# Janus: A Generic QoS Framework for Software-as-a-Service Applications

Qingye Jiang
School of Information Technologies
The University of Sydney
Sydney NSW 2006, Australia
Email: qjiang@ieee.org

Young Choon Lee
Department of Computing
Macquarie University
Sydney NSW 2109, Australia
Email: young.lee@mq.edu.au

Albert Y. Zomaya
School of Information Technologies
The University of Sydney
Sydney NSW 2006, Australia
Email: albert.zomaya@sydney.edu.au

***Abstract*—The move from the traditional Software-as-a-Product (SaaP) model to the Software-as-a-Service (SaaS) model is apparent with the wide adoption of cloud computing. Unlike the SaaP model, the SaaS model delivers a diverse set of software features directly from public clouds to a large number of arbitrary users with varying quality of service (QoS) requirements. QoS is typically assured by admission control. However, there are two outstanding issues with traditional QoS systems: (1) they are usually designed and developed with a special purpose, making them difficult to be reused for other use cases; and (2) they have limited scalability (i.e., vertical scalability) due to the write-intensive nature of admission control workload. In this paper, we present Janus - a QoS framework that is generic and scalable for SaaS applications taking full advantage of cloud's inherent horizontal scalability (scaling-out). Janus uses a multi-layer architecture to eliminate the communication between nodes (being scaled out) in the same layer achieving horizontal scalability without sacrificing vertical scalability. Janus ensures accurate admission control (QoS decisions) using a distributed set of leaky buckets with a refill mechanism. Janus also adopts a key-value request-response mechanism for easy integration with the actual application. We extensively evaluate Janus on AWS cloud with both Apache HTTP server benchmarking tool and a photo sharing web application. Our experimental results demonstrate that (a) Janus achieves linear scalability both vertically and horizontally, and (b) Janus can be integrated with existing applications with a minimum amount of code change. In particular, Janus achieves more than 100,000 requests per second with only 10 nodes (4 vCPU cores on each node) in the QoS server layer and 90% of the admission control decisions were made in 3 milliseconds.**

***Keywords*—admission control, distributed system, scalability, write intensive workload**

## I. Introduction

In QoS management, admission control refers to the action of determining whether a particular request can be admitted such that all admitted requests can be served with the desired performance. Traditionally, QoS is implemented as an integral part of the device or system, hard coded with a very limited set of QoS rules specifically tailored for the device or system [1], [2]. Different devices or systems need to implement their own QoS module, which cannot be shared with or ported to other devices or systems. This is not an issue when the software is sold with the traditional Software-as-a-Product (SaaP) model, where the software is installed on the end user's own computing resource. In recent years, there is a trend to deliver software features directly from public clouds in the form of Software-as-a-Service (SaaS). With the SaaS model, software is usually deployed on a centralized system, providing service to multiple tenants. The increased workload on the SaaS application leads to the increased pressure on the QoS system. As such, the scalability of the QoS system becomes an important issue in the move to the SaaS model.

Admission control differs from other API services (with a "write once, read many" pattern) in that most requests are composite requests that include 'one read operation and one write operation'. In a service oriented architecture, the end user purchases a certain capacity (quota) from the service provider, which can be the number of requests per second/minute/hour. When the user makes an API call, the QoS system needs to perform (a) a read operation to get the current quota, (b) deduct one from the current quota, and (c) immediately perform a write operation to set the new quota. This is a write intensive workload that requires read-after-write strong consistency. Adding read replicas to the data layer introduces complexity and demands more computing resources (network bandwidth in particular), but cannot bring performance gain to the system due to the eventual consistent nature of the read replicas. Therefore, the horizontal scalability of QoS systems remains a challenging problem.

There exists very little literature on the horizontal scalability of QoS systems. Most existing attempts come from the telecommunication industry, focusing on improving the QoS of router devices [3], [4]. Papazoglou et. al. [5], [6], [7] list QoS as an important research direction in service-oriented computing. To the best of our knowledge, this paper is the first work for SaaS applications addressing two outstanding issues with traditional QoS systems: (1) their application-specific nature and (2) their limitation to vertical scaling.

In this paper, we present Janus[1] as a generic and scalable QoS framework. Janus employs a multi-layer architecture to create multiple independent partitions within a QoS system (Figure 1). Such a design eliminates the communication between different nodes in the same layer, improves both scalability and performance the QoS system. The specific contributions of this paper are:

- We design and implement a multi-layer architecture that includes a load balancer layer, a request router layer, a QoS server layer, and a databases layer. Such a multi-layer design is effective in segregating

[1]Janus is a gatekeeper god in ancient Roman religion and myth.

QoS requests into multiple independent partitions to achieve both vertical and horizontal scalability.

- We use a distributed set of leaky buckets with a refill mechanism to make QoS decisions. This provides accurate admission control according to the quota the user purchases, but still allows occasional burst operations when the user accumulates credit.
- We use a key-value request-response mechanism so that Janus can be integrated with a wide range of devices, applications, or services that demand QoS.
- We demonstrate that Janus achieves linear scalability both vertically and horizontally. Existing web application can be easily integrated with Janus via a simple wrapper layer.

In the performance evaluation, a modified version of the Apache HTTP server benchmarking tool (which is commonly known as "ab") is used to generate massive concurrent QoS requests to Janus. Janus achieves more than 100,000 requests per second with only 10 nodes (4 vCPU cores on each node) in the QoS server layer. In the application integration evaluation, we use a photo sharing web application to demonstrate how to integrate Janus with existing applications. We observe that 90% of the admission control decision were made in 3 milliseconds, which is only a small overhead as compared with the application's own latency. The horizontal scalability of various layers, the painless integration process, plus the very low latency overhead, prove that Janus can be used to provide QoS service for a wide range of SaaS applications.

The rest of this paper is organized as follows. Section II describes the architecture design of Janus. Section III describes the implementation of Janus. Section IV describes how Janus can be used in various use cases. In Section V, we evaluate the performance of Janus, using AWS cloud as the test environment. Section VI reviews related work, followed by our conclusions in Section VII.

## II. Architecture Design

Janus consists of four major layers – **Load Balancer**, **Request Router**, **QoS Server**, and **Database** (Figure 1). The load balancer layer distributes QoS requests from the QoS client across multiple request router nodes/instances. The request router layer further segregates the QoS requests to the appropriate QoS server for decision making. The QoS server layer makes the admission control decision based on pre-defined Qos rules in the database. The database layer stores QoS rules as well as other system properties that are used for initialization and check pointing.

A QoS request comes with a QoS key. The composition of the QoS key depends on the nature of the service provided to the end user. For a web service with a single feature, different user might purchase different access rates, then the QoS key can be the user identification. For a NoSQL database service, a particular user might purchase different access rates for different databases, then the QoS key can be the combination of the user identification and the database name. The QoS response is a boolean value where *TRUE* indicates access should be allowed and *FALSE* indicates access should be denied.

