Spring 5-15-2020

Real-Time Communication in Cloud Environments

Chong Li

Washington University in St. Louis

Washington University in St. Louis

School of Engineering and Applied Science

Department of Computer Science and Engineering

Dissertation Examination Committee:
Chenyang Lu, Chair
Sanjoy Baruah
Christopher D. Gill
Roch Guerin
Jing Li

Real-Time Communication in Cloud Environments

by

Chong Li

A dissertation presented to
The Graduate School
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

August 2020
Saint Louis, Missouri

Contents

List of Figures

List of Tables

Acknowledgments

My greatest thanks go to Dr. Chenyang Lu, Dr. Roch Guerin and Dr. Christopher D. Gill. They have been giving me invaluable advice throughout all the research projects in this dissertation. Without them, none of these projects would have been successful.

I would also like to thank the other committee members, Dr. Sanjoy Baruah and Dr. Jing Li, for their advice.

I would like to thank all the great members in Cyber-Physical Systems Lab. In our weekly lab meetings, they have been giving me insights from different perspectives. To all the friends I met in St.louis: the time we spent together would be my happiest memory.

Last but not least, I would like to thank my parents for their encouragement and support through my Ph.D journey.

Chong Li

Washington University in Saint Louis
August 2020

Dedicated to my parents, Yuhua and Wei.

ABSTRACT OF THE DISSERTATION

Real-Time Communication in Cloud Environments

by

Chong Li

Doctor of Philosophy in Computer Science

Washington University in St. Louis, August 2020

Research Advisor: Professor Chenyang Lu

Real-time communication is critical to emerging cloud applications from smart cities to industrial automation. The new class of latency-critical applications requires latency differentiation and performance isolation in a highly scalable fashion in a virtualized cloud environments. This dissertation aims to develop novel cloud architecture and services to support real-time communication at both the platform and infrastructure layers. At the platform layer, we build SRTM, a scalable and real-time messaging middleware (platform) that features (1) latency differentiation, (2) service isolation through rate limiting, and (3) scalability through load distribution among messaging brokers. A key contribution of SRTM lies in the exploitation of the complex interactions among rate limiting and load distribution. At the infrastructure layer, we develop VATC, a virtualization-aware traffic control framework in virtualized hosts. VATC provides a novel network I/O architecture that achieves differentiated packet processing with rate limiting while being scalable on multi-core CPUs. The research is evaluated in a cloud testbed in the context of Internet of Things applications.

Chapter 1

Introduction

Deploying distributed applications in cloud environments is becoming more common with the recent emergence of IoT cloud, Industry 4.0, and 5G networks. In these areas, the cloud-based applications typically have two main requirements:

- *Scalability*: they need to handle highly concurrent connections from distributed sensors, actuators, or cloud-edge devices. For example, intelligent transportations [21] control city-wise traffic signals based on vehicle volume data from thousands of widely-distributed sensors.

- *Latency*: they need to support diversity in the end-to-end latency guarantees they offer. For example, intelligent transportation applications require round-trip latency below 1 second [50], while industrial automation applications have much lower latency requirements of the order of 0.5-1 ms [56], and weather monitoring applications are essentially insensitive to latency.

Nowadays, satisfying the *scalability* requirements has been realized by leveraging the Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) of clouds. PaaS provides coordinative running platforms for distributed deployment. For large-scale cloud applications, messaging middleware (platform) [3, 4, 40] is one critical PaaS, as it supports scalable communication paradigms (e.g., pub/sub). IaaS is enabled by virtualization, which allows applications running in isolated virtual machines (VMs) so that infrastructure (including CPU, NIC, disk, etc) of every single powerful physical host can be divided and provided to multiple cloud customers.

