Impact of Distributed Rate Limiting on Load Distribution in a Latency-sensitive Messaging Service*

Chong Li, Jiangnan Liu, Chenyang Lu, Roch Guérin, Christopher D. Gill
Washington University in St. Louis, Dept. Computer Science & Engineering
E-mail: {chong.li, liu433, lu, guerin, cdgill}@wustl.edu

***Abstract*—The cloud's flexibility and promise of seamless autoscaling notwithstanding, its ability to meet service level objectives (SLOs) typically calls for some form of control in resource usage. This seemingly traditional problem gives rise to new challenges in a cloud setting, and in particular a subtle yet significant trade-off involving load-distribution decisions (the distribution of workload across available cloud resources to optimize performance), and rate limiting (the capping of individual workloads to prevent global over-commitment). This paper investigates that trade-off through the design and implementation of a real-time messaging system motivated by Internet-of-Things (IoT) applications, and demonstrates a solution capable of realizing an effective compromise. The paper's contributions are in both explicating the source of this trade-off, and in demonstrating a possible solution.**

***Index Terms*—real-time, messaging service, distributed rate limiting, load distribution**

I. Introduction

A. Background

The cloud and its many "*aaS" instantiations [1] has ushered in a new era of access to computations that has enabled an explosion in distributed applications, in particular in the Internet-of-Things (IoT) space [2]. IoT applications commonly involve a large volume of data generated across geographically diverse sources (sensors) that need to be processed and often acted upon in a timely manner, *e.g.,* for actuation. The cloud's computational scalability and communication flexibility have made it an attractive platform for IoT applications [3]–[5]. This has spurred the development of communication platforms such as Microsoft Azure Service Bus and Amazon AWS IoT, which support a publish/subscribe (pub/sub) paradigm that lets a large number of senders and receivers connect without needing a complex mesh of one-to-one connections.

Fig. 1 illustrates a typical architecture, with topics as the abstraction connecting publishers (senders) and subscribers (receivers). Message brokers mediate between publishers and subscribers by receiving, queueing and forwarding messages for different topics. Publishers publish messages to a broker *for* a given topic, with subscribers subscribing to the topic to receive messages *from* all brokers responsible for that topic.

Scalability is realized by having multiple brokers across which to distribute the workload [6], [7], whether from different topics or individual topics with a heavy message load,

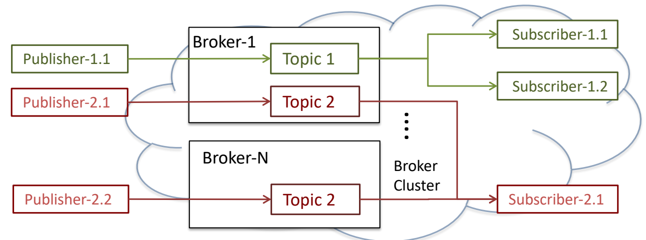


Fig. 1: Pub/sub messaging platform structure

e.g., Topic 2 in Fig. 1. As brokers are shared resources, their workloads must be controlled to ensure that service level objectives (SLOs) are met. This is particularly important for IoT applications that require timely data delivery. If an application/topic was to (accidentally or intentionally) misbehave and generate a much higher message load than anticipated, it could overwhelm the platform resources, and affect the SLOs of other topics. A standard approach to address this issue is to *rate limit* the message volume of each topic. Rate limiting is used in several public cloud platforms and is commonly implemented through a software API gateway [8], [9].

In practice, the rate limiting mechanism is in the form of a *token bucket* [8], [10]–[14] specified by two parameters, (r, b), as part of the topic's SLO; r is the rate at which tokens are generated and b the maximum number of tokens the topic can accumulate. Each token allows one message to be processed by the broker. In the absence of tokens, arriving messages are viewed as non-conformant and must wait for one.

As illustrated next, in practice the rate limiting function is often distributed across multiple message streams. The paper's contributions are in (i) explicating and quantifying the negative impact on latency of this splitting; and (ii) developing, implementing, and evaluating a system that mitigates it.

B. A Motivating Example

Consider an intelligent transportation system [15] (ITS) processing vehicle volume data from tens of thousands of sensors across an urban region, to control traffic signals within a second or less [16]. In an ITS, sensors are publishers of information that cloud servers subscribe to and process. The need for timely responses in turn calls for provisioning the messaging infrastructure to meet the system's SLO, typically in the form of a tail latency guarantee for message delivery.

* This work was supported by NSF grant CNS 1514254

As brokers are shared between the ITS and other topics, meeting SLOs calls for (rate) limiting message workloads. The volume of messages the ITS sensors generate is therefore first profiled, and the ITS rate limiter configured[1] to ensure conformance with the resulting traffic envelope [17].

Rate limiting is commonly implemented at a single gateway [8], [9], which has obvious scalability limitations. Furthermore, as rate limiting is typically in terms of application data units, *e.g.,* messages, a gateway introduces an additional application "hop" as it must reconstruct (from TCP or UDP packets) application data units to (rate) limit them. This extra application hop adds latency that is detrimental to latency-sensitive applications. These disadvantages have motivated distributed rate limiting (DRL) solutions [10], [11], [18]–[20].

A DRL system involves multiple rate limiters, *e.g.,* one sub-token bucket per topic and broker, which collaborate to ensure that the aggregate traffic still conforms to the original (r, b) workload envelope. Solutions differ in the level of "collaboration" between sub-token buckets. At one extreme, sub-token buckets share and update a common state to determine message conformance. This emulates centralized decisions while leaving enforcement distributed, but can incur a high communication overhead. At the other extreme, the original (r, b) token bucket is statically split into sub-token buckets (r_l, b_l), where $\sum r_l = r$ and $\sum b_l = b$ to preserve overall conformance with the original token bucket. This eliminates communication overhead, but at a loss in flexibility.

This latter option is our focus in spite of the increased access delay it can produce. We term this increase the *DRL penalty,* which arises even with perfectly matched sub-token buckets and message loads. Although the result is intuitive, the paper formally explicates the DRL penalty's origin, and leverages this understanding to design, implement, and evaluate a scalable real-time messaging (SRTM) platform that mitigates this penalty. SRTM is built on top of the NSQ open-source messaging middleware [21] and is publicly available [22].

II. Problem Statement & Goal

The basic question we seek to answer focuses on load distribution (LD) of messages across brokers, namely,

> **LD**: Given a new rate limited topic with envelope (r, b), how do we "best" distribute its publishers across brokers with known workloads?

where *best* relates to a common end-to-end tail latency SLO.

This is a question that has been extensively investigated in the context of call admission and load balancing. In our setting, the main challenge is that latency is affected by *both* the message processing latency and the access delay that rate limiting may introduce. Processing latency decreases as publishers are distributed over more brokers because of the lower per-broker processing load, but, as we shall see, rate limiting latency increases. Answering **LD,** therefore, involves a trade-off, and exploring it is the paper's primary motivation.

[1]In practice, this configuration often includes a "margin" to account for possible deviations from the original profile.

A. Why splitting a token bucket is bad

This section formally establishes that splitting a token bucket into multiple sub-token buckets negatively affects the rate limiting (access) delay, *i.e.,* what we termed the *DRL penalty*, even when publishers are perfectly distributed across sub-token buckets. As stated next, splitting a token bucket into multiple sub-token buckets always increases the cumulative rate limiting delay that messages experience, *irrespective* of how messages are distributed across sub-token buckets.

Proposition 1. *Given a two-parameter token bucket (r, b) and a general message arrival process where messages each require one token to exit the bucket, splitting this one-bucket system into multiple (say, k) sub-token buckets with parameters (r_l, b_l) such that $r = \sum_{l=1}^{k} r_l$ and $b = \sum_{l=1}^{k} b_l$, can never improve the running sum of the message delays, irrespective of how messages are distributed to the k sub-token buckets. More generally, denoting as $S(t)$ and $S^{(k)}(t)$ the sum of the delays accrued by all messages up to time t in the one-bucket and k-bucket systems, respectively, we have*

$$S^{(k)}(t) \geq S(t)\,,\ \forall t$$

The proof is in the appendix, but while general, the result provides little insight into its causes. To address this, we explore the special case of independent Poisson publishers for which the rate limiting delay can be explicitly expressed.