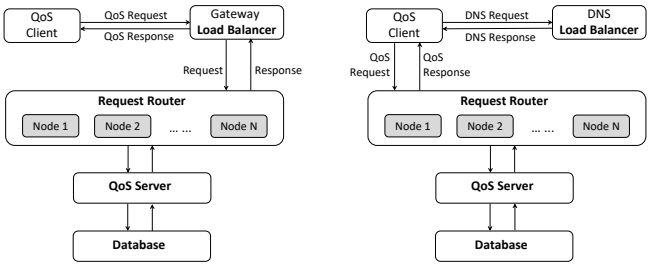



Fig. 1: Janus Architecture Design: (a) with Gateway Load Balancer and (b) with DNS Load Balaner.

### A. Load Balancer

The load balancer serves as the service endpoint for Janus. The load balancer can be implemented as either a gateway load balancer (Figure 1a) or a DNS load balancer (Figure 1b).

As shown in Figure 1a, a gateway load balancer is an appliance that accepts QoS requests via HTTP/HTTPS from the QoS client, then distributes the requests across multiple request router nodes behind the load balancer. The load balancer needs to support the HTTP/HTTPS protocol, because the request router nodes only accept HTTP/HTTPS requests. Different load balancer might support different routing algorithms. For example, the round robin routing algorithm distributes incoming HTTP/HTTPS requests to the back end nodes one by one, while the least connections algorithm distributes incoming HTTP/HTTPS requests to the back end node with the least number of outstanding HTTP/HTTPS requests. If deployed in an on-premise data center, the load balancer can be a physical device or a software appliance. If deployed on public clouds, most public cloud service providers provide load balancing services, such as the Amazon Elastic Load Balancer (ELB).

As shown in Figure 1b, a DNS load balancer is a DNS server that accepts DNS query requests from the operating system running the QoS client, then returns a list of IP addresses representing the request router nodes. Most DNS servers support the round robin routing algorithm. That is, with each DNS response, the IP address sequence in the list is permuted. Usually, the QoS client attempts to connect the request router with the first IP address returned from the DNS query. As such, on different connection attempts, the same QoS client would connect to different request router nodes, thus distributing the requests among different request router nodes.

DNS load balancing occurs at a higher level in that DNS resolution happens before the HTTP/HTTPS connection is made. As such, it is possible to use a combination of DNS load balancing and gateway load balancing to improve the performance of the whole system. For example, a gateway load balancer usually has a single IP address and certain load balancing capacity limits such as throughput or requests per second. In large-scale deployments, multiple gateway load balancers can be deployed. The client connects to different gateway load balancer nodes via DNS resolution, while the gate load balancer nodes further distribute the requests to the worker nodes in the back end.

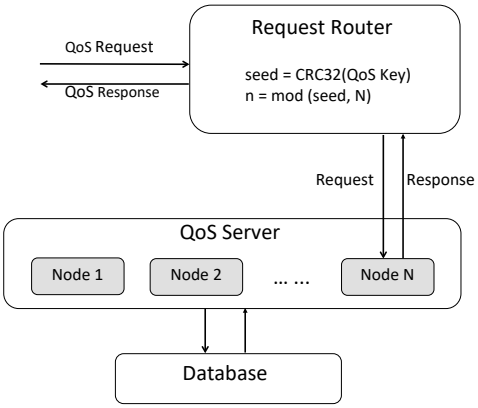



Fig. 2: Request Routing Algorithm.

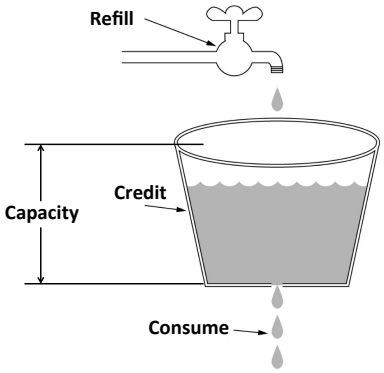



Fig. 3: Leaky Bucket QoS Algorithm.

### B. Request Router

The request router is a stateless web application that receives incoming QoS requests from the QoS client - probably through a gateway load balancer. As shown in Figure 2, the request router determines which QoS server in the back end should be contacted using a hashing algorithm. The input parameter to the hashing algorithm is the QoS key, which is a string. The hash function uses a 32-bit cyclic redundancy checksum (CRC) algorithm to generate an integer from the QoS key, then determines the target QoS server using the modulo operation. The request router forwards the QoS request to the selected QoS server for processing. With a fixed number of QoS servers in the back end, QoS requests with the same QoS key are always routed to the same QoS server, regardless of which request router node is handling the request segregation. This is very similar to a traditional web application with a database cluster in the back end, where the web front end picks the database node via a hashing algorithm. Because of the stateless nature of the request router, it can be dynamically scaled in or scaled out based on the actual workload.

### C. QoS Server

The QoS server receives incoming QoS requests from the request router, determines whether access should be allowed or denied, then produces the QoS response. At the core of the QoS server is the leaky bucket [8] QoS algorithm, as shown in Figure 3. For each QoS key we maintain a leaky bucket with a certain capacity, where the water level represents the remaining credits. The bucket is being refilled at a constant rate, which is the access rate the user purchases from the service provider. When the user makes an API call to the service, the user consumes one credit from the bucket. From a higher level, the QoS server layer can be visualized as a set of distributed leaky buckets.

Assuming that the leaky bucket is initially fully filled with an initial credit equal to the capacity *C* of the bucket, the bucket is being refilled at rate *A* and consumed at rate *B*. Then the available credit at any time *t* can be noted as *f(t)*, where

$$f(t) = C + (A - B) * t \tag{1}$$

When the refill rate A is greater than the consume rate B, the credit gradually accumulates but it cannot exceed the capacity of the bucket *C*. When the refill rate *A* is smaller than the consume rate *B*, the credit gradually depletes but it cannot be less than zero. As such, the value of *f(t)* is [0, C], or

$$0 <= f(t) <= C \tag{2}$$

The QoS server maintains a local QoS rule table. A QoS rule includes the QoS key, the capacity of the leaky bucket, the refill rate, and the current credit. Each QoS rule is represented by a leaky bucket. When the QoS server receives a QoS request, it looks into the QoS rule table for the particular QoS rule with that particular QoS key. If the QoS rule does not exist in the QoS rule table, the QoS server creates the QoS rule with the information from the database in the back end, then puts the newly created QoS rule into the QoS table. The QoS server looks into the leaky bucket associated with the QoS rule to make QoS decisions. If the current credit is greater than zero, it returns *TRUE* as the QoS response, otherwise it returns *FALSE* as the QoS response.

With the leaky bucket QoS algorithm, Janus allows the end user to accumulate unused credits for burst traffic. For example, a particular end user might purchase an access rate of 100 requests per second, but the service provider sets the capacity of the leaky bucket to 1000. If the end user stops accessing the service for more than 10 seconds and the leaky bucket becomes full, then the end user is allowed to achieve higher request rates such as 500 requests per second – assuming that the service itself is capable of supporting it – until the leaky bucket is depleted.