On the other hand, satisfying the *latency* requirement is more challenging because resources of platforms and infrastructure are shared by applications with diverse Service Level Objectives (SLOs) [1]. In such shared environments, a common solution is latency (service) differentiation with applications mapped to service classes that match their latency SLOs. Service differentiation is a well-studied problem with many possible solutions. Because of its simplicity, a common approach in shared execution platforms or infrastructure is to rely on prioritization [39, 47, 52]. In particular, service providers can instantiate dedicated instances (e.g., messaging brokers, VMs) for each service class, and leverage priority-based process or VM schedulers to prioritize those instances.

The benefits of the simplicity of priority-based systems notwithstanding, they also require that access to the higher priority classes be controlled. If an application within a high-priority service class (accidentally or intentionally) misbehaves and generates much higher network traffic than anticipated, it could overwhelm the shared resources (CPU and NIC). This could in turn affect the service guarantees of other applications, especially those within low-priority service classes. A standard approach to mitigate this risk relies on *rate limiting* [2], which

[1]An application's SLO specifies the latency target to be met if the traffic is conformant to a rate limiting configuration.

[2]Rate limiting is also required by an application's SLO.

upper bounds the volume of traffic that an application is allowed to inject into the shared system.

By jointly supporting latency differentiation and rate limiting, real-time communication is generally achieved. However, in cloud environments with PaaS and IaaS, the special network architecture requires real-time communication enforced at two different layers, which is the main challenge this dissertation tackles.

1.1 Challenge: Real-Time Communication at Two Layers

Figure 1.1 conceptually shows the communication at PaaS and Iaas layers. At the *platform layer*, messages are delivered between applications through messaging middleware (platform). Applications, as message senders/receivers, are deployed either outside (as Internet-of-Things) or inside of cloud. Meanwhile, because intra-cloud applications run in VMs, the communication between them goes through the *infrastructure layer*. This communication is specifically handled by the network I/O of virtualized hosts.

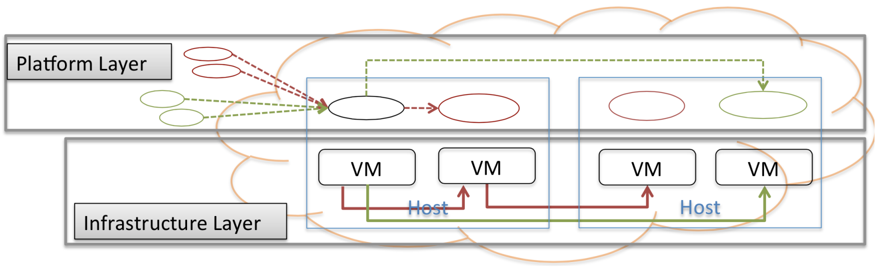


Figure 1.1: Communication at *platform layer* and *infrastructure layer*

This dissertation tackles the real-time communication at the two layers through two systems: (1) SRTM, a scalable and real-time messaging middleware at the platform layer; and (2) VATC, a virtualization-aware traffic control framework at the infrastructure layer. To support real-time communication, both systems implement latency differentiation and rate limiting.

1.1.1 SRTM

The scalability and latency requirements of IoT cloud applications require a scalable messaging platform that supports service differentiation. Therefore, we design and implement such a Scalable Real-Time Messaging (SRTM) middleware (platform).

In SRTM, messages of each service class are handled by dedicated (software) brokers, and we leverage priority-based process schedulers, e.g., SCHED_FIFO in Linux, to prioritize those brokers. Therefore, messages with different latency requirements are differentiated. As this part of work is straightforward, we focus on rate limiting and scalability (through load distribution across brokers) of SRTM in this dissertation. Between these two features, there are complex interactions that introduce negative impact on latency. Exploring this impact and proposing a solution with SLO enforcement is the main contribution of SRTM.

1.1.2 VATC

At the infrastructure layer (virtualized hosts), NIC and CPU resources are shared by network streams with different latency requirements. To enforce differentiated and isolated NIC sharing, existing traffic control relies on priority-based packet schedulers (e.g., Prio) and rate limiting of Linux queueing disciplines. However, as virtualization introduces additional components that require CPU resource, existing traffic control shows limitations which incur increased latency on real-time network traffic. To mitigate these limitations, we propose VATC, a virtualization-aware traffic control framework, which achieves differentiated and isolated CPU sharing among network streams. The key contribution of VATC is providing a novel underlying network I/O architecture in virtualized hosts.