Let λ denote the aggregate (Poisson) message arrival rate of a topic, and (r, b) the parameters of its token bucket. Under Poisson arrivals, the system behaves like a modified M/D/1 queue [23], [24] with a job arrival rate of λ and a service time of $1/r$ (the time needed to generate one token), with messages delayed only when upon arrival the unfinished work U in the M/D/1 system exceeds $b - 1$[2]. The expected delay in an (r, b) token bucket is then easily found to be of the form:

$$E\left[T_{TB}^{(1)}\right] = \frac{1}{2r\left(1 - \frac{\lambda}{r}\right)} \cdot P_{\text{M/D/1}}(U > b - 1) \qquad (1)$$

where $P_{\text{M/D/1}}(U > b - 1)$ [25, Section 15.1] captures the odds that a message is delayed, while $\frac{1}{2r(1-\lambda/r)}$ is the expected delay of messages that have to wait for tokens.

The general form of Eq. (1) still holds for sub-token buckets after (randomly) splitting the original message load across them. Denoting their parameters as (r_l, b_l) and assuming, for simplicity, arrival rates λ_l such that $\lambda/r = \lambda_l/r_l, \forall\, l$, the two factors behind the DRL penalty are readily identified as:
(i) $P_{\text{M/D/1}}(U_l > b_l - 1) \geq P_{\text{M/D/1}}(U > b - 1)$: smaller buckets ($b_l \leq b$) imply that messages are more likely to wait;
(ii) $\frac{1}{2r_l} > \frac{1}{2r}$: with lower token rates ($r_l \leq r$), messages that have to wait (due to lack of tokens), wait longer.

Hence, Eq. (1) states that with Poisson arrivals, splitting the token bucket k-ways yields at least a k-fold increase in

[2]An empty queue maps to a full bucket, while a queue of b or more maps to an empty bucket. Note also that implicit in the model is the assumption that $b \geq 1$, *i.e.,* you need to be able to accumulate at least one (message) token to transmit messages.

average access delay (assuming $r_l = \frac{r}{k}, \forall l$), and likely more (because $P_{\text{M/D/1}}(U_l > b_l - 1) \geq P_{\text{M/D/1}}(U > b - 1)$).

The contribution of **(ii)** is unavoidable and not dependent on the assumption of Poisson arrivals. On the other hand, **(i)** is explicitly dependent on the assumption of Poisson arrivals. This hints at the possibility of mitigating this penalty by properly crafting the arrival process at each broker.

To better understand when and why this may be the case, consider a scenario where a token bucket (r, b) is split in two equal sub-token buckets $(r/2, b/2)$. Assume now that instead of independent Poisson publishers, publishers are synchronized, *i.e.,* generating messages at the same time to create an aggregate burst (batch Poisson). Splitting publishers equally across sub-token buckets then also splits the burst in the same proportion as the bucket size. All other parameters (*e.g.,* load) being the same, $P(U_2 > b/2 - 1) \approx P(U > b - 1)$, and the odds of messages being delayed are mostly unchanged[3].

In the next section, we translate this intuition into principles that mitigate the DRL penalty to best meet a topic's SLO.

III. SRTM Principles

A tongue-in-cheek restatement of the challenge faced by DRL decisions would be "to split, or not to split?" Given the finding of Proposition 1, a natural guideline is to *only split if you have to.* In other words, distribute a topic's publishers across the fewest brokers while ensuring that the resulting message processing loads do not result in SLO violations. Additionally, the above intuition points to another postulate, namely, *if you split the load, split the burst*, at least to the extent possible. Specifically, publishers whose message transmission times tend to be correlated, and therefore contribute to forming a burst, should be assigned to different brokers.

Our approach to distributing publishers of a new topic to message brokers builds on this insight. Section IV provides details on the resulting design, but we give next a brief overview of and motivation for those choices. Specifically, SRTM incorporates three principles:

1) *Concentration*: Identify the smallest number of brokers needed to meet the new topic's SLO;
2) *Max-min*: Maximize the minimum workload (and consequently token rate) assigned to any broker;
3) *Correlation-awareness*: Assign publishers to brokers to minimize inter-publisher correlation at each broker.

Concentration directly derives from Proposition 1 that identifies that the DRL penalty is unavoidable when splitting. Hence, it is natural to defer such a decision until absolutely necessary, *i.e.,* to reduce the message processing latency.
Max-min is similarly inspired by Eq. (1) and the linear increase in access delay as the token rate decreases.
Correlation-awareness seeks to select publishers to reduce message "bursts" in proportion to the decrease in bucket size.

[3]As mentioned earlier, the impact of lower token rates is still present.

IV. SRTM Design

A typical configuration of SRTM is shown in Fig. 2. Its core component, *Load Distributor*, realizes the principles put forth in Section III and assigns a topic's publishers to brokers based on the SLO and the brokers' current workload. It is supplemented by a *TB Adaptor* that periodically collects sub-token buckets statistics and adjusts (r_l, b_l) parameters in response to significant shifts in load. In this section, we focus on the design of the *Load Distributor*. Other components, including the *TB Adaptor*, are detailed in Section V.

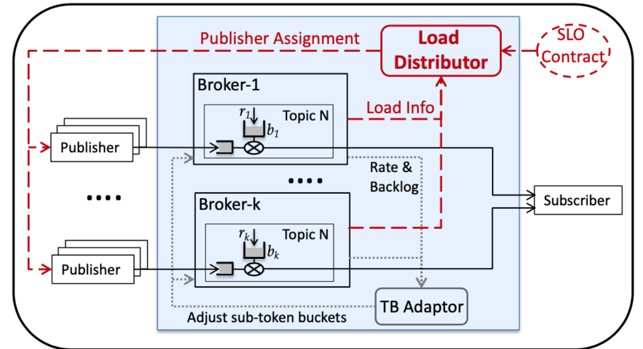


Fig. 2: SRTM Architecture Overview

A. Design Challenges

The Load Distributor's design rests on the three principles presented in Section III. However, realizing them calls for addressing two additional practical challenges.

Estimating Capacity. The *concentration* principle calls for determining the smallest number of brokers that can accommodate a new topic subject to the SLO (tail latency target). This is an "admission control" problem, where the system assigns publishers to a broker until the SLO is violated.

This calls for estimating a broker's spare capacity and ability to handle a new topic. This is challenging as message processing involves inter-dependent and concurrent tasks. For example, the NSQ open-source messaging middleware [21] uses the Go language (Golang) [26], which provides lightweight and scalable concurrency through Goroutines. This is well suited to IoT applications that involve many concurrent connections, and motivated our choice of NSQ as the basis for SRTM. However, modeling the behavior of the Goroutine runtime scheduler, including its reliance on a work-stealing strategy to exploit multicore systems, is non-trivial. Furthermore, depending on both the level of parallelism (number of publishers) of a topic and how publishers generate messages, performance bottlenecks may migrate across Goroutines. This makes a model impractical[4] and led us to rely instead on a measurement-based self-profiling approach (see Section IV-B).

Accounting for Correlation. The other design challenge is realizing the *correlation-awareness* principle, *i.e.,* identifying an assignment of publishers that results in the smallest possible

[4]The same challenges would arise in any modern high-concurrency system.

increases in $P(U_l > b_l - 1)$. This requires precise temporal characterization of the workload across publishers.

One option minimizes inter-arrival time variance at each broker. This is computationally complex and because variance does not fully capture temporal correlation, may not always be effective. Alternatives that rely on indices of dispersion [27]–[29] are equally if not more complex, as is directly measuring temporal correlation across publishers. Those challenges led us to rely instead on an empirical approach, namely, user-specified publisher correlation keys that reflect IoT application-level semantics (see Section IV-C for details).