### D. Database

The database layer is a traditional relational database, which stores all the QoS rules in a table. For high-availability a master/slave configuration can be setup. For a large-scale service-oriented application, the number of QoS rules would be quite large because (a) the service might have a large user base, and (b) multiple rules can be configured for a single user.

Over time, existing rules can be deleted or modified, and new rules can be created, depending on the needs from each individual end user. As such, it would be expensive for each

QoS server to maintain a local copy of all the QoS rules in its local QoS rule table and keep them in synchronization with the QoS rule table in the database. Instead, when the QoS server receives the request for a particular QoS key for the first time, it makes a query to the database for the corresponding QoS rule and creates a new leaking bucket for the QoS key in the local QoS table. With this approach, new QoS keys/rules are immediately effective as soon as they are added to the database. When the query to the database returns an empty result, the QoS key requested does not exist in the database. Depending on the nature of the service provided, this can be a guest/test access request, or an unauthorized access request. In this case, the QoS Server sets up a new leaking bucket for the QoS key in the local QoS table with the default QoS rules. For example, the default QoS rules can be a combination of zero capacity and zero refill rate to deny access, or a combination of a small capacity and a small refill rate to grant limited access.

To keep the local QoS rules in synchronization with the database, the QoS server makes queries to the database with the QoS keys in the local QoS rule table with a configurable update interval, then updates the local QoS rule table with the query results. For check-pointing, the QoS server also updates the database with the current credit for the QoS keys in the local QoS table with a configurable update interval. When a particular QoS server fails, the handling of the failed QoS server depends on its high-availability configuration. When high-availability is enabled for the QoS server, minimum downtime can be achieved because the new master node already has an up-to-date local QoS table. When high-availability is not enabled for the QoS server, a replacement QoS server is launched for the failed QoS server. The replacement QoS server gradually reinitializes the local QoS rule table by making queries to the database when it receives requests with new QoS keys. With the check-pointing mechanism, the replacement QoS server will use the last check-pointed credit information from the database as the initial credit value. Because the total number of QoS servers remains the same, the hash results – and hence the routing rules – produced by the request router layer remain the same. As such, a failed QoS server is a localized failure in that it does not impact the normal operation of other QoS servers in the system.

With this approach, the QoS servers communicate with the database only when (a) receiving requests with new QoS keys, (b) synchronizing with the database with a configurable update interval, and (c) check-pointing to the database with a configurable update interval. These are infrequent access requests and do not present a heavy workload on the database. Each QoS server only maintains a subset of the global QoS rule table, effectively reducing the memory requirement on the QoS server. There is no communication between the QoS servers in Janus. They are totally unaware of the existence of each other. Furthermore, there is no disk I/O on either the request router layer or the QoS server layer.

## III. Implementation

Janus is designed to be generic and can be deployed on either public clouds or on-premise data centers. In this paper, we carry out our development and testing on AWS only. We make the effort to reuse existing technologies as much as possible. If the functional and performance requirements of a particular component can be met with an existing technology, we directly use the existing technology in our implementation. For example, we use Route53 for DNS load balancing, ELB for gateway load balancing, RDS for the database layer, and EC2 instances as the nodes in the request router and QoS server layers. In subsequent sections, all mentioning of "node" or "instance" refers to EC2 instances launched in the ap-southeast-2 (Sydney) region. The operating system running on the EC2 instances is 64-bit Ubuntu Server 16.04.3 LTS.

Janus can also be deployed on other public clouds or on-premise data centers. Most public cloud service providers offer services similar to EC2, RDS, ELB, and Route53. If deployed in on-premise data centers, the load balancer layer can be implemented with a hardware load balancer with support for the HTTP protocol, while nodes in other layers can be implemented on either physical servers or virtual machines.

### A. Load Balancer

In the case of DNS load balancing, we use the Amazon Route53 DNS service as the load balancing layer. Janus is represented by a domain name whose *A* record includes the IP addresses of the request router nodes. With each DNS query request, the IP address sequence in the DNS query result is permuted.

In the case of gateway load balancing, we use the Amazon ELB as the load balancing layer. Janus is represented by a domain name of the ELB. It should be noted that ELB is in fact a combination of DNS load balancer and gateway load balancer. with AWS, each AWS region has multiple isolated locations known as Availability Zones (AZs). ELB supports load balancing across multiple availability zones by creating one or more load balancer nodes in each availability zone. The DNS endpoint for the ELB is managed by the Amazon Route53 DNS service. The DNS query result includes the IP addresses of all load balancer nodes allocated to the ELB. With each DNS query request, the IP address sequence in the DNS query result is permuted.

### B. Request Router

The request router is implemented with PHP (version 7.0.22) running on the Apache web server (version 2.4.18). It receives QoS requests from the QoS client via HTTP. For admission control purposes, QoS requests need to be processed in a very fast manner. For performance considerations, the request router uses UDP instead of TCP to communicate with the QoS server in the back end. Although the TCP protocol ensures reliable connection and communication, the overhead of opening and closing a large volume of short-lived TCP connections is too expensive. With its connect-less nature, the UDP protocol can achieve higher communication efficiency.

However, the UDP protocol does not ensure reliable communication. To compensate for the possible data lost in UDP communication, we use a 100-microsecond communication timeout and a maximum number of 5 retries. In the best case, the communication between the request router and the QoS server is completed at the first attempt within 100 microseconds. In the worse case, the communication between the request router and the QoS server fails after 5 retries, which is 500 microseconds. When the request router fails to obtain a

response from the QoS server after 5 retries, the request router returns a default reply to the QoS client.

### C. QoS Server

The QoS server is implemented with Java running on OpenJDK version 1.8.0. The major components include (a) the local QoS table, (b) the UDP listener thread, (c) the worker threads, and (d) high-availability and system maintenance threads. The local QoS table is represented by a synchronized hash map, where the key is the QoS key and the value is the leaky bucket representing the corresponding QoS rule. The local QoS table is maintained by a house-keeping thread, which refills the leaky buckets in the local QoS table with predefined intervals. The UDP listener thread receives UDP packets from the request router and pushes the packets into a FIFO. There are *N* worker threads polling the FIFO for packets to process, where *N* equals to the number of vCPU's available on the QoS server. Based on the state of the leaky bucket, the worker thread produces the QoS response and sends it back to the request router via UDP. The worker thread does not care about whether the request router receives the response or not. If the request router does not receives a response within the timeout period, the requester router will actively resend the same request to the QoS server until a response is received, or has reached the maximum number of retries. The high-availability thread waits for incoming connections from slave nodes, and sends back the current local QoS table upon request. The system maintenance thread periodically checks with the database to see if there are any updates to the rules in the local QoS table. When updated rules are found, the corresponding leaky bucket representing the corresponding QoS is updated with the latest values.