1.2 Organization

The rest of this dissertation is organized as follows: Chapter 2 describes how SRTM tackles the latency problem caused by rate limiting and load distribution. Chapter 3 presents the latency differentiation, rate limiting and scalability of VATC. In Chapter 4, we discuss some open questions and conclude this dissertation.

Chapter 2

SRTM: A Real-time Messaging IoT Case Study

2.1 Background & Motivations

2.1.1 Background

The cloud and its many "*aaS" instantiations [44] has ushered in a new era of access to computations, and this has in turn enabled an explosion in distributed applications, in particular in the Internet-of-Things (IoT) space [33].

IoT applications commonly involve a large volume of data generated across many sources (sensors) distributed over geographically diverse locations, and that need to be processed and often acted upon in a timely manner, *e.g.,* for actuation purpose. Consequently, they require effective data transfer and processing solutions. The combination of the cloud's inherent computational flexibility and scalable communication platforms are what makes it an attractive platform for IoT applications [3, 4, 40]. This has led to the development of communication platforms such as Microsoft Azure Service Bus and Amazon AWS IoT. Those platforms are based on a publish/subscribe (pub/sub) paradigm, which lets a large number of senders and receivers connect without the need for a complex mesh of one-to-one connections.

Fig. 2.1 illustrates the typical architecture for such a system, with topics as the abstraction used to connect 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 brokers to receive messages *from* that topic. Scalability is realized by having multiple brokers across which to distribute the workload [18, 60], both from different topics as well as for individual topics with a heavy message load, *e.g.,* Topic 2 in Fig. 2.1. This enables rapid access to additional capacity when needed.

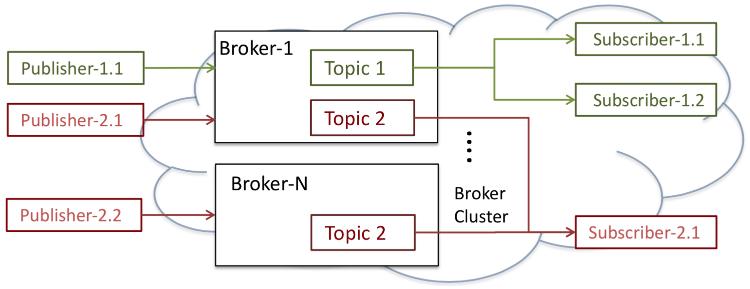


Figure 2.1: Pub/sub messaging platform structure

As with any shared resource, the workload of message brokers needs to be controlled to ensure that service level objectives (SLOs) are met. This is particularly important for IoT applications that require timely delivery (and processing) of their data. 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 (CPU and memory), and in turn 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 commonly implemented through a software API gateway [2, 23].

In practice, the rate limiting mechanism is in the form of a *token bucket* [7, 59, 39, 47, 2, 79], where each message requires a token before it can be processed by its broker. In the absence of tokens, an arriving message must wait for one (alternatively, it can be dropped). A topic's token bucket is typically part of its SLO and is specified by two parameters, (r, b). The parameter r gives the rate at which tokens are generated, and therefore bounds the long-term message rate the topic is entitled to. The parameter b indicates the maximum number of tokens the topic can accumulate, which in turn bounds the maximum message burst it can send to its broker without incurring an access delay (waiting for tokens). Their combination specifies the workload "envelope" [54] that the rate controller enforces.

2.1.2 A Motivating Example

Consider an intelligent transportation system [21] that has to process vehicle volume data acquired from thousands of sensors distributed over an entire urban region, and respond within a second or less [50] to ensure proper control of traffic signals. In this system, the sensors serve as publishers of information and cloud servers responsible for processing that information as subscribers to the messaging service[3]. The need for timely responses calls for provisioning the messaging infrastructure to meet the system's SLO, typically in the form of latency guarantee for message delivery (from publishers to subscribers).

