaps in the research. First, it demonstrates the shortcomings of various fair sharing algorithms when used by customers with dissimilar or varying usage levels. It builds on Peng and Plale's [1] work by quantifying the execution costs of an Elasticsearch query and using that value to regulate query activity. Finally, it demonstrates how a simplified version of Khamse-Ashari et al.'s [4] Max-min Fair DRR algorithm can be used to enforce fair sharing on a SaaS API. Our Single-Server variant of Khamse-Ashari et al.'s [4] algorithm meets the needs of an API designed to enforce fair sharing among its clients and also maximizes the throughput between the API and Elasticsearch.

4 Fair Sharing Implementations

A REST API was constructed to regulate traffic to an Elasticsearch cluster using a variety of fair-sharing algorithms in multiple usage scenarios. The four algorithms, token-bucket, original DRR, Peng and Plale's modified DRR, and Single-Server Max-Min Fair DRR, were evaluated using the test API system on the Elasticsearch cluster. Each algorithm was implemented to manage four separate user queues, and all inbound requests to the API included an API key that identified which client originated the request. The requests contained search terms that were processed on Elasticsearch using an identical search query. All implementations shared several commonalities, including a credit / token balance for each queue. All implementations used a timer that would iterate periodically and perform some sort of maintenance on all the queues. Each algorithm used the same refill rate of ten credits every 20 ms, or 500 credits per second.

Copyright © Society for Science and Education, United Kingdom

4.1 The Token-Bucket Algorithm

The token-bucket algorithm [13] implementation was the simplest to implement. On each iteration of the timer, the algorithm would increment the token balance of each queue's bucket up to a maximum of 100,000 tokens. Inbound requests would first be scored by the linear regression algorithm and then the estimated value would be compared to the token balance. If a sufficient balance existed the request would be processed immediately. Otherwise, the request would remain in the queue and be re-evaluated during the next iteration of the timer.

4.2 Peng and Plale's Modified Deficit Round Robin Algorithm

The implementation of Peng and Plale's [1] modified deficit round robin algorithm was very similar to the token-bucket algorithm but with one important difference. On each timer iteration, this algorithm counted the number of queues that were either empty or had insufficient tokens to process their next query. It would only refill the token balance for all clients once all queues had reached that state. This algorithm seemed to perform better when a larger number of tokens was added at a slower rate, therefore a refill rate of 500 credits every second was used. It is the numerical equivalent of the ten credits per 20 ms rate used with the other algorithms. As with the token-bucket algorithm, requests were processed immediately if possible and queued for later evaluation if insufficient tokens existed for that user.

4.3 Deficit Round Robin

Shreedhar and Varghese's [3] deficit round robin (DRR) algorithm was implemented using the same timer interval as the previous algorithms. Each inbound request was added to a queue for the corresponding user, and requests were only processed during a round-robin iteration. During each 20 ms iteration of the timer, the algorithm would visit every queue in turn, zeroing out the credit balances of any inactive queues and allocating a quantum of ten credits to the active queues. During each queue's turn it would be allowed to process requests until it no longer had a sufficient balance to process the next item in the queue. The remaining balance (or deficit) would be retained in the queue for the next iteration.

4.4 Single-Server Max-Min Fair Deficit Round Robin

The Single-Server Max-Min DRR, a modification of the Multi-Server Max-Min Fair DRR [4], was modified for use by a single server as depicted in Figure 1. In this simpler implementation, the quantity of service provided by this single server was assumed to be a constant equal to our refill rate of 10 credits per 20 ms times the number of clients. Each inbound request was added to a queue for the corresponding user, and requests were only processed during a round-robin iteration. During each iteration of the timer, the algorithm would first count the number of queues that were idle and zero out the credit balances of each inactive queue. The algorithm would then visit every queue again in turn, allocating a quantum of ten credits to the active queues. Each active queue would also be given its share of credits that were not being used by any inactive queues, rounded down to the nearest integer. For example, if all four queues were active during this iteration then each queue would receive its normal 10 credit allocation. If one queue was idle, then its share would be divided equally among the other three queues and each one would get 13 credits. If only two queues were active and the other two were idle, then each active queue would receive 20 credits.

```
// Variables and constants
const integer N         // Total number queues
const integer Q         // Per queue quantum (credits), same for all queues
integer DC[1..N]        // Per queue deficit counter (credit balance)
integer NIA             // Number of inactive queues
queue queue[1..N]       // The queues

while (TRUE) do
   // Count inactive queues
   for i in 1..N do
      if queue[i].empty() then
         NA := NA + 1
      end if
   end for

   // Service active queues
   for i in 1..N do
      if not queue[i].empty() then
         DC[i] := DC[i} + Q    // Normal allocation
         if NIA > 0 do         // Additional fair share of unused credits
            DC[i] := DC[i} + floor( (NIA * Q) / (N - NIA) )
         end if

         // Process queue traffic
         while( not queue[i].empty() and
                  DC[i] ≥ queue[i].head().size() ) do
            DC[i] := DC[i] − queue[i].head().size()
            send( queue[i].head() )
            queue[i].dequeue()
         end while
         if queue[i].empty() then
            DC[i] := 0
```

end if

end if

end for

end while

Figure 1. Single-server max-min fair DRR algorithm.