When high-availability is desired, an optional slave node can be configured for each QoS server. The slave node continuously replicates the local QoS rule table from the master node at a configurable interval. The request router identifies the QoS server nodes in the back end via their DNS names. The master-slave configuration is managed by the Amazon Route53 DNS health check and fail over mechanism. The DNS resolution result for a master-slave QoS server pair only contains the IP address of the master node. In the event that the master node fails, the slave node becomes the new master node, and its IP address replaces the IP address of the failed old master node. The new master node already has an up-to-date version of the local QoS table, allowing the QoS server to continue functioning with minimum interruption. After a successful fail over, we can terminate the original failed master node and launch a new slave node to form a new master-slave pair.

### D. Database

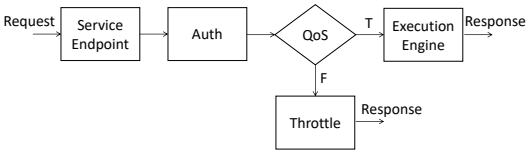
The database is implemented on top of MySQL.The QoS rules table includes four columns - the QoS key, the refill rate, the capacity of the leaky bucket, and the remaining credit in the bucket. The size of a QoS rule is approximately 100 bytes. The size of the table can be easily fit into the memory of a decent database server. In most cases, the QoS server retrieves and updates a single record in the QoS rules table at a time. As such, we set the QoS key as the primary key in the QoS rules table to speed up queries. To fully utilize the abundant memory available on the database server, on system startup we issue a query "SELECT * FROM qos_rules" to load the full table into the memory.

In our implementation we use an RDS instance running MySQL 5.7. The underlying operating system for RDS MySQL instances is Amazon Linux instead of Ubuntu. When high-availability is desired, the RDS instance can be deployed in a Multi-AZ fashion. In this case, the RDS instance includes a master node in one availability zone and a standby node in another availability. Similar to the master-slave fail over mechanism in the QoS server, the RDS instance is represented by a DNS name managed by Amazon Route53. When the master node is in a healthy state, the DNS resolution result contains only the IP address of the master node. When the master node fails, the standby node is promoted to be the new master node, and the DNS resolution result contains the IP address of the new master node.

## IV. Use Cases

In a typical service-oriented architecture without QoS support, the end user makes API calls to the service endpoint. The service performs the necessary authentication and authorization, then passes the API calls to the execution engine for processing. The execution engine produces the response, which is returned to the end user. To add QoS support, for each API call the service makes a request to Janus with the QoS key. If Janus returns *TRUE*, the service passes the API call to the execution engine for processing; if Janus returns *FALSE*, the service actively throttles the API call and returns a throttling message to the end user. Figure 4 shows the flow diagrams for both service-oriented architectures with and without QoS support.


(a) Service without QoS support




(b) Service with QoS support

Fig. 4: Flowchart for services with and without QoS Support.

Janus can be integrated with existing applications with minimum code changes. For example, a service provider may grant access to its web services based on the end user's username or IP address. When there is no QoS in place, API calls from certain end users may exceed the desired request rate, resulting in resource depletion in the execution engine. With QoS in place, the service provider can use the end user's username or IP address as the QoS key to sell different access rate to different end users. In this case, the access rate is represented by the refill rate of in the QoS rule.

We use a photo sharing web application developed in PHP to demonstrate how to integrate Janus with existing applications. The index page of the web application performs the following steps: (a) obtains the IP address of the end user,

TABLE I: EC2 instance types.

| | vCPU Cores | Memory (GB) | Network (Mbps) | Price (USD/hr) |
|---|---|---|---|---|
| c3.large | 2 | 3.75 | 250 | 0.188 |
| c3.xlarge | 4 | 7.5 | 500 | 0.376 |
| c3.2xlarge | 8 | 15 | 1000 | 0.752 |
| c3.4xlarge | 16 | 30 | 2000 | 1.504 |
| c3.8xlarge | 32 | 60 | 10000 | 3.008 |
| r3.xlarge | 4 | 30.5 | 500 | 0.455 |
| r3.2xlarge | 8 | 61 | 1000 | 0.910 |

(b) connects to a Memcached server for session sharing, (c) connects to a MySQL server to query for the latest N user uploaded images, and (d) generates the HTML response based on the above-mentioned query results. In this demo, we use the IP address of the end user as the QoS key and perform the QoS check before performing step (b). As shown in the following code snippet, the integration between Janus and the existing web application can be easily done via a wrapper. In real life applications, the service provider can define different QoS rules for a list of known IP address, while requests from unknown IP addresses will use the default QoS rule.

```
<?php
include("qos_client.php");
$key = $_SERVER['REMOTE_ADDR'];
$qos = qos_check($key);
if ($qos) {
// Allow access to original code
include("original_index.php");
}
else {
// Throttling
header("HTTP/1.1 403 Forbidden");
}
?>
```

A wide range of use cases can be derived from the above-mentioned example. Using IP address as the QoS key allows reasonable anonymous browsing, at the same time mitigating the threats from malicious or unintentional surge requests. Crawlers from certain search engines might also produce occasional burst workloads, quickly depleting the computing resources for a website. QoS rules can be setup with the User-Agent string in the HTTP request header as the QoS key, allowing access from search engines with a reasonable access rate. For a NoSQL database service, an end user might purchase different access rates for different databases in its account, then the QoS key can be the combination of the user identification and the database name.

## V. Evaluation

In this section, we study the performance characteristics of Janus, with a focus on the vertical and horizontal scalability of the request router and the QoS server layer. Due to page limits we do not present results on the database layer because (a) it sits at the tail of the QoS system, (b) it receives only a very small amount work workload, and (c) it does not have a significant impact on the performance of the QoS system.

Our evaluation study is carried out on AWS and involves a variety of EC2 instance types with a web application. Table I describes the configuration of the various EC2 instance types being used in the evaluations. The largest test setup include 15 EC2 instances with a total number of 200 vCPU cores. When gateway load balancing is used, the load balancer layer is an ELB with an HTTP listener. The database layer is a r3.2xlarge RDS MySQL instance with a Multi-AZ master-standby high-availability configuration. There are 100 M QoS keys in the database, each QoS key is associated with a different QoS rule ranging from 1 request per second to 10 K requests per second. The size of the test data in the database is approximately 10 GB, which is significantly less than the 61 GB memory available on the RDS instance.

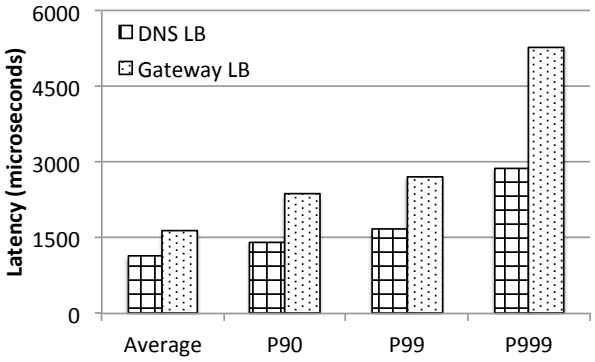



Fig. 5: Gateway Load Balancer vs DNS Load Balancer.