[3]In closing the resulting control loop, actuation signals coming from the cloud servers, now serving as publishers, are sent back, with traffic lights the corresponding subscribers.

Because multiple topics share the same brokers, meeting SLOs calls for (rate) limiting the users' message workload. Hence, the volume of messages our intelligent transportation system sensors generate is first profiled, with this profile used to configure its rate limiter, *i.e.,* to ensure little to no access delay as long as it conforms to the corresponding traffic envelope[4].

As alluded to, the rate limiting functionality is commonly implemented at a single gateway. This is a centralized solution with obvious limitations when it comes to scalability. 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) those application data units to perform its functionality, *i.e.,* rate control the number of *messages* it lets in. This additional application hop results in unnecessary added latency that can be particularly detrimental to real-time applications. These disadvantages have been acknowledged before and have motivated the exploration of distributed rate limiting (DRL) solutions [7, 24, 59, 66, 65].

A DRL system involves multiple rate limiters, each associated with a different resource to which an application has been assigned, *e.g.,* brokers. Those rate limiters are then collaborating to ensure that between them, the aggregate traffic they let in conforms to the same overall (r, b) workload envelope as a centralized limiter. In the context of our intelligent transport system, different sets of publishers (sensors) are assigned to different message brokers (to distribute the message workload), with a separate rate limiting function at each broker that controls the volume of messages it has to process. In general, a DRL solution calls for splitting the original (r, b) token bucket into (r_l, b_l) sub-token buckets assigned to individual resources, and possibly dynamically adjusting those in response to workload fluctuations. As a matter of fact, most prior works have focused on this latter issue, albeit in

[4]In practice, the rate limiter is often configured with some "margin" to account for possible deviations from the original profile.

the context of applications that were not overly latency sensitive [59], or sought to compute (r_l, b_l) combinations that realized a specific trade-off between cost and performance [66, 65].

Our aim in this paper is different. Instead, our focus is on explicating that while distributing a rate limiter may be necessary, *e.g.,* to handle a heavy-load topic, it is inherently associated with an increase in access latency. In particular, as we establish, the simple fact of distributing a token bucket *increases the access delay it introduces.* With this knowledge in hand, we propose, implement, and evaluate a possible solution to mitigate this penalty while preserving the benefits of workload distribution. Specifically, Sections 2.4 to 2.6 report on the design, implementation, and evaluation of a scalable real-time messaging platform (SRTM) built on top of the NSQ open-source messaging middleware [17] and available for others to use (https://github.com/Chong-Li/SRTM.git).

2.2 Problem statement & Goal

The basic load distribution (LD) question we seek to answer is as follows:

> **LD**: Given a new topic with rate limiter (r, b) and a set of message brokers with existing workloads, how should we distribute publishers of the new topic across brokers to "best" meet the topic's SLO (target latency)?

where our definition of *best* is in terms of an efficient use of resources, *e.g.,* yielding a greater residual capacity for equal performance, or the ability to support a higher messaging workload.

This question is illustrated in Fig. 2.2 for a configuration that involves a new topic that can either assign all its publishers to a single broker with a resulting message processing utilization of ρ at the broker, or elect to split its publishers across two brokers, each then with a message processing utilization of ρ_1 and ρ_2, respectively, where $\rho_1, \rho_2 < \rho$. In the latter case, the original topic's token bucket (r, b) is split in two sub-token buckets (r_1, b_1) and (r_2, b_2), one at each broker, with parameters that verify $r = r_1 + r_2$ and $b = b_1 + b_2$, so as to preserve the same aggregate long-term message rate and message burst. Answering **LD** then calls for identifying the configuration that best meets the topic's performance (latency) goals.