B. Iterative Workload Distribution

When a new topic arrives, the Load Distributor employs an iterative process to distribute publishers across brokers. An iteration works as follows: (1) the Load Distributor estimates the minimum number of brokers, k, on which it is possible to *concentrate* the topic's workload; (2) publishers are then assigned to the k brokers in conformance with the *max-min* and *correlation-awareness* principles; (3) after publishers have been assigned to the k brokers, each broker runs a profiling phase (measuring latency) to determine whether it can accommodate its new workload without violating the SLO.

After Step (3), each broker knows if it needs to shed some publishers to remain in compliance with the SLO. If there are no SLO violations, the Load Distributor terminates. Otherwise, brokers whose SLO was violated determine how many publishers they need to shed, report this number to the Load Distributor, and mark themselves as "full." The Load Distributor then starts a new iteration to distribute released publishers, possibly involving additional brokers. The process ends when all publishers are assigned to brokers that can accommodate them, or the system runs out of brokers[5].

Next, we detail each step, except for the correlation-aware assignment of publishers that is covered in Section IV-C.

Step (1) starts by estimating each broker's residual message processing capacity ($rcap$), *i.e.,* the difference between its maximum message processing capacity[6] ($mcap$) and its current message load (the sum of its topic's message $rate$s). Brokers are then sorted in decreasing $rcap$ order to identify the **minimum** initial number k of brokers to which to assign the new topic's workload (the smallest number of brokers whose sum of $rcap$ values exceeds the topic's message *rate*).

Step (2) involves computing a workload $quota$ for each broker, which is set to the minimum of the broker's residual capacity $rcap$, and its fair share $rate/k$ of the workload. This maximizes the minimum assignment at each broker (the *max-min* principle). Once $quota$s are set, brokers' $rcap$ values are updated to reflect their new allocation, and publishers are assigned to brokers to realize those allocations in a *correlation-aware* manner, as per Section IV-C.

Step (3) acknowledges that $rcap$ estimates only account for message rates, and therefore ignore other factors that affect performance, *e.g.,* arrival burstiness, concurrency, interactions across workloads, etc. Step (3), therefore, relies on a measurement-based approach to evaluate the *actual* performance of each broker after Step (2).

Specifically, Step (3) includes an **online-fitting** profiling phase, whose goals are to (i) test whether each broker's SLO is still met after adding the new publishers (by measuring end-to-end message latency), and (ii) if it is not, determine how many publishers the broker needs to shed. This latter determination uses an iterative binary search[7] to identify how many publishers the broker can accommodate without violating its SLO. Excess publishers are returned to the Load Distributor, with the broker marked as full.

The next iteration parallels the first with updated variables. The variable ua_rate records the aggregate *unassigned* message rate from excess publishers, and k increases by the smallest number of additional brokers needed to accommodate ua_rate while conforming to the *max-min* and *correlation-awareness* principles. Iterations continue until all publishers are assigned to brokers, or the system runs out of brokers.

In spite of its accuracy, this iterative process has disadvantages. In particular, its measurement-based nature implies that convergence can take time. We quantify this in Section VI-E.

C. Correlation-aware Allocation

Our approach to capturing correlation across publishers is pragmatic, with users providing correlation information based on application semantics commonly available in IoT settings. This is akin stream processing partition keys in Azure IoT hub [30, p. 123], which let users identify correlated data streams for greater processing efficiency. Temporal correlation is common in many IoT deployments, *e.g.,* sensors in proximity to each other detecting similar physical phenomena, so that partition keys can be readily assigned. We adapt the concept of partition keys in SRTM, and provide APIs to let users label correlated publishers with the same **correlation group key**.

The Load Distributor then seeks to assign publishers with the same correlation key to different brokers, thereby splitting the message "bursts" they jointly generate. Specifically, given a set of correlation groups (as specified by the user) and a target rate ($quota$) to be assigned to a broker (based on $rcap$ and $rate/k$), the Load Distributor sets the group's rate contribution to $quota$ in proportion to its total message rate. We evaluate the efficacy of this approach in Section VI-D.

V. SRTM Implementation

SRTM is built on NSQ, a state-of-the-art messaging platform that we subsequently use as our baseline for comparison purposes. We first review NSQ's architecture before presenting SRTM and the implementation challenges we encountered, some broadly applicable to systems running on high concurrency platforms such as Go runtime.

[5] More brokers may be spawned by issuing a request to the cloud.

[6] This depends on the SLO and relies on a benchmarking phase.

[7] Performed in parallel by all the brokers with violated SLOs.

A. NSQ Architecture Overview

NSQ is comprised of a set of goroutines as shown in Fig. 3. Messages from publishers arrive on separate TCP connections each handled by an `IOLoop` goroutine. `IOLoop` goroutines then move messages into a dedicated buffer of the corresponding `Topic` goroutine. When a `Topic` goroutine is scheduled for execution, it runs to completion by pulling messages from its buffer until it becomes empty and forwarding those messages to `MsgPump` goroutines. The `MsgPump` goroutines send messages to subscribers through separate TCP connections. The Go runtime employs a work-stealing scheduler to schedule goroutines on multicore platforms. The combination of lightweight goroutines and a work-stealing scheduler contributes to NSQ's scalability and its ability to handle a large number of publishers on a single broker.

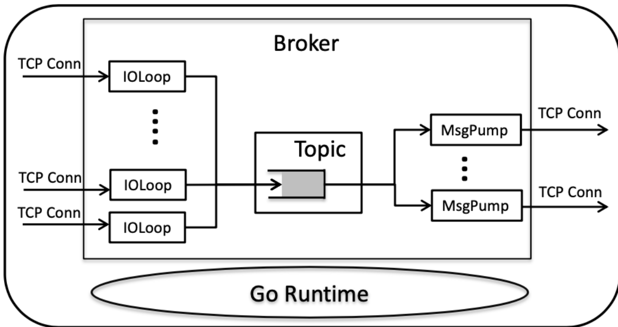


Fig. 3: Goroutines in an NSQ broker

B. Rate Limiting and Latency Optimization

SRTM's rate limiting token bucket is added to the `Topic` goroutine, which tracks the state of its token bucket at runtime. When a `Topic` goroutine is scheduled, it updates its token bucket state and pulls messages from its message buffer only if it has tokens. Otherwise, the goroutine returns and messages wait in the message buffer until tokens become available.

The token bucket logic is well understood, but implementing its precise temporal behavior in a high concurrency environment like Go runtime is non-trivial. A token bucket determines message conformance based on its state (number of tokens) *and* the message arrival time. The easiest approach involves timestamping messages when the `IOLoop` goroutine reads them from the TCP receive buffer. However, because of the cooperative scheduling of goroutines, there can be a significant lag between when messages are first received in the TCP buffer and when the `IOLoop` goroutine is scheduled. This can in turn artificially "batch" messages together and consequently lead the token bucket logic to mistakenly consider as non-conformant, and therefore delay, messages that were originally conformant. The resulting delays can become significant when the number of goroutines in the system is large.

SRTM employs TCP-layer timestamping to eliminate those errors. The arrival of each TCP packet (sk_buff) is recorded inside the Linux kernel (in the tcp_v4_rcv function). Whenever an `IOLoop` reads data, the TCP arrival timestamps are also copied (as out-of-band data) to user space. As user data (messages) often do not map 1:1 to TCP packets (fragments), each `IOLoop` maintains a table, in which every entry includes a *starting_sequence_number* (Seq_s), an *ending_sequence_number* (Seq_e) and a *TCP_arrival_timestamp* (T_{tcp}). Each entry indicates that the data between Seq_s and Seq_e (of the in-flight TCP bytestream) was received by the TCP layer at T_{tcp}[8]. When an `IOLoop` reconstructs a message from the bytestream, it determines the message's last entry based on the sequence number of the last byte of the message. The correct *TCP_arrival_timestamp* (T_{tcp}) of this message is then derived from that entry. As TCP arrival timestamps are independent of goroutine scheduling, SRTM is able to enforce rate limiting with high temporal accuracy.