To generate workload on Janus, the client side includes a dedicated EC2 instance fleet with ten c3.8xlarge nodes. The clients run a modified version of the Apache HTTP server benchmarking tool (which is commonly known as "ab") to generate a large amount of concurrent QoS requests with different QoS keys. To understand the results obtained from these tests, we refer to the web service test results reported by Zhao et. al. [9] in 2016.

### A. Load Balancer

This evaluation compares the performance difference between the gateway load balancer and the DNS load balancer. In order to do this, we use two c3.8xlarge nodes in the QoS server layer and two c3.8xlarge nodes in the request router layer. In the case of gateway load balancer, we use an ELB with an HTTP listener and Janus is represented by the DNS endpoint of the ELB. In the case of DNS load balancer, we use the Amazon Router53 DNS service and Janus is represented by a domain name whose A record contains the IP addresses of the two request router nodes. On two separate c3.8xlarge nodes, we run a QoS client in a single-thread fashion, creating a modest workload at approximately 1000 requests per seconds. Each QoS client makes 100,000 QoS requests to Janus and records the round-trip latency of each QoS request.

Figure 5 presents the results from the load balancer tests. For the DNS load balancer, the average latency is 1140 microseconds and 90% of the QoS requests (P90) are served within 1410 microseconds. For the gateway load balancer, the average latency is 1650 microseconds and 90% of the QoS requests are served within 2370 microseconds. Similar performance difference are observed in the P99 and P99.9

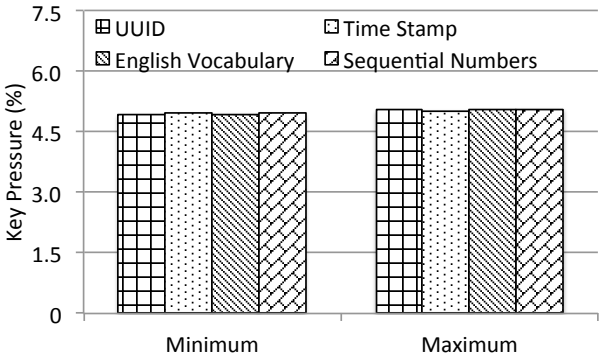


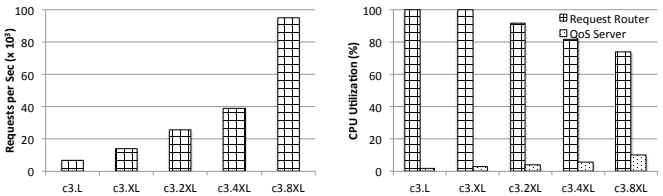

Fig. 6: Minimum and maximum key pressure for 500,000 QoS keys across 20 QoS servers behind the Request Router layer.

metric. On average, using the gateway load balancer adds approximately 500 microsecond to the round-trip latency, as compared to the DNS load balancer.

In this case of gateway load balancer, the QoS client establishes an HTTP connection to one of the load balancer nodes. The load balancer node puts the connection from the QoS client on hold, then establishes another connection to the request router for QoS check. When the load balancer receives a response from the request router, it passes the response back to the QoS client, then closes the connection with the request router. As such, the elevated level of latency shown in Figure 5 is caused by the additional TCP connection initiated by the load balancer node.

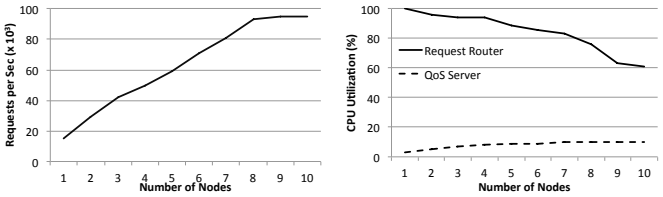
The problem with DNS load balancing is that by default most operating systems cache DNS resolution results until the time-to-live (TTL) property of the DNS record expires. In our evaluations we set the TTL property of the Janus DNS endpoint to 30 seconds. We observe that QoS requests from the same client node always hit the same request router node within the TTL cycle, resulting in imbalanced workload in the request router layer. This can become a serious problem when the number of request router nodes is more than the number of client nodes. If there are **M** request router nodes and **N** client nodes ($M > N$), during a TTL cycle there are only **N** request router nodes receive QoS requests, while the other request router nodes are idling. Such skewness in workload distribution significantly out-weights the 500 microsecond gain in round trip latency.

In the case of gateway load balancing, the above-mentioned problem does not exist because the gateway load balancer can be configured to route traffic using various algorithms. In our evaluations we use a round robin algorithm to distribute the QoS requests to the request router nodes behind the ELB. We observe a uniform distribution of workload across all request router nodes. Furthermore, using the ELB brings significant benefit in the management of Janus. In particular, the request router layer can be managed by an Auto Scaling group, where the capacity of the request router layer can be automatically adjusted based on a variety of metrics such as the average latency observed on the load balancer, the average CPU utilization on the request router nodes, etc. Therefore, we continue to use the gateway load balancing approach in our subsequent evaluations.




(a) Throughput (b) CPU Utilization

Fig. 7: Vertical scalability of the Request Router.




(a) Throughput (b) CPU Utilization

Fig. 8: Horizontal scalability of the Request Router.

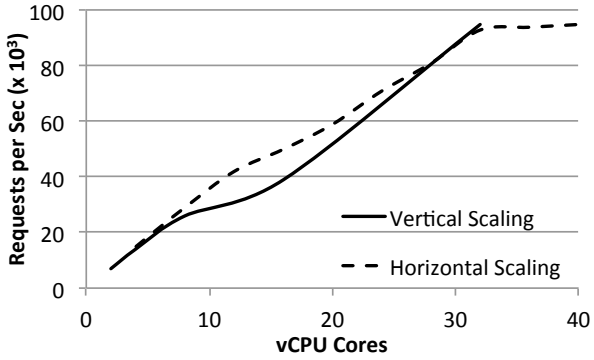



Fig. 9: Vertical vs horizontal scalability of the Request Router.

In our figures we do not present the CPU utilization for the database server because it is well below 1% through out our evaluation. Other resource consumptions such as disk I/O and network I/O are also very low. This is also true for all subsequent evaluations presented in this paper.

### *B. Request Router*

In a partitioned distributed system it is important that the workload is evenly distributed across all the partitions in the system. To evaluate the effectiveness of the above-mentioned request routing algorithm, we calculate the distribution of 500,000 QoS keys in a QoS system with 20 QoS servers behind the request router. In this evaluation we simulate four different types of QoS keys, including (a) randomly generated UUID's in "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx" format, (b) randomly generated date time string in "YYYY-MM-DD-HH-MM-SS" format, (c) unique words from the English vocabulary, and (d) sequential numbers starting from 1500000001 to 1500500000. Assuming that each QoS server receives equal workload from the request router then its key pressure should be 5% of the total workload. As shown in

Figure 6, among all four categories, the minimum key pressure is 4.933% and the maximum key pressure is 5.065%, while the standard deviations are smaller than 0.03%. Therefore, the request routing algorithm is capable of evenly distributing incoming QoS requests across all the QoS servers behind the request router.