This is a question that has been extensively investigated, even if some care needs to be exercised in cases of servers with uneven speeds, *e.g.,* see [63]. As a general rule, access to greater (message) processing capacity, *i.e.,* distributing the workload across more processors, yields better raw performance (because of the resulting lower load on individual processors), and this is embedded in most load-balancing strategies. The situation is different when latency is affected by *both* the message processing latency and the access delay that the rate limiting function may introduce. In particular and as we illustrate next, splitting the rate limiting function has a negative impact on its latency. We term this the *DRL penalty.* Therefore, the answer to **LD** now involves a trade-off between lowering message processing latency and increasing access (rate limiting) latency. Exploring this trade-off is a primary motivation for this paper.

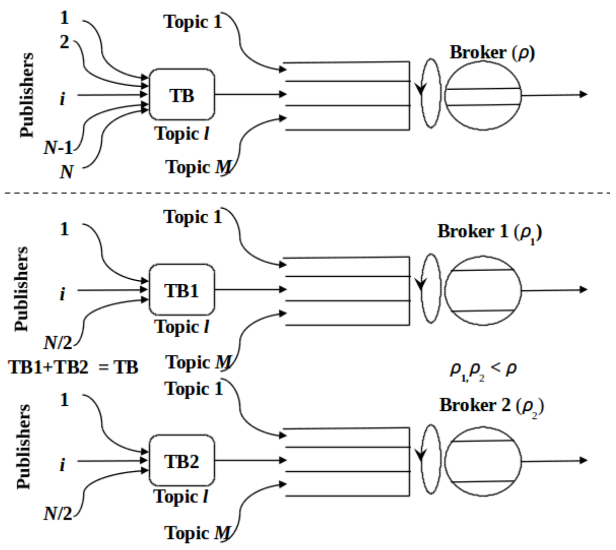


Figure 2.2: Alternative DRL configurations

2.2.1 Splitting a token bucket is bad

Given a two-parameter token bucket (r, b), then splitting this token bucket into multiple, say, k, sub-token buckets with parameters (r_l, b_l) requires $r = \sum_{l=1}^{k} r_l$ and $b = \sum_{l=1}^{k} b_l$. We next explore how splitting increases token bucket access delay with an example of Poisson message arrival pattern.

Let λ denote the aggregate (Poisson) message arrival rate of a topic, and (r, b) the parameters of its token bucket. Under the assumption of Poisson arrivals, the system behaves like a modified M/D/1 queue [31, 32] 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 $b-1$. The expected delay in an (r, b) token bucket is then of the form:

$$E[T_{TB}^{(1)}] \quad = \quad \frac{1}{2r} \cdot \frac{P_{\mathrm{M/D/1}}(U > b-1)}{\left(1 - \frac{\lambda}{r}\right)} \tag{2.1}$$

where $P_{\mathrm{M/D/1}}(U > b-1)$ can be computed as shown in [62, Section 15.1] and captures the odds that a message is delayed in the token bucket while $\frac{1}{2r(1-\lambda/r)}$ is the expected delay of messages that have to wait for tokens.

Under Poisson arrivals, Eq. (2.1) still holds after (randomly) splitting messages across, say, k brokers, with corresponding message arrival rates and sub-token bucket parameters that verify $\sum_{l=1,\ldots,k} \lambda_l = \lambda$, $\sum_{l=1,\ldots,k} r_l = r$, and $\sum_{l=1,\ldots,k} b_l = b$, where for ease of exposition, we assume that $\lambda/r = \lambda_l/r_l, \forall\, l$, *i.e.,* message and token rates are perfectly matched,

From Eq. (2.1), we can readily identify the two factors that contribute to the DRL penalty. Specifically,

(i) $P_{\mathrm{M/D/1}}(U_l > b_l - 1) \geq P_{\mathrm{M/D/1}}(U > b - 1)$: smaller buckets ($b_l \leq b$) imply that messages are more likely to have 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. (2.1) states that under the assumption of Poisson arrivals, splitting the token bucket k-ways yields at least a k-fold increase in access delay (*e.g.,* assuming $r_l = \frac{r}{k}, \forall\, l$), and likely more (because
$P_{\mathrm{M/D/1}}(U_l > b_l - 1) \geq P_{\mathrm{M/D/1}}(U > b - 1)$).