C. Handling Garbage Collection

Garbage Collection (GC) in the Go runtime can also impact message processing tail latency. When GC is triggered, the Go runtime uses marker goroutines to mark memory allocations, a process that can consume up to 25% of CPU time [31]. As a result, we found significant increases in the 99th percentile of the message processing latency when GC is triggered.

As GC is usually triggered on demand in Go [32], we can minimize GC invocations by reducing dynamic memory allocation and hence slowing the growth of the heap size. We developed a GC-friendly version of NSQ by replacing instances of dynamic memory usage with statically allocated memory. Specifically, we created a pre-allocated ring buffer for each `IOLoop`, such that data read from the TCP socket is directly written into an existing slot in the ring buffer instead of having to request a dynamic memory allocation. In addition to on-demand GC, the Go runtime forces GC at least every 2 minutes (by default). We disabled this feature and in SRTM GC is only triggered on demand. These optimizations do not fully eliminate the impact of GC, but they mitigate it for the 99th percentile latency that is our target.

D. Adapting to Workload Changes

SRTM performs load distribution whenever a new topic arrives, but over time traffic may shift across brokers while remaining conformant to its global (r, b) traffic profile. The ensuing mismatch between traffic and sub-token buckets can result in significant DRL penalties. To avoid this, SRTM's static rate limiting configuration is augmented with a TB Adaptor that adjusts sub-token bucket parameters in response to "long-term" traffic shifts.

Specifically, brokers periodically report topics' rate and burst statistics to the TB Adaptor. Our implementation uses a per-topic $10s$ history window to track average message rate and maximum backlog in $1s$ increments. The TB Adaptor sets token rates in proportion to average message rates, and token bucket sizes in proportion to maximum backlogs. SRTM administrators can tune the overhead by adjusting the length of the history window and the frequency of updates.

[8]Entries are appended to the table when the `IOLoop` reads data and the associated TCP arrival timestamps from the OS kernel.

The TB Adaptor, however, only addresses the DRL penalties caused by shift in traffic among publishers. Such shifts can also result in traffic overloads at a broker. Resolving this calls for migrating publishers among brokers. This feature has been implemented (and is used in the workload distribution), but extending its use to load adaptation is left as future work.

VI. Evaluation

Towards evaluating SRTM, Section VI-A starts with a simple experiment to tease out the DRL penalty and the trade-off it introduces. Sections VI-B to VI-D follow with explorations of the relative benefits derived from SRTM's three principles. Section VI-E concludes by quantifying how long SRTM takes to converge after the arrival of a new topic.

The evaluation is carried out on a testbed with 7 physical hosts, each with two 8-core Intel Xeon E5-2630 processors, 8 GB of memory, and connected by 40 Gbps Ethernet. Two hosts are used to emulate publishers, another two are dedicated to subscribers, and 6 brokers are deployed on two more hosts. Each broker has 2 dedicated CPU cores, the default CPU configuration of Amazon-MQ instances [33]. The Load Distributor and TB Adaptor run on the remaining host.

The goals of our evaluation are to stress test SRTM along the different dimensions that typical IoT systems might exercise. This includes varying per topic workload, number of publishers, level of correlation, and in general message arrival patterns. Ideally, this would be realized using real-life traces from deployed IoT systems. Unfortunately, such traces are scarce and the few that exist are limited to specific environments, *e.g.,* campuses [34], [35]. As a result, we resort to synthetic workloads. Conversely, those have the advantage of allowing us to systematically stress-test SRTM along key dimensions of IoT traffic, *i.e.,* intensity, burstiness, and correlation.

We consider both *time-triggered* and *event-triggered* workloads. Our earlier ITS example is an instance of a time-triggered system with messages generated periodically from sensors across a city [15], [36]. Conversely, thermostats in smart buildings are instances of publishers that generate event-triggered messages, *e.g.,* when temperature crosses a threshold. We rely on periodic arrivals for time-triggered settings and Poisson arrivals for event-triggered. In both, we use homogeneous publishers[9] and vary the number of publishers to explore the impact of workload. Both periodic and Poisson publishers may generate messages in a *batch*, with the *batch size* contributing to the traffic burstiness. For example, a gateway controlling a group of sensors would generate a batch of messages if the sensors are controlled by a common timer or triggered by the same event. Finally, we group publishers in sets of different sizes to investigate the role of correlation.

In all experiments, our SLO is a 99th percentile end-to-end message latency of $1ms$[10]. When the SLO cannot be met, we report the best configuration. Additionally, a topic's aggregate token bucket was configured using a profiling phase that

[9]The SRTM system can handle heterogeneous publishers.

[10]This reflects the end-to-end (rate limiting, network, processing) latency in our testbed. Real-world deployments would have larger network delays.

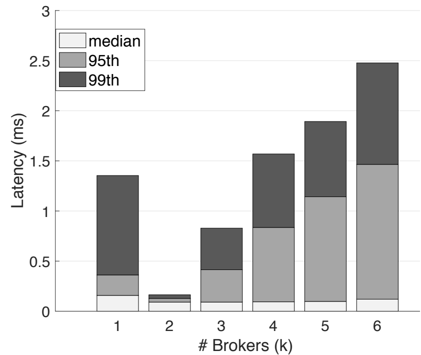


Fig. 4: DRL penalty for Poisson traffic
(6,000 publishers, 10 msg/s per publisher, batch size of 1).

gathered a representative traffic trace. The trace was used to perform an offline simulation of the token bucket performance using a token rate r set 10% higher than the topic's message rate, and searching for the smallest bucket size b that ensured a 99th percentile token bucket access delay of zero.

Since no existing messaging service performs load distribution in a way cognizant of the DRL penalty, we first compare our approach against a baseline using NSQ and standard **Load Balancing (LB)** that evenly distributes a topic's publishers across all available brokers. In other words, LB is blind to any publishers' property except for their message load, and assigns them to equalize load across brokers with sub-token buckets configured at each broker in exact proportion to the message load. Following this comparison, we compare SRTM to a series of baselines that increasingly incorporate its design principles, namely, *concentration*, *max-min*, and *correlation-awareness*. This isolates the contribution of each principle, while quantifying their cumulative impact in SRTM.

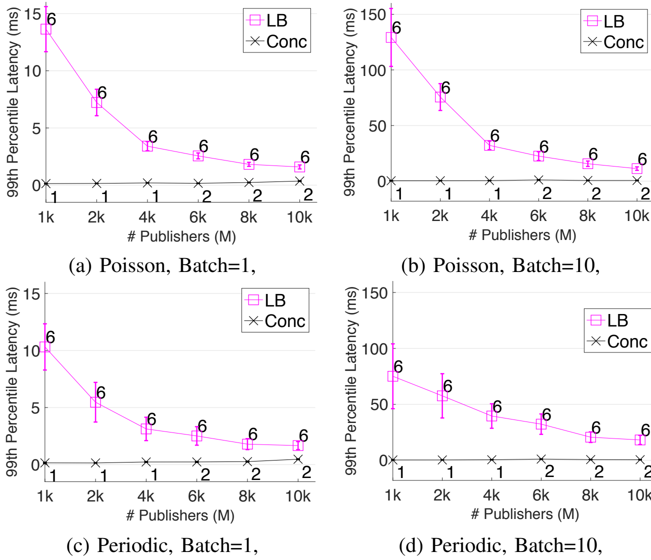
A. Illustrating the DRL Penalty

This first experiment involves a new topic with 6,000 publishers that each generate single messages (batch size of 1) according to a Poisson process with rate 10 msg/s. To illustrate the DRL penalty and the trade-off it introduces, publishers are distributed across an increasing number k of initially idle brokers ($k = 1$ to 6), with the topic's token bucket correspondingly split. Fig. 4 shows the median, 95th, and 99th percentile of the end-to-end latency for the 6 configurations.