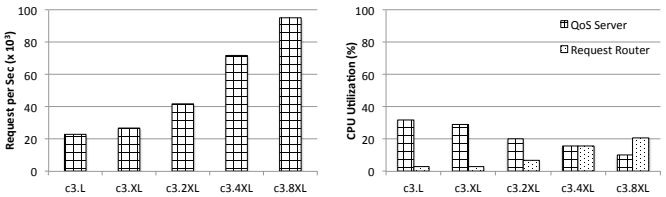
To evaluate the vertical scalability and horizontal scalability of the request router layer, we need to ensure a fixed processing capacity in the QoS server layer. This is achieved by provisioning a single c3.8xlarge node in the QoS server layer. To understand how vertical scaling affects the performance of the request router, we use a single node in the request router layer and change the instance type of the request router node. To understand how horizontal scaling affects the performance of the request router, we use a fixed instance type (c3.xlarge) for the request router node and change the number of nodes in the request router layer.

Figure 7 presents the results from the vertical scaling tests. As shown in Figure 7a, the processing capacity (throughput) of Janus increases when the size of the request router node becomes bigger. As shown in Figure 7b, when the request router node is small (c3.large and c3.xlarge), the CPU resource on the request router is depleted under heavy workload. When the request router node becomes bigger (c3.2xlarge and beyond) we observe some minor CPU underutilization. When the request router layer has sufficient processing capacity, the pressure is shifted to the QoS server layer, which is reflected in the increase in CPU utilization on the QoS server.

Figure 8 presents the results from the horizontal scaling tests. As shown in Figure 8a, the processing capacity (throughput) of Janus linearly increases when the number of the request router nodes increases. The processing capacity stops growing when there are more than 8 nodes in the request router layer, indicating the processing capacity of the QoS server might be the bottleneck. We notice that the maximum throughput in Figure 7a is very close to the maximum throughput in Figure 8a, which supports the speculation that the QoS server is the bottleneck. Figure 8b shows the CPU utilization on both the request router nodes and the QoS server node. As the number of nodes in the request router layer increases, the CPU utilization on each request router node decreases. Accordingly, the QoS server node now receives more traffic, hence the increase in its CPU utilization.

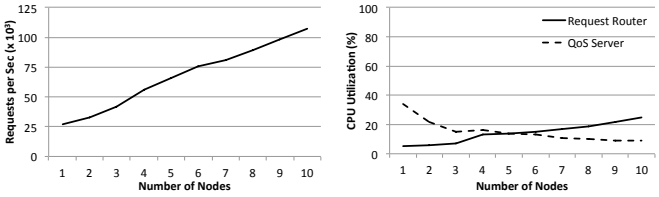
Figure 9 compares the performance of vertical scaling and horizontal scaling for the request router layer. With the same amount of vCPU cores in the request router layer, Janus achieves approximately the same throughput, regardless of the scaling technique being used.

During our testings we observe on the request router nodes a large number of TCP/IP connections in TIME_WAIT state. This is commonly considered as an issue that impacts the performance of client-server applications with very high concurrency. Various researchers [10], [11] suggested to reduce the value of *net.ipv4.tcp_fin_timeout* and enable *net.ipv4.tcp_tw_recycle* and *net.ipv4.tcp_tw_reuse*. We experiment with the suggested configurations in our tests. We do observed a reduction in the number of TCP/IP connections in TIME_WAIT state, but we do not observe any significant performance improvements.



(a) Throughput (b) CPU Utilization

Fig. 10: Vertical scalability of the QoS Server.



(a) Throughput (b) CPU Utilization

Fig. 11: Horizontal scalability of the QoS Server.

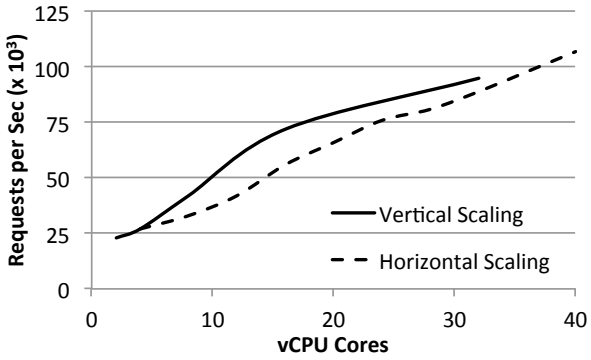


Fig. 12: Vertical vs Horizontal scalability of the QoS Server.

### C. QoS Server

This evaluation includes both the vertical scalability and horizontal scalability of the QoS server layer. To provide a fixed processing capacity in the request router layer, we deploy 5 c3.8xlarge nodes in the request router layer. To understand how vertical scaling affects the performance of the QoS server, we use a single node in the QoS server layer and change the instance type of the QoS server node. To understand how horizontal scaling affects the performance of the QoS server, we use a fixed instance type (c3.xlarge) for the QoS server node and change the number of nodes in the QoS server layer.

Figure 10 presents the results from the vertical scaling tests. As shown in Figure 10a, the processing capacity (throughput) of Janus increases when the size of the QoS server node becomes bigger. As shown in Figure 10b, there is significant CPU underutilization on both the request router layer and the QoS server layer. The CPU underutilization on the request router layer is expected, because we intentionally provision more processing capacity than needed. The CPU underutilization on the QoS server layer is largely due to the implementation of the locking mechanism being used to manage the QoS rules

in the local QoS table. This can be further optimized in our future work.

Figure 11 presents the results from the horizontal scaling tests. As shown in Figure 11a, the processing capacity (throughput) of Janus linearly increases when the number of the QoS server nodes increases. Figure 11b shows the CPU utilization on both the request router nodes and the QoS server nodes. As the number of nodes in the QoS server layer increases, the whole system is now capable of handling more traffic, hence the increase in the CPU utilization on request router nodes. At the same time, by distributing the workload across multiple QoS server nodes, each QoS server node now receives less traffic, hence the decrease in the CPU utilization on the QoS server nodes.

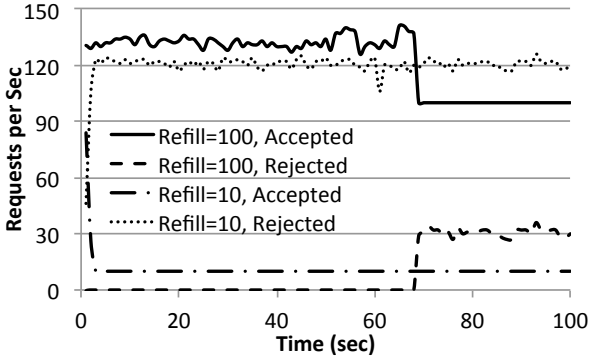
Figure 12 compares the performance of vertical scaling and horizontal scaling for the QoS server layer. With the same amount of vCPU cores in the QoS server layer, Janus achieves slightly higher throughput when vertical scaling is used. However, vertical scaling cannot scale indefinitely, because the largest instance type being used in this evaluation has only 32 vCPU cores. By adding more nodes to the QoS server layer, horizontal scaling can achieve higher throughput than vertically scaling to the biggest instance type.