We note that the delay increase from the slower token rates r_l is unavoidable and not dependent on the assumption of Poisson arrivals. On the other hand, the fact that $P_{\mathrm{M/D/1}}(U_l > b_l - 1) \geq P_{\mathrm{M/D/1}}(U > b - 1)$ is explicitly dependent on the assumption of a Poisson process[5]. This hints at the possibility that for different arrival processes this penalty may not always arise, or may be mitigated by properly crafting the arrival process at each broker.

[5]Randomly splitting a Poisson process still yields a Poisson process.

To better understand when and why this may be the case, consider the arguably extreme but illustrative example of M synchronized publishers, *i.e.,* all generating messages at the same time to create an aggregate burst of M messages. In such an extreme scenario, as publishers are split across brokers so is the burst, with the burst size at each broker decreasing in the same proportion as the bucket size. All other parameters, *e.g.,* load, being the same, this ensures that $P(U_l > b_l - 1) = P(U > b - 1)$. In other words, the DRL penalty is now gone.

In the next section, we discuss how we translate this intuition into a set of principles aimed at mitigating the DRL penalty while meeting topics' SLOs.

2.3 SRTM Goals and Principles

As reflected in **LD**, SRTM seeks to offer a messaging service (for IoT applications) that is both efficient in its use of cloud resources and capable of enforcing latency guarantees (SLOs). A common SLO is in the form of a tail latency guarantee, *e.g.,* a 99th percentile latency below 1 ms. This calls for both controlling the messaging workload that originates from users (through rate limiting) and for determining how to best distribute that workload across message processing resources (brokers). As discussed in the previous section, the difficulty lies in the opposing effects of load distribution on message processing and rate limiting latency, respectively.

A tongue-in-cheek restatement of the challenge faced by DRL decisions would be "to split, or not to split?" Given the finding above, 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, our 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 2.4 provides details on the resulting design, but we give next a brief overview 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, and consequently reduce the burstiness of the message arrival process at each broker as much as possible.

Concentration seeks to avoid or minimize DRL penalty whenever feasible. In particular, as highlighted by Eq. (2.1), the DRL penalty can grow linearly and often super-linearly with the number of sub-token buckets across which the workload is split. Hence, it is natural to avoid splitting a topic as long as the broker's load does not yield a processing latency that violates the topic's SLO.

Max-min is similarly inspired by Eq. (2.1) and the fact that, irrespective of the arrival process, the access delay of messages that experience a delay is inversely proportional to