The figure illustrates the trade-off between the lower processing delay from larger k (access to more processing capacity) and the increase in DRL penalty. When $k = 1$, the workload saturates a single broker, as illustrated by its large tail latency. Adding a second broker alleviates the problem while keeping the DRL penalty small, but further increases in k see a progressive degradation in performance as the gains from greater capacity fail to offset the growing DRL penalty.

B. The Benefits of Concentration

In this section, we compare a standard **Load Balancing (LB)** solution to an alternative, **Conc**, that only incorporates SRTM's *concentration* principle. Both LB and Conc, evenly



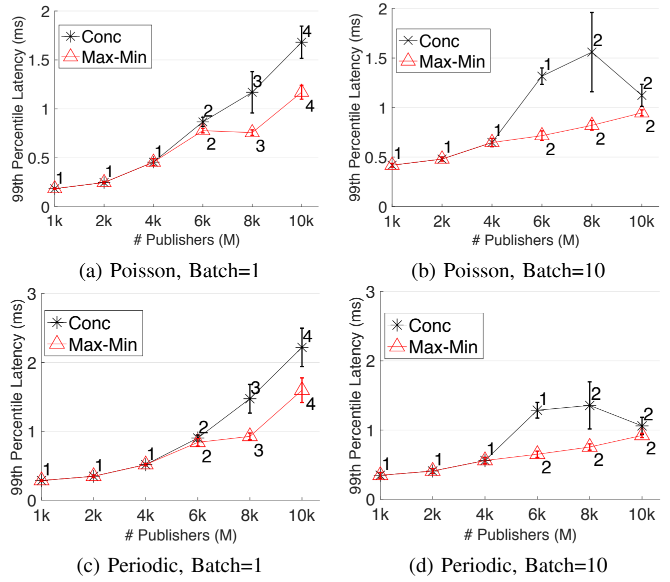
(a) Poisson, Batch=1,

(b) Poisson, Batch=10,

(c) Periodic, Batch=1,

(d) Periodic, Batch=10,

Fig. 5: Impact of Concentration on 99th percentile Latency. (10 msg/s per publisher)



(a) Poisson, Batch=1

(b) Poisson, Batch=10

(c) Periodic, Batch=1

(d) Periodic, Batch=10

Fig. 6: Impact of Max-Min on 99th percentile Latency. Initial (Poisson) load on brokers 1-6 (in kmsg/s): 10, 40, 50, 60, 70, 80 (w/ batch size of 10 messages).

distributes a topic's publishers across brokers. However, unlike LB which performs this distribution across *all* brokers, Conc limits itself to the *smallest* necessary number of brokers. As in SRTM, this number is first estimated based on the topic's message rate and the brokers' $rcap$ values, and then validated using a measurement-based approach.

The comparison is carried out for different workloads by varying the number of publishers associated with a topic. As before, publishers have a fixed message rate of 10 msg/s, and we now consider Poisson and periodic publishers. We also vary the burstiness of each publisher's message generation process by changing the size of the message batch it generates (1 or 10). Periodic publishers are independent of each other, with a randomly selected phase for their period. Experiments start again with idle brokers and are repeated 10 times. The results are shown in Fig. 5 with the mean and standard deviation of the 99th percentile latency reported for each configuration. The number of brokers across which the topic's workload is distributed is also shown next to each data point.

Figs. 5a and 5b (5c and 5d) report the results for Poisson (periodic) publishers and batch sizes of 1 and 10, respectively. Results are qualitatively consistent across scenarios, and illustrate the benefit of the *concentration* principle (Conc meets the topic's SLO for all configurations, while LB consistently fails to). The figures also highlight two intuitive factors.

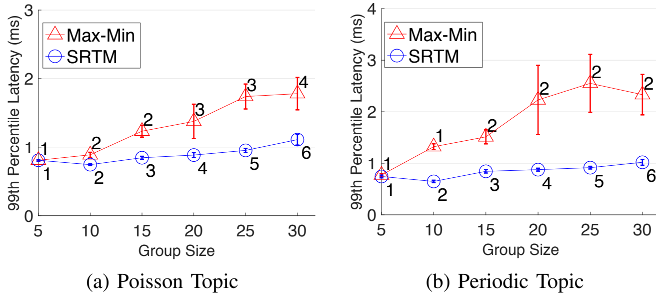
The first is the impact of the topic's workload on the DRL penalty: as the number of publishers, and therefore the topic's aggregate message rate, increases, the DRL penalty decreases. A higher token rate means a smaller token generation time and therefore, as indicated in Eq. (1), a shorter delay. Hence, while the DRL penalty is still present, a high message rate means that the token rate at each broker even after splitting the traffic remains high enough to ensure a comparatively small penalty relative to the message processing delay.

The other factor the figures bring to light is how traffic burstiness amplifies the DRL penalty. This again can be explained by looking at Eq. (1). When applied to a batch (Poisson) arrival process, the bucket size and the token rate are scaled down by the batch size, which both contribute to an increase in delay (the last message in a batch of size 10 that finds an empty token bucket must wait for 10 tokens).

C. The Benefits of Max-Min

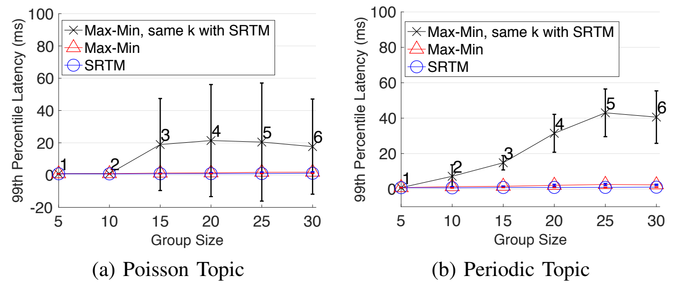
To evaluate the impact of the *max-min* principle, we reuse the scenarios of the previous section, but rely on configurations where the new topic finds brokers with varying pre-existing message loads (as shown in the caption of Fig. 6). We also introduce a new baseline algorithm, **Max-Min**, that extends Conc by incorporating the *max-min* principle. Like Conc, Max-Min targets accommodating the new topic with the smallest possible number of brokers, but instead of aiming to equalize the *total* load at each broker it seeks to maximize the *minimum topic* load assigned to a broker.

Figs. 6a and 6b (6c and 6d) compare Conc and Max-Min for a new topic with Poisson (periodic) publishers. The presence of existing workloads on the 6 brokers affects the new topic's distribution across brokers. When the topic's workload is low, it fits on the most lightly loaded broker and Conc and Max-Min perform identically. Their performance, however, starts deviating as the topic's load increases (beyond 4k publishers) and needs to use multiple brokers. For example, when the topic boasts 8k publishers, it needs to be distributed across brokers 1, 2, and 3. Conc equalizes the three broker's loads, while Max-Min instead seeks to maximize the topic's load on broker 3 that, because it has the heaviest existing load,



(a) Poisson Topic (b) Periodic Topic

Fig. 7: Impact of Correlation-aware Allocation.
(Each publisher has a batch size of 10 and an average message rate of 10 msg/s; Topic has 1,000 publishers).



(a) Poisson Topic (b) Periodic Topic

Fig. 8: Impact of Correlation-aware Allocation.
(Max-Min uses the same number of brokers as SRTM).

receives the smallest share. This results in a 99$^{\text{th}}$ percentile latency of $1.50ms$ for the publishers assigned to broker 3 under Conc, while it is $0.87ms$ under Max-Min, and the difference primarily arises from the larger DRL penalty under Conc (because of the smaller sub-token rate at broker 3).