5 Experimental Setup and Methods

The API for this experiment was developed in JavaScript using Node.js and Express to serve as a proxy to Elasticsearch for customer requests in the experiments. Node.js uses a non-blocking, event-driven server model, which made it possible to construct the API so that multiple requests could wait within queues without issue. The API connected to Elasticsearch using the JavaScript drivers and queued user requests according to a specified strategy. Implementations of the token-bucket [13], deficit round-robin [3], modified deficit round-robin [1], and the Single-Server Max-Min Fair DRR algorithm were all available for regulating search traffic within the API. In addition to user requests, the API was also used to load data and train linear regression models. Figure 2 is a graphic illustrating the API along with its clients, queues, and the fair sharing algorithm.

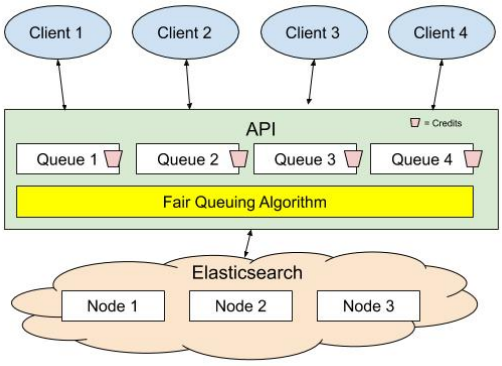


Figure 2. The research API design.

A three-node Elasticsearch cluster was used to perform these experiments. Each node was hosted on an Intel-based server with a 4 GHz quad-core, hyperthreaded processor, and 32 GB RAM. A 1 TB NVMe SSD storage device was dedicated to storing index data, and each node had a separate SSD as a boot drive. Elasticsearch 7.8.1 was installed on each node, and 16 GB of RAM were allocated to Elasticsearch's JVM heap while the rest of the memory was left for OS file caching. Two additional computers were used to run the API and simulate user requests, and all machines were connected via a one gigabit Ethernet switch.

Each customer was given an identical index consisting of public domain documents from the Hathi Trust digital library [16]. There were 539,777 documents in each index totaling 248 GB of data per user, and each document contained the digitized text of a book. Each Elasticsearch index was allocated 12 shards to store these documents, which resulted in 20GB of data stored in each shard.

User interaction was simulated with the Apache JMeter testing suite [17]. Server logs containing queries from AOL's search engine gathered from March 1 to May 31, 2006 [18] were used to simulate realistic user input. A set of 10,000 queries were randomly selected from the AOL data set, and the same data file was used by all JMeter processes.

Separate tests were conducted with the API running each of the fair sharing algorithm implementations. For each algorithm, an identical list of 10,000 randomly selected query terms were submitted to the API at different rates by using multiple concurrent threads in Apache JMeter. Concurrent user activity was simulated for each of the four hosted indexes for three different scenarios: a balanced usage scenario, an unbalanced usage scenario, and an idle client usage scenario.

The balanced usage scenario used 25 threads per user to simulate a situation where the load on each customer's index was as close to each other as possible. Each JMeter thread constantly submitted queries to the API as quickly as possible. Because the balanced usage scenario involves the same traffic to all four queues at the same rate, it should be inherently fair. The unbalanced usage scenario used thread settings of 1, 10, 15, and 25 in JMeter to apply a user load at different rates. This unbalanced scenario simulated a hosting situation where one client's search service is much more popular than another. This is when an API must use a fair sharing algorithm to enforce fairness.

The idle client usage scenario used thread settings of 0, 0, 15 and 25 to simulate a load on only half the client queues but completely idle usage on the other two. In this scenario, not only must an algorithm regulate the usage of the active queues fairly; it must also ensure that traffic to the search engine continues to be processed at the highest possible rate. This scenario highlights the advantages that a max-min compliant algorithm has over one that is fair but could be unnecessarily keeping requests in a queue while the capacity still exists to service the request.

In order to determine the throughput rates for each queue being serviced, log entries were maintained on the API that captured the time that each query was processed, the client's id, the token balance, the execution cost in milliseconds, and the current queue depth for that client. Each experiment was conducted for a 60 second interval and the resulting log data was stored and analyzed. The API was activated for five seconds before each experiment was conducted, which gave the API time to initialize its execution time prediction code.

6 Results

The four algorithms were tested using the same research API and Elasticsearch cluster to evaluate their load balancing properties as well as the throughput achieved in each testing scenario. Each experiment was executed for 60 seconds, and the resulting log files were converted into graphs for analysis. The results of these experiments are grouped together by scenario and presented as line graphs containing a separate colored line for each client.