### D. Application Integration

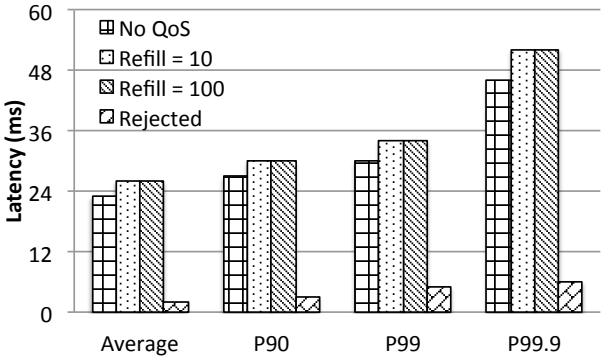
In the application integration evaluations, the photo sharing web application is deployed behind an ELB, with 5 c3.xlarge web server nodes in the back end. The web server is Apache version 2.4.18 and PHP version 7.0. The Memcached server is a dedicated r3.large node with Memcached version 1.5.4. The MySQL server is another dedicated r3.large node with MySQL server version 5.7. Janus is deployed behind another ELB, with 2 c3.xlarge nodes in the request router layer and 2 c3.xlarge nodes in the QoS server layer. The test client accesses the web application in a multi-thread fashion with an access rate of 130 requests per second, with an intentionally added noises.

In one of the tests, the test client runs on an EC2 instance with a known IP address. The custom QoS rule defines a refill rate of 100 requests per second, and a bucket capacity of 1000 requests. In the other test the test client runs on an EC2 instance without a known IP address. The default QoS rule defines a refill rate of 10 requests per second, and a bucket capacity of 100 requests. As shown in Figure 13a, when the custom QoS rule is used, the client is capable of achieving 130 requests per second for an extended period of time without being throttled. Because the consuming rate from the test client is greater than the refill rate, the leaking bucket is gradually depleted. After that the test client is only able to access the web application at 100 requests per second and the additional requests are throttled. When the default QoS rule is used, the consuming rate is far greater than the refill rate, and the leaking bucket is quickly depleted in a couple of seconds. After that the test client is only able to access the web application at 10 requests per second and the additional requests are throttled.

Figure 13b presents the round-trip latency metrics as measured from the test client. Before QoS integration, 90% of the requests are served in 27 milliseconds (P90). With QoS integration, 90% of the successful requests are served in 30 milliseconds, while the rejected requests are throttled




(a) Accepted and Rejected Requests




(b) Latency Statistics

Fig. 13: Application integration tests.

in 3 milliseconds. Similar trends are also observed in the P99 and P99.9 metrics. That is, QoS integration does not significantly impact the performance of successful requests, while the rejected requests are throttled in a timely manner.

## VI. Related Work

QoS itself is a mature subject in literature and has been applied to a variety of subjects in information and communications technology [12], [13]. The leaky bucket algorithm [8] was widely used in routing [14], [15], [16], [17], switching [18], [19], traffic shaping [20], and defense [21] as an integral part of the firmware running on the router, switch, firewall, or load balancer. In recent years, there have been a vast amount of research effort on applying QoS principles to applications and services such as gaming [22], multimedia [23], cloud computing [24], [25], [26], [27], [28], and service-oriented computing [29]. The majority of existing literature focuses on improving QoS performance on a single node.

Kim et. al. [4] propose a network QoS control framework for converged fabrics that automatically programs multiple router devices with the necessary QoS parameters, leveraging a set of QoS extensions of OpenFlow APIs. Chen et. al [3] propose a scalable QoS multicast routing protocol (SoMR) by carefully selecting the network sub-graph in which it searches for a path that can support the QoS requirements. The operations of SoMR are decentralized and rely only on the local state stored at each router.

Papazoglou et. al. [5], [6], [7] point out that QoS is an important research direction in service-oriented computing. The authors recognize that traditionally QoS quantifies the degree to which applications and systems support the availability of services at a required performance level. The authors further points out that service-oriented computing demands higher availability and introduces increased complexity. As such, QoS "encompasses important functional and non-functional attributes such as performance metrics, security attributes, transactional integrity, reliability, scalability, and availability".

The request router approach has been extensively used in developing scalable systems. For example, Anderson et. al. [30] use the request router technique to perform routing for a scalable network storage system. Gritter et. al. [31] use the request router technique to scale content delivery on the Internet. Lakshman et. al. [32] report that the request router technique is used in Cassandra, a decentralized structured storage system. Nishtala et. al. [33] report that Facebook uses the request router technique to scale its Memcache cluster. In general, such workload is considered read-intensive, and is quite different from the write-intensive QoS discussed in this paper.

In the area of parallel and distributed systems, researchers seem to favor TCP over UDP in communication due to the reliability offered by the TCP protocol. However, there have been some successful adoption of UDP in parallel and distributed systems. For example, He et. al [34] use UDP to achieve predictable high-performance bulk data transfer for data-intensive scientific applications.

## VII. Conclusion

In this paper, we present the design and implementation of Janus - a generic and scalable QoS framework for admission control purposes. In Janus, we introduce a multi-layer design, which includes a load balancer layer, a request router layer, a QoS server layer, and a database layer. The request router layer segregates incoming QoS requests into multiple independent partitions, where each partition is represented by a QoS server. The request routing algorithm is effective in evenly distributing QoS requests to the QoS server nodes behind the request router layer. The QoS server is a distributed set of leaky buckets with a refill mechanism. This provides accurate admission control according to the quota the user purchases, but still allows occasional burst operations when the user accumulates credit. In our evaluations we demonstrate that Janus achieves linear scalability both vertically and horizontally. For example, Janus achieves more than 100,000 requests per second with only 40 vCPU cores in the QoS server layer.

In Janus, we adopt a key-value request-response mechanism so that it can be made generically available to a variety of devices, applications, and systems. We use a photo sharing web application to demonstrate that Janus can be easily integrated with existing web applications.90% of the admission control decision were made in 3 milliseconds, which is only a small overhead as compared with the application's own latency. The horizontal scalability of various layers, the painless integration process, plus the very low latency overhead, prove that Janus can be used to provide QoS service for a wide range of SaaS applications.

## References


[1] L. Cherkasova and P. Phaal, "Session-based admission control: A mechanism for peak load management of commercial web sites," *IEEE Transactions on computers*, vol. 51, no. 6, pp. 669–685, 2002.

[2] M. Amirijoo, J. Hansson, and S. H. Son, "Specification and management of qos in real-time databases supporting imprecise computations," *IEEE Transactions on Computers*, vol. 55, no. 3, pp. 304–319, 2006.