As the two sets of figures show, Poisson and periodic publishers yield similar outcomes, but the experiments also offer interesting insight into the influence of SRTM's architecture on performance. In particular, Figs. 6b and 6d display tail latency for a new topic with publishers that generate bursts of 10 messages, for which both Conc and Max-Min perform as well if not better than when publishers generate bursts of size 1. This initially counter-intuitive behavior is because, in this scenario, performance is dominated by the processing of the per publisher `IOLoop` goroutines. Under a bursty arrival process, a publisher's `IOLoop` is scheduled less frequently, which results in a smaller number of simultaneously active `IOLoops` for the Go runtime scheduler to service. This smaller number of `IOLoops` allows the broker to handle a higher overall message load under both Max-Min and Conc[11].

Figs. 6b and 6d also reveal an improvement in performance of Conc when the new topic goes from 8k to 10k publishers. With 8k publishers, the residual number of publishers assigned to the second broker is smaller than with 10k publishers, which results in higher DRL penalty. This highlights the benefit of max-min, which avoids small residual assignments to the last broker that can produce a high DRL penalty.

D. The Benefits of Correlation Awareness

As discussed in Section IV-C, correlation in message generation across publishers is captured through correlation *groups*. We now compare Max-Min to **SRTM**, which incorporates correlation group information when distributing publishers across brokers. Max-Min is oblivious to that information, while SRTM seeks to leverage it to distribute publishers across brokers to reduce arrival burstiness in lockstep with the smaller bucket sizes. To isolate the impact of correlation, we again assume that a new topic arrives at a set of idle brokers.

[11]As described earlier, such complex interactions in the NSQ architecture are what necessitated the on online-fitting Step (3) of Section IV-B.

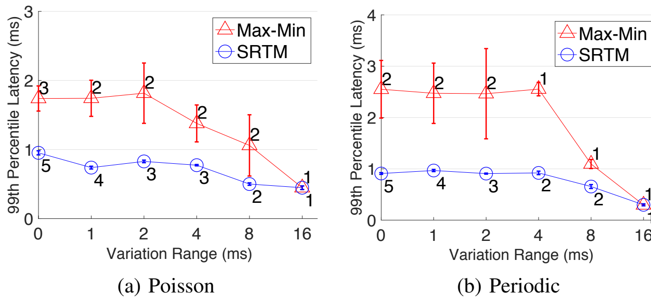
Because correlation groupings can be coarse, *e.g.,* reflecting physical proximity rather than precise synchronization, we consider two scenarios. The first assumes perfectly correlated publishers within the same group, *i.e.,* message generation times are synchronized, while the second relaxes this assumption by introducing variability (within some range) in the times at which publishers from the same group generate messages.

1) Scenario 1: Perfect correlation: In this set of experiments, publishers marked as belonging to the same group are synchronized in their message generation times. We vary group sizes across experiments, with larger groups corresponding to larger (synchronized) message bursts generated by each group. The main consequence of such an increase is that meeting the $1ms$ SLO for burstier traffic calls for distributing the topic across more brokers. This affects both Max-Min and SRTM, but the fact that SRTM relies on group information to split publishers from the same group across brokers enables it to mitigate the resulting increase in DRL penalty.

The results are shown in Figs. 7a and 7b for Poisson and periodic publishers, respectively. Both figures illustrate that SRTM is able to gain access to more broker capacity (and break message bursts) without incurring a significant increase in DRL penalty. In contrast, Max-Min is forced to use fewer brokers, as its "blind" assignment of publishers ultimately results in a DRL penalty that exceeds the benefits of distributing the topic's message load across more brokers. This is further illustrated in Fig. 8 that reports the performance of Max-Min when, for comparison purposes, it is forced to use the same number of brokers as SRTM. As anticipated, this makes its performance even worse.

2) Scenario 2: Imprecise correlation: This next set of experiments explores the extent to which the benefits of a correlation-aware distribution of publishers remain when correlation information is imprecise. Specifically, the message generation times of publishers within the same group are now evenly spread over an interval. As the size of the interval increases, correlation between publishers weakens.

Results are reported in Fig. 9 that compares SRTM and Max-Min. The experiments rely on the same type of publishers as in Scenario 1 with a group size of 25, and the interval across which messages from the same group were distributed varying from $0ms$ to $16ms$ ($0ms$ corresponds to the perfect synchronization configuration of Scenario 1). The results are



(a) Poisson (b) Periodic

Fig. 9: Latency evaluation - imprecise correlation.
(Each publisher has a batch size of 10 and an average message rate of 10/s; Topic has 1,000 publishers).

again consistent for both Poisson and periodic publishers and demonstrate that, at least when variations in arrival times are small (a few percents of the average message inter-arrival time), leveraging correlation information still helps mitigate the DRL penalty. The figure also illustrates another side-effect of increasing the spread of message arrival times from publishers in a group. Tail latency as well as the number of brokers across which publishers are distributed decrease for both SRTM and Max-Min. This is a direct consequence of the lower burstiness associated with this larger spread.

E. Load Distribution Latency

As described in Section IV-B, SRTM employs a measurement-based solution to "fine-tune" its capacity estimates and allocations of publishers to brokers. This fine-tuning is carried out in the online-fitting component of Step (3) of the load distribution mechanism. As this approach can take time to converge, we explore this *load distribution latency* in the same set of experiments reported in Section VI-D. Specifically, we benchmark the time it takes to distribute the publishers of a new topic arriving at a set of empty brokers. The topic's message load is $10k\ msg/s$ ($1,000$ publishers, each with a message rate of $10\ msg/s$ and a burst size of 10). We vary burstiness by changing the (synchronized) publishers' group size. When burstiness is low (small group size), a single broker can accommodate the new topic, but the number of brokers needed increases with burstiness (group size) so that 6 brokers are required when the group size reaches 30.

As the topic's workload is low and brokers are initially empty, the initial $rcap$-based assignment is always $k = 1$. Depending on the topic's burstiness, additional brokers may be needed (resulting in an exploration of progressively larger values of k). This is reflected in the rows of Table I. Each row corresponds to a different burstiness, and within a row, a column (k value) represents one iteration of the load distribution.

As stated, an iteration starts with an assignment of publishers to a single broker, and then seeks to assess if the SLO is met. If it is, the load distribution process successfully completes. If it is not, a binary search is initiated to determine the maximum number of publishers the broker can accommodate. The search stops as soon as latency is within

Group Size	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$	Time (sec)
5	1						64
10	2	1					189
15	5	3	1				558
20	5	4	2	1			745
25	3	4	4	2	1		871
30	3	5	3	2	3	1	1058

TABLE I: Load distribution latency of SRTM
k: iteration index; each k column gives the number of measurement rounds in that iteration; the last column reports the total load distribution time.

20% below the SLO target of $1ms$. Entries in the table give the number of search rounds. Each round lasts 60 secs to ensure a representative traffic sample, and this time is the primary contributor to latency, whose total value is reported in the last column. Once the number of publishers a broker can accommodate has been identified, excess publishers are then assigned to the next broker and k is increased by 1.

Table I indicates that as the burstiness (group size) of the topic increases, so does the duration of the load distribution phase. This is expected, since our estimate of $rcap$ is oblivious to workload burstiness, and therefore becomes less accurate as it increases. For bursty topics, initial assignments systematically over-estimate the number of publishers a broker can accommodate. This triggers repeated iterations, each calling for a binary search. In the "worst" case (group size of 30), the cumulative effect of those searches results in a load distribution phase that lasts in excess of 15 minutes. This is clearly long, though not unreasonable when dealing with, say, an IoT deployment that remains in place for weeks or months.

We note that improvements are possible for topics' workloads from well-understood traffic profiles, *e.g.,* from specific applications, for which customized models can be developed. This is, however, beyond the scope of this paper.