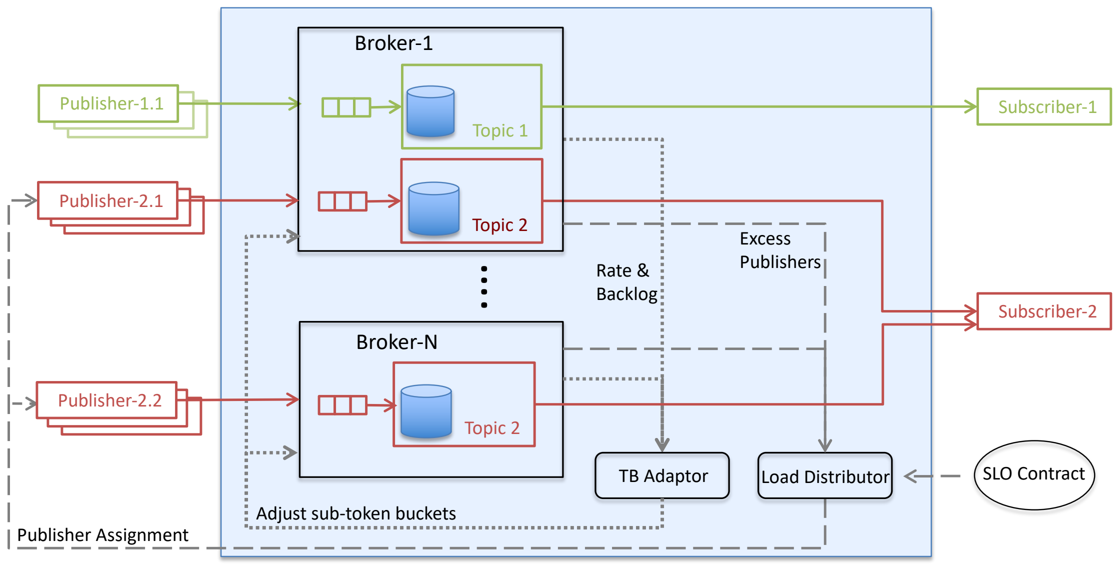


Figure 2.3: SRTM Architecture Overview. The example illustrated has two topics, with Topic 2 distributed across multiple brokers.)

the token rate. Hence, keeping the minimum token rate across sub-token buckets as high as possible is desirable for achieving tail latency guarantees.

Correlation-awareness seeks to select publishers so as to decrease the “burst” of the arrival process at each sub-token bucket in a manner that parallels the decrease in the size of their respective bucket size. We note that decreasing arrival bursts should benefit both the DRL penalty and the message processing delay in the brokers.

2.4 SRTM Design

This section presents the design of SRTM, with Fig. 2.3 offering a high-level overview of a typical configuration. It consists of individual brokers (numbered Broker-1 to Broker-N), with publishers of a given topic distributed across one or more brokers. The core component of SRTM is a *Load Distributor* which is responsible for distributing a new topic's publishers to brokers based on the topic's SLO and the brokers' existing load (as Topic 2 in Fig. 2.3 shows). In addition, a *TB Adaptor* tracks the status of sub-token buckets at run time and triggers adjustments of their (r_l, b_l) parameters in response to message traffic changes. In this section, we focus on the design of the SRTM *Load Distributor,* as it is the primary component responsible for realizing the principles put forth in Section 2.3. Other components, including the *TB Adaptor*, are detailed in Section 2.5.

2.4.1 Design Challenges

The Load Distributor is designed based on the three principles presented in Section 2.3. However, realizing them calls for addressing two practical challenges.

Estimating Capacity. In contrast to traditional load balancers SRTM does not seek to evenly distribute load across available resources. Instead, the *concentration* principle calls for determining the smallest number of brokers that can accommodate a new topic subject to its SLO (tail latency target). This, and to a lesser extent the *max-min* principle, is essentially an "admission control" problem, where the system keeps assigning publishers to a broker unless the topic's SLO is violated[6]. Such admission control decisions require an accurate estimation of the number of publishers a broker can handle subject to the SLO. In practice, estimating the capacity of brokers is challenging as message processing involves a set of inter-dependent and concurrent tasks.

[6]We assume a common SLO across, say, "real-time" topics.

For example, the NSQ open-source messaging middleware [17] is implemented using the Go language (Golang) [22], which provides lightweight and scalable concurrency through Goroutines. This is well suited to IoT applications that involve large numbers of concurrent connections, and motivated our choice of NSQ as the basis for implementing SRTM. However, modeling the behavior of the underlying 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 migrate across NSQ components. This makes a model-based approach mostly impractical and leads us to instead rely on a measurement-based approach [35]. Specifically, we use iterative measurements to discover how to best distribute a topic's publishers across brokers while meeting its latency target (see Section 2.4.2).