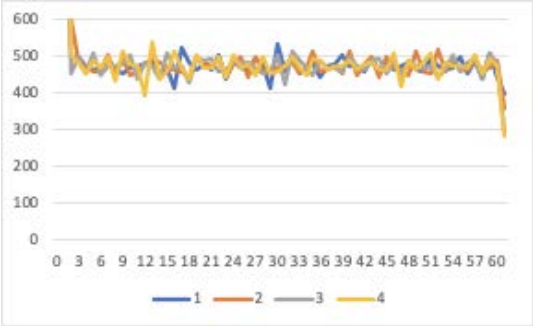
Throughput graphs (figures 3, 5 and 7) show lines representing the sum of the query execution costs in milliseconds for each second of the experiment's run time. Overlapping lines indicate clients that have processed similar amounts of query work during that time period. Queue depth graphs (figures 4, 6 and 8) contain lines indicating the average queue depth per second for a client during the experiment's run time. If a queue depth is consistently zero then it did not experience a sufficient load to saturate the queue. Queues with higher request rates will experience deeper queue depth, but a fair sharing algorithm should moderate requests so that throughput remains similar for all clients with saturated queues.

Copyright © Society for Science and Education, United Kingdom

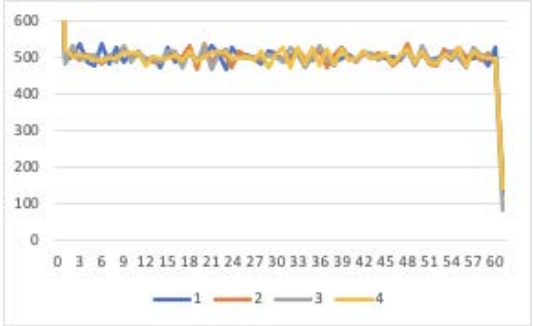
6.1 Balanced Usage Scenario

Each of the four algorithms were tested with the balanced usage scenario, which used 25 threads per user in JMeter to simulate a balanced workload for each client. Because this workload was naturally balanced, very little regulation needed to take place in order to align user service levels with one another. The throughput graphs all look very similar to one another, and the queue depth graphs show all four algorithms maintaining saturated queues throughout the testing period. Figure 3 shows the throughput for each algorithm in this scenario, and figure 4 shows the average queue depth per second for each client.

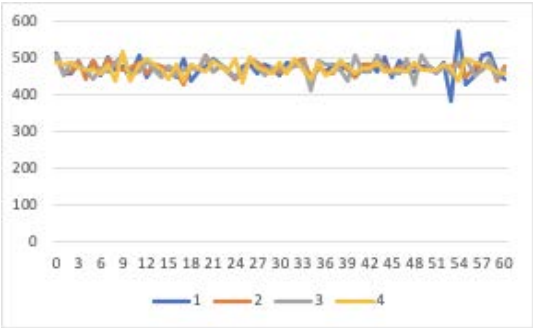
a) Token-Bucket



b) Peng and Plale's MDRR



c) Deficit Round-Robin



d) Single-Server Max-Min Fair DRR

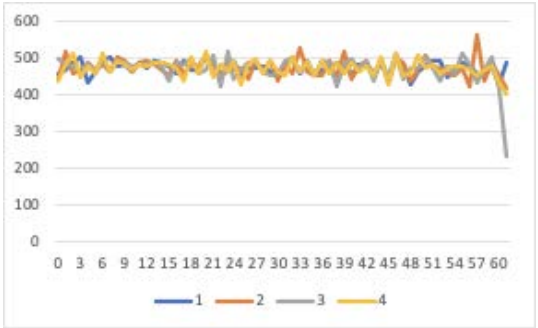
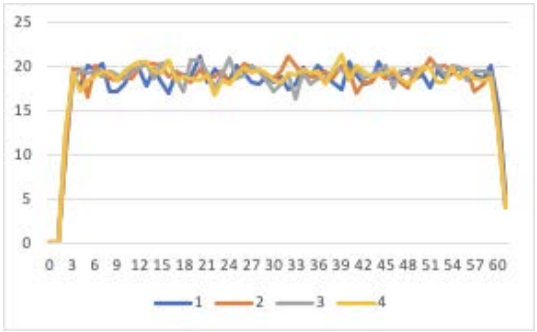


Figure 3. Throughput per second for the balanced scenario.

a) Token-Bucket



b) Peng and Plale's MDRR



c) Deficit Round-Robin

d) Single-Server Max-Min Fair DRR

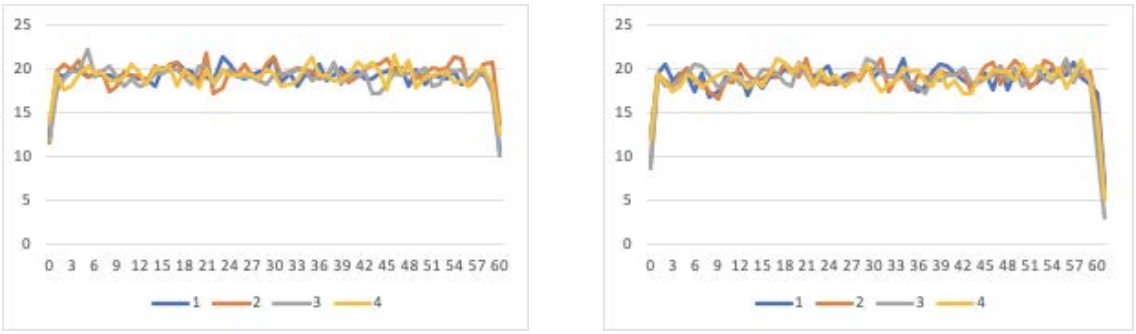


Figure 4. Average queue depth per second for the balanced scenario.