[3] S. Chen and Y. Shavitt, "Somr: A scalable distributed qos multicast routing protocol," *Journal of Parallel and Distributed Computing*, vol. 68, no. 2, pp. 137–149, 2008.

[4] W. Kim, P. Sharma, J. Lee, S. Banerjee, J. Tourrilhes, S.-J. Lee, and P. Yalagandula, "Automated and scalable qos control for network convergence." *2010 Internet Network Management Workshop/ Workshop on Research on Enterprise Networking (INM/WREN)*, vol. 10, no. 1, pp. 1–1, 2010.

[5] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: State of the art and research challenges," *Computer*, vol. 40, no. 11, 2007.

[6] M. P. Papazoglou and W.-J. Heuvel, "Service oriented architectures: approaches, technologies and research issues," *The International Journal on Very Large Data Bases (VLDB)*, vol. 16, no. 3, pp. 389–415, 2007.

[7] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: a research roadmap," *International Journal of Cooperative Information Systems*, vol. 17, no. 02, pp. 223–255, 2008.

[8] E. A. Khalil, "A new proposal erica+ switch algorithm for traffic management," 1963.

[9] Y. Zhao, S. Li, S. Hu, H. Wang, S. Yao, H. Shao, and T. Abdelzaher, "An experimental evaluation of datacenter workloads on low-power embedded micro servers," *Proceedings of the VLDB Endowment*, vol. 9, no. 9, pp. 696–707, 2016.

[10] K. Kumazoe, M. Tsuru, and Y. Oie, "Performance of high-speed transport protocols coexisting on a long distance 10-gbps testbed network," in *Proceedings of the 1st international conference on Networks for grid applications*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007, p. 2.

[11] X. Shi, H. Jin, H. Jiang, X. Pan, D. Huang, and B. Yu, "Toward scalable web systems on multicore clusters: making use of virtual machines," *The Journal of Supercomputing*, pp. 1–19, 2012.

[12] P. Ferguson and G. Huston, *Quality of service: delivering QoS on the Internet and in corporate networks*. Wiley New York, 1998, vol. 1.

[13] C.-S. Chang, *Performance guarantees in communication networks*. Springer Science & Business Media, 2012.

[14] Q. Ma and P. Steenkiste, "Quality-of-service routing for traffic with performance guarantees," in *Building QoS into Distributed Systems*. Springer, 1997, pp. 115–126.

[15] S. Murthy and J. Garcia-Luna-Aceves, "Congestion-oriented shortest multipath routing," in *INFOCOM'96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE*, vol. 3. IEEE, 1996, pp. 1028–1036.

[16] R. Guérin, S. Kamat, V. Peris, and R. Rajan, "Scalable qos provision through buffer management," in *ACM SIGCOMM Computer Communication Review*, vol. 28, no. 4. ACM, 1998, pp. 29–40.

[17] F. Hao and E. W. Zegura, "On scalable qos routing: performance evaluation of topology aggregation," in *Proceedings of the 9th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, vol. 1. IEEE, 2000, pp. 147–156.

[18] L. Zhang, "Virtual clock: A new traffic control algorithm for packet switching networks," in *ACM SIGCOMM Computer Communication Review*, vol. 20, no. 4. ACM, 1990, pp. 19–29.

[19] K. Bala, I. Cidon, and K. Sohraby, "Congestion control for high speed packet switched networks," in *INFOCOM'90, Ninth Annual Joint Conference of the IEEE Computer and Communication Societies. The Multiple Facets of Integration. Proceedings, IEEE.* IEEE, 1990, pp. 520–526.

[20] G. Niestegge, "The leaky bucketpolicing method in the atm (asynchronous transfer mode) network," *International Journal of Communication Systems*, vol. 3, no. 2, pp. 187–197, 1990.

[21] D. K. Yau, J. Lui, F. Liang, and Y. Yam, "Defending against distributed denial-of-service attacks with max-min fair server-centric router throt-

tles," *IEEE/ACM Transactions on Networking (TON)*, vol. 13, no. 1, pp. 29–42, 2005.

[22] T. Henderson and S. Bhatti, "Networked games: a qos-sensitive application for qos-insensitive users?" in *Proceedings of the ACM SIGCOMM workshop on Revisiting IP QoS: What have we learned, why do we care?* Acm, 2003, pp. 141–147.

[23] M. Reisslein, K. W. Ross, and S. Rajagopal, "Guaranteeing statistical qos to regulated traffic: The multiple node case," in *Proceedings of the 37th IEEE Conference on Decision and Control*, vol. 1. IEEE, 1998, pp. 531–538.

[24] M. Xu, L. Cui, H. Wang, and Y. Bi, "A multiple qos constrained scheduling strategy of multiple workflows for cloud computing," in *Proceedings of 2009 IEEE International Symposium on Parallel and Distributed Processing with Applications*. IEEE, 2009, pp. 629–634.

[25] R. N. Calheiros, R. Ranjan, and R. Buyya, "Virtual machine provisioning based on analytical performance and qos in cloud computing environments," in *Proceedings of 2011 international conference on Parallel processing (ICPP)*. IEEE, 2011, pp. 295–304.

[26] Z. Zheng, X. Wu, Y. Zhang, M. R. Lyu, and J. Wang, "Qos ranking prediction for cloud services," *IEEE transactions on parallel and distributed systems*, vol. 24, no. 6, pp. 1213–1222, 2013.

[27] D. Bruneo, "A stochastic model to investigate data center performance and qos in iaas cloud computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 560–569, 2014.

[28] D. Serrano, S. Bouchenak, Y. Kouki, T. Ledoux, J. Lejeune, J. Sopena, L. Arantes, and P. Sens, "Towards qos-oriented sla guarantees for online cloud services," in *Proceedings of 2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2013, pp. 50–57.

[29] M. Alrifai, T. Risse, P. Dolog, and W. Nejdl, "A scalable approach for qos-based web service selection." in *Proceedings of the 6th International Conference on Service-Oriented Computing (ICSOC)*. Springer, 2008, pp. 190–199.

[30] D. C. Anderson, J. S. Chase, and A. M. Vahdat, "Interposed request routing for scalable network storage," in *Proceedings of the 4th Symposium on Operating System Design and Implementation (OSDI)*. USENIX Association, 2000.

[31] M. Gritter and D. R. Cheriton, "An architecture for content routing support in the internet." in *Proceedings of the 2nd Unix Symposium on Internet Technologies (USITS)*, 2001.

[32] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[33] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab *et al.*, "Scaling memcache at facebook." in *Proceedings of the 10th Symposium on Network System Design and Implementation (NSDI)*, 2013.

[34] E. He, J. Leigh, O. Yu, and T. A. DeFanti, "Reliable blast udp: Predictable high performance bulk data transfer," in *Proceedings of the 2002 IEEE International Conference on Cluster Computing*. IEEE, 2002, pp. 317–324.