VII. Related Work

Distributed rate limiting has been realized in several cloud services through software platforms, *e.g.,* Cloud Bouncer [10], Tyk [18] and Doorman [37], which rely on coordination protocols to set rate limiting parameters across resources as a function of global contracts and workload partitions. Work by Raghavan et al. [11] describes a system of (distributed) rate limiters that drop packets probabilistically to emulate the effect of a single aggregate rate limiter on transport layer congestion. Retro [38] relies on a centralized controller and measurements together with distributed controllers to enforce objectives such as fairness or latency guarantees by adapting resource allocation across tenants. Stanojevic et al. [19], [20] develop analytical models that account for both load balancing and rate limiting in determining how to optimally control access to distributed resources as a function of demand.

Those works aim to emulate a centralized system in spite of their distributed nature, while SRTM acknowledges the impact of distribution (the DRL penalty) and attempts to mitigate it.

Another body of work targets SRTM's *concentration* principle to reduce operational or energy costs while meeting SLOs. WorkloadCompactor [14] explores how to best shape (through token buckets) co-located workloads with latency constraints to maximize how many a server can accommodate. STeP [39] has a similar cost reduction goal by co-locating tenants with compatible usage patterns. Other systems [40]–[43] realize such an outcome by co-locating latency-sensitive workloads with data-intensive (batch-processing) workloads. The aspect of mitigating the DRL penalty is absent from those works.

Leveraging workload *correlation* derived from traces to improve system performance has been the focus of earlier work. STeP [39] used FFT co-variance while Verma et al. [44] and Carpo [45] relied on Pearson correlation. In contrast SRTM relies on application semantics to identify correlation.

Finally, real-time messaging has been the target of a number of recent efforts in specialized environments, *e.g.,* air traffic control [46], robotics [47], or IoT [48] as in this paper. None of these works, however, consider the DRL challenge.

VIII. Conclusion

This paper's contributions are in identifying and explicating the DRL penalty that arises when rate limiting is distributed across servers, and in designing and developing SRTM, a messaging platform that mitigates this penalty. SRTM relies on three core principles: *concentration*, *max-min* and *correlation-awareness*, and was evaluated empirically on a local testbed. The evaluation demonstrated its ability to successfully mitigate the DRL penalty, while preserving the ability to scale by distributing workload across servers when needed. SRTM was developed on top of the NSQ open-source messaging platform and is publicly available for others to use [22].

Appendix – Proof of Proposition 1

Under a general message arrival process, a token bucket (r, b) where each message requires one token, behaves like a modified G/D/1 queue with messages experiencing delays only when the unfinished work in the system exceeds b. We rely on this property and the next lemma to establish Proposition 1.

Lemma 2. *At any point in time t, the unfinished work $U(t)$ in a work-conserving G/D/1 queue is smaller than or equal to the total unfinished work $U^{(k)}(t) = \sum_{l=1}^{k} U_l(t)$ in a set of k work-conserving G/D/1 queues with the same aggregate service rate and fed the same arrival process.*

The proof is omitted and relies on a single work-conserving G/D/1 queue clearing work at least as fast as k parallel work-conserving G/D/1 queues with the same aggregate service rate.

We are now ready to prove Proposition 1.

Proof. We first establish the result for the case $k = 2$, and wlog assume that $r = 1$. We establish that at any time t the number $N(t)$ of messages experiencing delays in the one-bucket system is less than or equal to the number $N_1(t) + N_2(t)$ of such messages in the two-bucket system, which together with the fact that in both systems messages waiting for tokens accrue delay at the same rate, yields the desired result. This is because, denoting as $S(t)$ and $S^{(2)}(t)$ the sums of the message delays incurred up to time t in the one and two-bucket systems, respectively, we have

$$S(t) = \int_0^t N(u)du \text{ and } S^{(2)}(t) = \int_0^t \left(N_1(u) + N_2(u)\right) du$$

The number $N(t)$ of messages waiting for tokens, *i.e.,* accruing delay, at time t in a one-bucket system with bucket size b is of the the form

$$N(t) = \lceil \max\{0, U(t) - b\} \rceil$$

where $\lceil x \rceil$ is the ceiling of x, and $U(t)$ is the unfinished work in the corresponding G/D/1 queue.

Similarly the total number of messages waiting for tokens in a two-bucket system with bucket sizes b_1 and b_2 such that $b = b_1 + b_2$ is of the form

$$N_1(t)+N_2(t) = \lceil \max\{0, U_1(t) - b_1\} \rceil + \lceil \max\{0, U_2(t) - b_2\} \rceil$$

Since we know that $\lceil x \rceil \leq \lceil x_1 \rceil + \lceil x_2 \rceil$, when $x \leq x_1 + x_2$, we focus on establishing that

$$\max\{0, U(t)-b\} \leq \max\{0, U_1(t)-b_1\}+\max\{0, U_2(t)-b_2\} \quad (2)$$

From Lemma 2, we know that $U(t) \leq U_1(t) + U_2(t)$. Next, we consider the cases $U(t) - b \leq 0$ and $U(t) - b > 0$.

Case 1: $U(t) - b \leq 0$

In this case, Eq. (2) is trivially verified.

Case 2: $U(t) - b > 0$

We further separate this case into two sub-cases:

Case 2a: $U_1(t) - b_1 \leq 0$ and $U_2(t) - b_2 \geq 0$ (or interchangeably $U_1(t) - b_1 \geq 0$ and $U_2(t) - b_2 \leq 0$)

In this case, Eq. (2) simplifies to

$$U(t) - b \leq U_2(t) - b_2$$

Applying again the result of Lemma 2, we have

$$\begin{aligned} U(t) \leq U_1(t) + U_2(t) &\Rightarrow U(t) - b \leq U_1(t) + U_2(t) - b_1 - b_2 \\ &\Rightarrow U(t) - b \leq U_2(t) - b_2 \,, \end{aligned}$$

where we have used $b = b_1 + b_2$ and $U_1(t) - b_1 \leq 0$. Hence, Eq. (2) again holds in Case 2a.

Case 2b: $U_1(t) - b_1 \geq 0$ and $U_2(t) - b_2 \geq 0$

In this case, Eq. (2) becomes

$$U(t) - b \leq U_1(t) - b_1 + U_2(t) - b_2 \,,$$

which again holds because of Lemma 2 and $b = b_1 + b_2$.

Since the case $U_1(t) - b_1 \leq 0$ and $U_2(t) - b_2 \leq 0$ is not possible under Case 2 (it would violate Lemma 2), this establishes that Eq. (2) holds in all cases. Accordingly, $N(t) \leq N_1(t) + N_2(t)\,, \forall t$, and as mentioned earlier, $S(t) \leq S^{(2)}(t)\,, \forall t$, which establishes the result for $k = 2$.

Applying the above approach recursively to groups of two sub-token buckets readily extends the result to $k > 2$. ☐

We note that Lemma 2 and Proposition 1 are independent of the order in which messages waiting for tokens are transmitted, as long as the schedule is "work-conserving," *i.e.,* messages leave as soon as one full token is available.

REFERENCES

[1] C. N. Hoefer and G. Karagiannis, "Taxonomy of cloud computing services," in *Proc. IEEE Globecom Workshops*, Miami, FL, December 2010.

[2] A. Botta, W. de Donato, V. Persico, and A. Pescapé, "Integration of cloud computing and Internet of Things: A survey," *Future Generation Computer Systems*, vol. 56, pp. 684–700, 2016.

[3] "AWS IoT Core," https://aws.amazon.com/iot-core/.

[4] "Azure IoT Hub," https://azure.microsoft.com/en-us/services/iot-hub/.

[5] K. Grünberg and W. Schenck, "A case study on benchmarking iot cloud services," in *Proc. International Conference on Cloud Computing*. San Francisco, CA: Springer, Jul. 2018, pp. 398–406.

[6] Microsoft, "Partitioned Queues and Topics," https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-partitioning.

[7] J. Rao, "How to choose the number of topics/partitions in a Kafka cluster," https://www.confluent.io/blog/.

[8] "Throttle API Requests for Better Throughput," https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-request-throttling.html.

[9] "Transform and Protect your API," https://docs.microsoft.com/en-us/azure/api-management/transform-api.