6.2 Unbalanced Usage Scenario

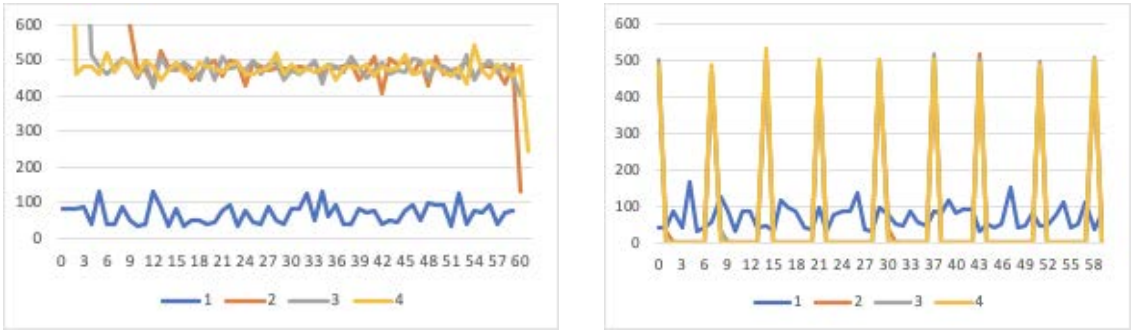
The unbalanced usage scenario highlighted some differences in the behavior of each algorithm. Because this mixture of activity included four users with 1, 10, 15, and 25 threads of JMeter activity respectively, queue saturation was observed only on clients 2 - 4 (Figure 6). The first user's queue depth remained at zero or one for all algorithms for a duration of the test, indicating that the request rate was below the refill rate for each algorithm. All four algorithms were able to successfully handle traffic for client 1, which hovered around 50 ms of execution time per second. Figure 5 shows the throughput for each client in this unbalanced scenario.

The token-bucket algorithm shows some high values during the first few seconds of the experiment. This can be attributed to the few seconds that the API was active before traffic was sent to it, which resulted in a small token balance being accumulated for each user. After this initial balance was used up, token-bucket reverted to servicing each queue at the rate at which it was refilled. The DRR algorithm also exhibited a very similar pattern, with both DRR and token-bucket exhibiting average throughput just under 500 ms per second. The Single-Server Max-Min Fair DRR algorithm also demonstrated a similar throughput pattern, although its average throughput had increased to around 650 ms per second.

Peng and Plale's [1] MDRR algorithm fared the worst in this scenario. Although it was able to consistently service client 1, this client's low usage rate meant that the rest of the algorithms had to wait a considerable amount of time between token refills. Because each client was refilled at the same rate, the token amount granted to the busier clients was used up quickly and so client requests began backing up in the queue once again. This is evident in the queue depth graph, where similar spikes can be seen between token refills.

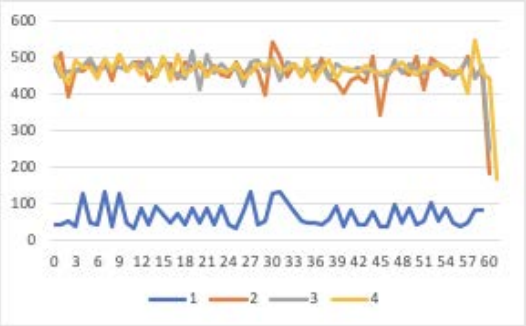
a) Token-Bucket

b) Peng and Plale's MDRR



Copyright © Society for Science and Education, United Kingdom

c) Deficit Round-Robin



d) Single-Server Max-Min Fair DRR

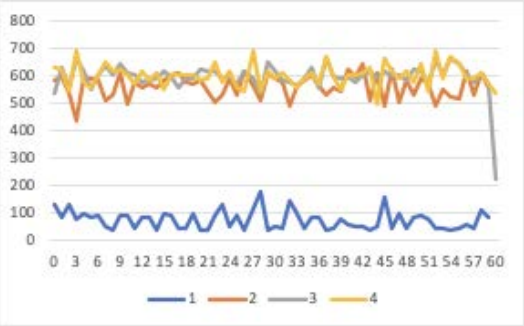


Figure 5. Throughput per second for the unbalanced scenario.

6.3 Idle Client Usage Scenario

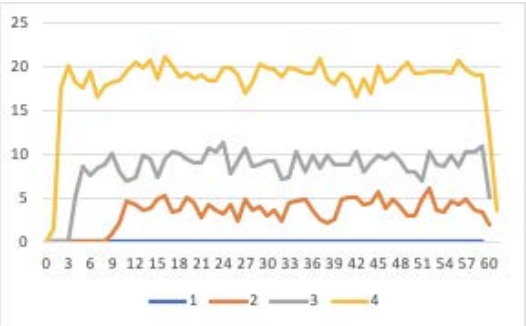
The idle client usage scenario tested an algorithm's ability to deal with extreme differences in usage levels, including several users who made no requests at all during the 60-second testing time period. Clients 3 and 4 made requests using 15 and 25 threads in JMeter, while clients 1 and 2 remained idle. The resulting throughput and queue depth charts are displayed in Figures 7 and 8.

Once again, the token-bucket and DRR algorithms exhibited very similar behavior and throughput once token-bucket's excess tokens acquired at startup had been used up. The Single-Server Max-Min fair DRR algorithm demonstrated throughput twice as high as the token-bucket or DRR algorithms. Its queue depth was also half that of token-bucket or DRR, which is consistent with its behavior of processing requests at twice the normal rate.

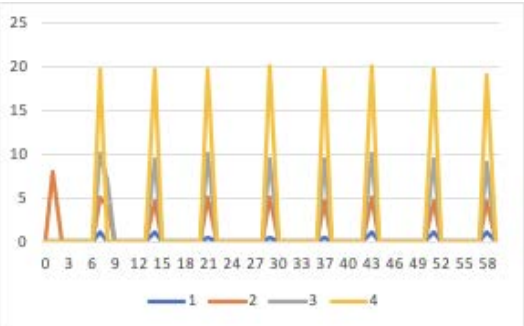
Peng and Plale's MDRR algorithm failed to run for very long during this experiment. Because their algorithm depends on all queues running out of tokens before a refill, the initial 500 tokens were quickly used up by clients 3 and 4. No refill was ever given because no requests came in from clients 1 and 2. The queue depth chart is empty for this algorithm's experiment because log entries were only recorded when a queue had been serviced. Since these queues quickly experienced deadlock, there were no more log entries for the rest of the experiment to indicate the level to which the queue rose before the experiment was halted. However, it is safe to assume that 100% of requests after the first second were queued and not serviced.

Figure 6. Average queue depth per second for the unbalanced scenario.

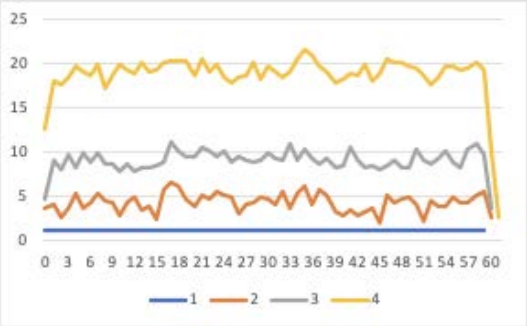
a) Token-Bucket



b) Peng and Plale's MDRR



c) Deficit Round-Robin



d) Single-Server Max-Min Fair DRR

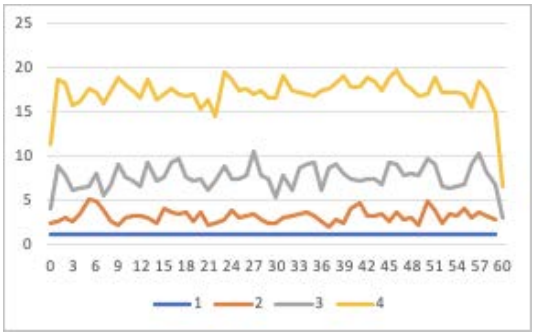
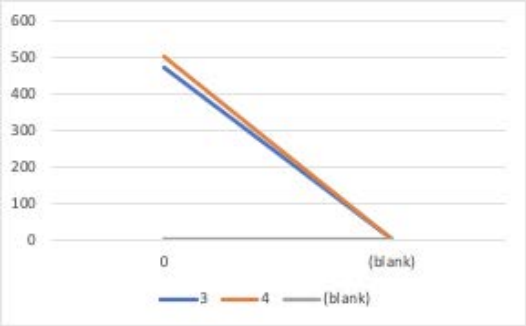


Figure 7. Throughput per second for the idle usage scenario.