[10] "Cloud Bouncer: Distributed Rate Limiting at Yahoo," https://yahooeng.tumblr.com/post/111288877956/cloud-bouncer-distributed-rate-limiting-at-yahoo.

[11] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren, "Cloud Control with Distributed Rate Limiting," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 4, p. 337–348, Aug. 2007. [Online]. Available: https://doi.org/10.1145/1282427.1282419

[12] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft, "Queues Don't Matter When You Can JUMP Them!" in *Proc. USENIX NSDI*, Oakland, CA, May 2015, pp. 1–14. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/grosvenor

[13] K. Jang, J. Sherry, H. Ballani, and T. Moncaster, "Silo: Predictable message latency in the cloud," in *Proc. ACM SIGCOMM*, vol. 45, no. 4. London, U.K.: Association for Computing Machinery, Aug. 2015, p. 435–448. [Online]. Available: https://doi.org/10.1145/2829988.2787479

[14] T. Zhu, M. A. Kozuch, and M. Harchol-Balter, "WorkloadCompactor: Reducing Datacenter Cost While Providing Tail Latency SLO Guarantees," in *Proc. SoCC*, ser. SoCC '17, Santa Clara, CA, Oct. 2017, p. 598–610. [Online]. Available: https://doi.org/10.1145/3127479.3132245

[15] "SCATS," http://www.scats.com.au/.

[16] K. Lan, Z. Wang, M. Hassan, T. Moors, R. Berriman, L. Libman, M. Ott, B. Landfeldt, and Z. Zaidi, "Experiences in deploying a wireless mesh network testbed for traffic control," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 5, pp. 17–28, 2007.

[17] S. Mao and S. Panwar, "A survey of envelope processes and their application in quality of service provisioning," *IEEE Communications Surveys*, vol. 8, no. 3, 3rd Quarter 2006.

[18] "Tyk: Rate limiting," https://tyk.io/docs/control-limit-traffic/rate-limiting/.

[19] R. Stanojevic and R. Shorten, "Load Balancing vs. Distributed Rate Limiting: An Unifying Framework for Cloud Control," in *Proc. IEEE ICC*, Dresden, Germany, Jun. 2009, pp. 1–6.

[20] R. Stanojevic and R. Shorten, "Generalized distributed rate limiting," in *Proc. IEEE IWQoS*, Charleston, SC, Jul. 2009, pp. 1–9.

[21] "NSQ," http://nsq.io/.

[22] "SRTM: A scalable real-time messaging service," https://github.com/Chong-Li/SRTM.

[23] A. W. Berger, "Performance analysis of a rate-control throttle where tokens and jobs queue," *IEEE J. Select Areas Comm.*, vol. 9, no. 2, pp. 165–170, February 1991.

[24] A. W. Berger and W. Whitt, "The impact of a job buffer in a token-bank rate-control throttle," *Stochastic Models*, vol. 8, no. 4, pp. 685–717, 1992.

[25] J. Roberts, U. Mocci, and J. Virtamo, Eds., *Broadband Network Teletraffic – Final Report of Action COST 242*. Springer, 1996.

[26] "The Go Programing Language," https://golang.org/.

[27] K. W. Fendick and W. Whitt, "Measurements and approximations to describe the offered traffic and predict the average workload in a single-server queue," *Proceedings of the IEEE*, vol. 77, no. 1, pp. 171–194, January 1989.

[28] R. Gusella, "Characterizing the variability of arrival processes with indices of dispersion," International Computer Science Institute, Tech. Rep. TR-90-051, September 1990.

[29] W. Whitt and W. You, "The advantage of indices of dispersion in queueing approximations," *Operations Research Letters*, vol. 47, no. 2, pp. 99–104, March 2019.

[30] A. Basak, K. Venkataraman, R. Murphy, and M. Singh, *Stream Analytics with Microsoft Azure*. Packet Publishing, Ltd., 2017.

[31] W. Kennedy, "Garbage Collection In Go : Part I - Semantics," https://www.ardanlabs.com/blog/2018/12/garbage-collection-in-go-part1-semantics.html.

[32] "Golang Package Debug: SetGCPercent," https://golang.org/pkg/runtime/debug.

[33] "Amazon MQ," https://aws.amazon.com/amazon-mq/.

[34] A. Sivanathan, D. Sherratt, H. H. Gharakheili, A. Radford, C. Wijenayake, A. Vishwanath, and V. Sivaraman, "Characterizing and classifying IoT traffic in smart cities and campuses," in *Proc. INFOCOM Workshops*, 2017.

[35] A. Sivanathan, H. H. Gharakheili, F. Loi, A. Radford, C. Wijenayake, A. Vishwanath, and V. Sivaraman, "Classifying IoT devices in smart environments using network traffic characteristics," *IEEE Transactions on Mobile Computing*, vol. 18, no. 8, pp. 1745–1759, 2019.

[36] "CityPulse," http://www.ict-citypulse.eu/page/.

[37] "Doorman," https://github.com/youtube/doorman.

[38] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi, "Retro: Targeted resource management in multi-tenant distributed systems," in *Proc. USENIX NSDI*, ser. NSDI '15, Oakland, CA, 2015, pp. 589–603.

[39] R. Taft, W. Lang, J. Duggan, A. J. Elmore, M. Stonebraker, and D. DeWitt, "STeP: Scalable Tenant Placement for Managing Database-as-a-Service Deployments," in *Proc. ACM SoCC*, ser. SoCC '16, Santa Clara, CA, Oct. 2016, p. 388–400. [Online]. Available: https://doi.org/10.1145/2987550.2987575

[40] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-Scale Cluster Management at Google with Borg," in *Proc. Tenth European Conference on Computer Systems*, ser. EuroSys '15, New York, NY, USA, 2015. [Online]. Available: https://doi.org/10.1145/2741948.2741964

[41] A. Goder, A. Spiridonov, and Y. Wang, "Bistro: Scheduling Data-Parallel Jobs Against Live Production Systems," in *Proc. USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, Jul. 2015, pp. 459–471. [Online]. Available: https://www.usenix.org/conference/atc15/technical-session/presentation/goder

[42] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms," in *Proc. ACM SOSP*, ser. SOSP '17, Shanghai, China, 2017, p. 153–167. [Online]. Available: https://doi.org/10.1145/3132747.3132772

[43] S. A. Javadi, A. Suresh, M. Wajahat, and A. Gandhi, "Scavenger: A Black-Box Batch Workload Resource Manager for Improving Utilization in Cloud Environments," in *Proc. ACM SoCC*, ser. SoCC '19, Santa Cruz, CA, 2019, p. 272–285. [Online]. Available: https://doi.org/10.1145/3357223.3362734

[44] A. Verma, G. Dasgupta, T. K. Nayak, P. De, and R. Kothari, "Server Workload Analysis for Power Minimization Using Consolidation," in *Proc. USENIX Annual Technical Conference (ATC)*, ser. USENIX'09, San Diego, CA, 2009, p. 28.

[45] X. Wang, Y. Yao, X. Wang, K. Lu, and Q. Cao, "CARPO: Correlation-aware power optimization in data center networks," in *Proc. IEEE INFOCOM*, Orlando, FL, Mar. 2012, pp. 1125–1133.

[46] D. Xu, H. Liu, and J. Zhai, "Study and implementation of cross-platform real-time message middleware in ATC systems," in *Advances in Artificial Intelligence, Software and Systems Engineering*, T. Z. Ahram, W. Karwowski, and J. Kalra, Eds. Springer, 2021.

[47] D. Yu and H. S. Park, "Real-time middleware with periodic service for industrial robot," in *Proc. IEEE International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, Jeju, South Korea, Jun. 2017, pp. 879–881.

[48] A. A. Al-Roubaiey, T. R. Sheltami, A. S. H. Mahmoud, and K. Salah, "Reliable middleware for wireless sensor-actuator networks," *IEEE Access*, vol. 7, pp. 14 099–14 111, 2019.