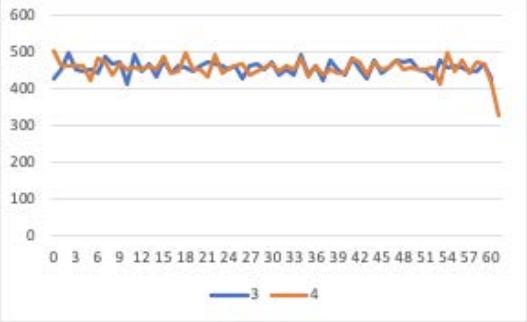
a) Token-Bucket



b) Peng and Plale's MDRR



c) Deficit Round-Robin



d) Single-Server Max-Min Fair DRR

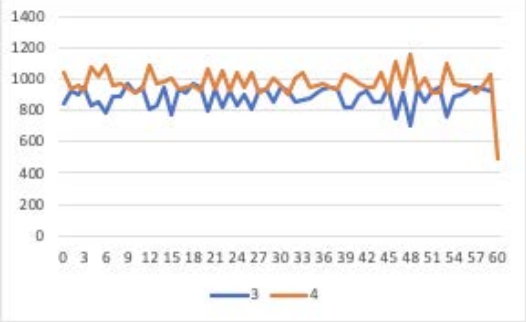
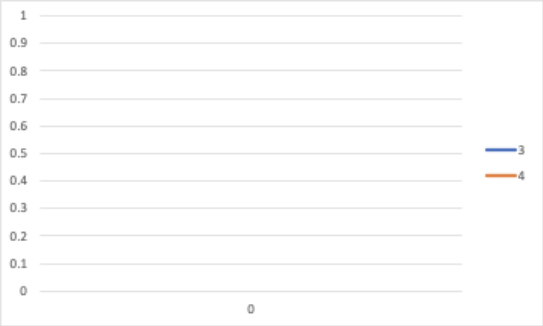


Figure 8. Average queue depth per second for the idle usage scenario.

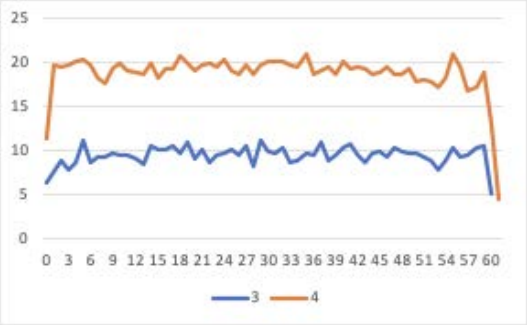
a) Token-Bucket



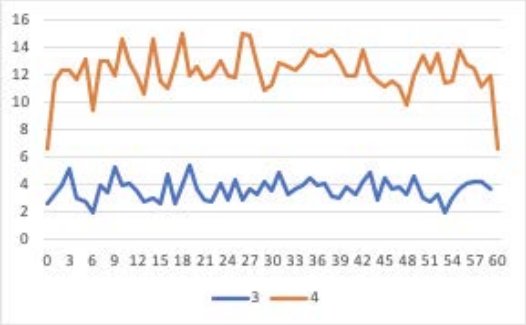
b) Peng and Plale's MDRR



c) Deficit Round-Robin



d) Single-Server Max-Min Fair DRR



7 Discussion

This experiment evaluated the performance of four different algorithms as they were applied to three different usage scenarios in our SaaS API. These scenarios were designed to simulate different usage patterns that often occur in the real world. They also served to highlight potential weaknesses in algorithms that performed well in certain scenarios but had reduced performance or completely failed in other scenarios. In this section we will discuss what we have observed about each algorithm's strengths and weaknesses based on their performance in the balanced, unbalanced and idle client scenarios. Table 1 contains the values for the average query throughput per second processed by each algorithm for each usage scenario. These values will help us compare each algorithm and highlight their differences in performance.

Table 1. Average throughput per second for each algorithm and usage scenario.

	Token-Bucket	Peng and Plale MDRR	Deficit Round-Robin	Single-Server Max-Min Fair DRR
Balanced Scenario	2,025	2,007	1,886	1,887
Unbalanced Scenario	1,574	297	1,426	1,806
Idle Client Scenario	974	16	904	1,841

7.1 Token-Bucket

While not specifically designed to implement fair queuing, the token-bucket algorithm provided very similar usage patterns to those generated by the original deficit round-robin algorithm. Once each user's token balance was exhausted, the token-bucket algorithm performed exactly like deficit round-robin. The request throughput and queue depth values were identical. However, one key difference was the large spikes in throughput were observed in the first few seconds of each scenario's experiment. This can be explained by the initial token balance that was built up before JMeter began sending requests to the API.

If a segment of time were to be selected randomly from a throughput graph for any application of the token-bucket algorithm it would not be guaranteed to be max-min fair. If during that time period one client had a significant token balance while the others had little or no tokens, then the first client would see much higher request throughput while the others could only process requests with throughput equal to their refill rate. Although tokens are given to each user at the same rate, this does not mean that usage happens evenly. This disparity in throughput, that is allocated to each active user with a saturated queue, would result in a Jain's fairness index value that is relatively low. Therefore, this algorithm would not be desirable for use in a scenario where one user's activity cannot adversely impact the performance of other users.

7.2 Peng and Plale's MDRR

Although Peng and Plale [1] named their algorithm "Modified Deficit Round-Robin," it is actually closer in implementation to the token-bucket algorithm. The primary difference between the two is that MDRR refills tokens only when all clients are empty instead of refilling tokens after each timer iteration. This avoids the scenario of one user accruing a much higher balance due to inactivity since all users start with the same balance and are refilled simultaneously. Indeed, for the balanced scenario where all clients are operating with a continuous heavy load this algorithm works very well.

The unbalanced scenario highlights a problem with this algorithm, however. If one client has much less traffic than the others it can cause a situation where the high-volume clients will run out of tokens and the API will completely stop servicing those clients until the slow client finally runs out of tokens. As seen in the throughput graph for the unbalanced scenario, this behavior was observed in our experiment. Throughput for the slow client continued as normal while the other clients experienced spikes of high throughput interspersed with several long pauses. This resulted in an average throughput that was approximately half of what the other algorithms were able to provide.

Testing Peng and Plale's MDRR with the idle client usage scenario resulted in an even more dramatic outcome. With MDRR, token balances are only refilled once all clients are unable to process requests with their existing tokens. Since two of the clients were idle in this scenario, their token balance was never used up which caused the active clients to experience zero throughput for the majority of the experiment. The unbalanced and idle scenarios both highlight substantial problems with this algorithm, making it unsuitable for use in real-world applications.

7.3 Deficit Round-Robin

The deficit round-robin algorithm did a good job of regulating requests evenly in all three usage scenarios. The throughput per client in the balanced, unbalanced and idle scenarios all reached a maximum of approximately 475 ms of execution per second. This is close to the quantum refill rate of 500 ms per

Copyright © Society for Science and Education, United Kingdom

second. The only client with lower throughput levels was the one with only a single thread of requests in JMeter in the unbalanced scenario. That client did not saturate the request queue at all, so their requests were limited by the request rate rather than the rate-limiting algorithm.

In the balanced scenario with four active clients, each client is given a throughput of approximately 475 ms of execution time per second. This resulted in a total throughput of 1,886 ms per second among all clients for that scenario. Although this throughput is the result of an arbitrary value chosen as the refill rate for this experiment, let us assume that this is indeed the maximum throughput that we are able to deliver with our search engine. This means that for the balanced scenario the DRR algorithm is max-min fair, since all clients are experiencing the same level of service and service cannot be added or subtracted from any one client without reducing the equity of the query throughputs.

If we apply this same level of reasoning to the unbalanced and idle scenarios, we see that the algorithm ceases to be max-min fair in these situations. The sum of our throughputs in the unbalanced scenario is 1,426 ms per second, and for the idle scenario it is 904 ms per second. This indicates that excess capacity is available to be used in our search engine but it is not being allocated to the clients. Therefore, while the DRR algorithm succeeds at balancing usage in all scenarios it is only max-min fair in situations where all clients have requests backed up and are consistently maintaining a queue depth.

7.4 Single-Server Max-Min Fair DRR

The Single-Server Max-Min Fair DRR implementation resulted in fair throughput that consistently maximized our Elasticsearch connection for every usage scenario. As with every other algorithm when testing the balanced scenario, the Single-Server Max-Min Fair DRR algorithm delivered close to 500 ms of throughput per second to each of the four clients. However, in the unbalanced scenario where only three of the four clients were saturating their queues this algorithm redistributed the extra throughput resulting in approximately 650 ms per second for the three saturated queues and approximately 50 ms per second for the final queue. This still resulted in all 2,000 ms of capacity being utilized.

In the idle client scenario, the Single-Server Max-min Fair DRR algorithm reallocated all the bandwidth that was not used by the idle clients. This allowed the two active clients to effectively double their throughput while still maintaining a fair balance between the active queues. The Single-Server algorithm was implemented with a floor function that rounded each bonus allocation down to the nearest integer. This was done for simplicity in logging, though there is no practical reason why fractional credits cannot also be used. Over time, these fractional values would add up to additional whole credits that could be used to service a request.

Although this algorithm performed admirably in each scenario we tested, some may be tempted to argue that it does not always achieve a 1.0 on Jain's fairness index because the allocations it provides at each moment in time are not always equal. Although the algorithm does allocate more throughput to some users than others, it always allocates throughput to uses with saturated queues evenly. If the value of "n" users is taken to mean the number of users with saturated queues, then both the original and the Single-Server Max-Min fair variations of DRR would achieve 1.0 scores on Jain's index. Underutilized or idle clients fall under the scenario mentioned by Shreedhar and Varghese [3] when they said that "if the queue for a flow is periodically empty and the flow misses round-robin opportunities, these are opportunities that the flow could not use anyway." The Single-Server Max-Min Fair DRR algorithm uses this excess

throughput to its advantage, resulting in an algorithm that is just as fair as the original DRR but with maximized throughput in all usage situations.

8 Conclusion and Future Works

This research identified four different algorithms which could be useful for enforcing fair sharing on a Search as a Service (SaaS) API: token-bucket, original DRR, Peng and Plale's modified DRR, and Single-Server Max-Min Fair DRR. A test API was constructed to evaluate each algorithm and collect the results of the query throughput for several different scenarios. Although all four algorithms behaved equally well for a balanced throughput scenario, the Single-Server Max-Min Fair DRR algorithm was the only one that delivered both consistently high throughput and consistently fair allocations for each client. The algorithm achieved this by re-allocating unused throughput to clients with saturated queues so that a max-min allocation was achieved.

Several things can be done as future work related to this research. First, although various usage patterns were used in this research, each scenario involved queries that were being submitted at a consistent rate. There is room for future research to be conducted to examine how these algorithms behave when usage from each client varies substantially over a longer period of time. Although a poisson-based random timer in JMeter would just introduce noise to the graphs, a continuously increasing or decreasing number of thread counts would simulate increasing user demand over time. This would simulate the ebbs and flows of user activity during a normal day.

More research is also needed to determine how the Single-Server Max-Min Fair DRR algorithm can be implemented when the API is served in a multi-server environment. Although a load balancer can distribute requests evenly among all API nodes, each node would still need to communicate with one another in order to compute a max-min allocation among all the nodes. This would likely require a centralized state to be shared among all nodes, similar to the central coordination system mentioned in Peng and Plale's implementation. This shared state may also need to maintain the token balances for all clients so that fairness can be guaranteed among the actions of all nodes. Since Khamse-Ashari et al.'s [4] work originally involved using their algorithm in a multi-server environment, their work should be examined further to determine if their methods could be applied to this sort of implementation.

ACKNOWLEDGEMENTS

This work has been partially supported by the Askew Institute of The University of West Florida, Pensacola, Florida.

REFERENCES

[1]. A. Peng and B. Plale, "A Mulit-tenant Fair Share Approach to Full-text Search Engine", in *2016 Seventh International Workshop on Data-Intensive Computing in the Clouds (DataCloud),* pp. 40-50, doi: 10.1109/datacloud.2016.010.

[2]. Amazon. "Amazon API Gateway Developer Guide: Throttle API requests for better throughput." https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-request-throttling.html (accessed Nov. 17, 2020).

Copyright © Society for Science and Education, United Kingdom

[3]. M. Shreedhar, G. Varghese, "Efficient fair queuing using deficit round-robin," *IEEE/ACM Transactions on Networking*, 4(3), pp. 375–385, doi: 10.1109/90.502236.

[4]. J. Khamse-Ashari, G. Kesidis, I. Lambadaris, B. Urgaonkar, and Y. Zhao. "Max-Min Fair Scheduling of Variable-Length-Packet-Flows to Multiple Servers by Deficit Round-Robin," in *2016 Annual Conference on Information Science and Systems (CISS),* pp. 390-395, doi: 10.1109/CISS.2016.7460534.

[5]. P. Dordal, *An Introduction to Computer Networks.* Chicago, Illinois: Loyola University Chicago, 2020. [E-book] Available: http://intronetworks.cs.luc.edu/

[6]. I. Marsic, *Computer Networks: Performance and Quality of Service*. New Brunswick, New Jersey: Rutgers, 2013. [E-book] Available: https://www.ece.rutgers.edu/~marsic/books/CN/

[7]. R. Jain, D. Chiu, and W. Hawe. "A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems," *ArXiv*, Sept. 1998, doi: cs.NI/9809099.

[8]. J. Nagle. "On Packet Switches with Infinite Storage," *IEEE Transactions on Communications*, 35(4), pp. 435–438, doi: 10.1109/tcom.1987.1096782.

[9]. A. Demers, S. Keshav, and S. Shenker. "Analysis and simulation of a fair queueing algorithm," *ACM SIGCOMM Computer Communication Review*, 19(4), pp. 1–12, doi: 10.1145/75247.75248.

[10]. P.E. McKenney, "Stochastic Fairness Queuing," *Internetworking: Research and Experience*, 2(January), pp. 113–131.

[11]. G. Varghese, *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices.* Amsterdam: Morgan Kaufmann, 2005, doi: 10.1016/B978-012088477-3/50017-5.

[12]. W. Tong, J. Zhao, "Quantum Varying Deficit Round Robin Scheduling Over Priority Queues," 2007 International Conference on Computational Intelligence and Security (CIS 2007), Computational Intelligence and Security, 2007 International Conference On, pp. 252–256. doi: 10.1109/CIS.2007.182.

[13]. J. Heinanen and R. Guerin. "A Two Rate Three Color Marker", IETF. https://tools.ietf.org/html/rfc2698 (accessed Jan. 11, 2021).

[14]. C. Macdonald, N. Tonellotto, and I. Ounis. "Learning to predict response times for online query scheduling," in *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '12)*, doi: 10.1145/2348283.2348367.

[15]. E.L. Fridge and S. Bagui. "Estimating Query Timings in Elasticsearch," submitted for publication.

[16]. *Datasets*, Hathitrust. [Online]. Available: https://www.hathitrust.org/datasets

[17]. "Apache JMeter." Apache Software Foundation. https://https://jmeter.apache.org/ (accessed Sept. 15, 2020).

[18]. *AOL Search Data Collection*, Archive.org. [Online]. Available: https://archive.org/details/AOL_search_data_leak_2006