Accounting for Correlation. The other design challenge arises in realizing the *correlation-awareness* principle. Specifically, based on Eq. (2.1), given that a workload is to be split across a number of brokers, SRTM's goal is to identify an assignment of publishers that results in the smallest possible increases in $P(U_l > b_l - 1)$ across brokers. This means crafting an arrival process at each broker that lowers the odds that the unfinished work exceeds the bucket size (more precisely, exceeds $b_l - 1$). This is challenging as it requires precise temporal characterization of the workload generated by individual publishers.

A reasonable option is to create an arrival process at each broker with the smallest possible inter-arrival time variance. However, crafting such arrival processes from individual publisher arrival processes is computationally complex. Furthermore, variance only reflects "stationary statistics" of the arrival process at each broker, and so does not fully capture temporal correlation. Consequently, minimizing the variance of the arrival process at each broker may not always realize our goal of minimizing increases in $P(U_l > b_l - 1)$. Alternatives that directly measure temporal correlation are even more complex.

Those challenges lead us to instead rely on an altogether different alternative, namely, user-specified publisher correlation keys that reflect IoT application-level semantics (see Section 2.4.3 for details).

2.4.2 Iterative Workload Distribution

When a new topic arrives, the Load Distributor is responsible for distributing its publishers among brokers to meet the topic's SLO. As discussed in Section 2.4.1, it is challenging to accurately estimate the number of publishers a broker can accommodate subject to an SLO. To overcome this challenge, the Load Distributor employs an iterative process that distributes publishers across brokers during a measurement-based profiling phase.

Each iteration of the load distribution process works as follows: (1) the Load Distributor first estimates the *minimum* number of brokers (k) with enough available capacity to accommodate the new topic; (2) the topic's 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 an independent profiling phase (measuring latency) to validate whether it can accommodate its new workload without violating its SLO.

After Step (3), each broker knows if it can handle its new workload or needs to shed some publishers to meet its SLO. If there are no SLO violations, the load distribution process is deemed successful and ends. Otherwise, brokers whose SLO was violated determine how many publishers they need to shed, report this number to the Load Distributor, and marks 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. When that happens, more brokers may be spawn by issuing a request to the cloud.

Next, we describe each step in more details, with the exception of the correlation-aware assignment of publishers, which is the subject of its own sub-section, Section 2.4.3. For simplicity, we first describe the process used in the first iteration that takes place when a new topic arrives. We then extend the discussion by describing how the variables used in each iteration are updated, with Appendix A providing the detailed pseudo code.

Step (1) reflects our *concentration* goal and seeks to determine the minimum number of brokers needed to accommodate a new topic with a given aggregate message *rate*. The residual message processing capacity (*rcap*) of each broker is first estimated based on the difference between its maximum message processing capacity[7] (*mcap*), and its currently allocated message rate (the sum of the message *rate*s from topics already assigned to the broker). Brokers are then sorted in decreasing order of *rcap* value to identify the **minimum** initial number k of brokers to which to assign the new topic's workload (the smallest number of brokers such that the sum of their *rcap* values exceeds the topic's message *rate*).

Step (2) is concerned with determining how to best distribute the message workload of the new topic across those k brokers according to the *max-min* and *correlation-awareness* principles. This involves computing for each broker a workload *quota* it should be assigned. This *quota* is set to the minimum of the broker's residual capacity *rcap*, and its fair share

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

rate/*k* of the workload. This maximizes the minimum assignment at each broker (the *max-min* principle), unless limited by residual capacity. Once *quotas* 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, which, as mentioned earlier, is described in Section 2.4.3.

Step (3) acknowledges that estimates for *rcap* only account for message rates, and therefore ignore many factors that affect performance, *e.g.,* arrival burstiness, concurrency (among connections from different publishers), 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) consists of a profiling phase, which we call **online-fitting**, 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 to return to compliance with the SLO. Under (ii), an iterative binary search[8] is triggered to estimate the maximum subset of additional publishers that the broker can accommodate without violating its SLO (see Appendix A for details). Excess publishers are returned to the Load Distributor, while the broker is marked as full.

This next iteration parallels the first one with updated variables. A separate variable *ua_rate* records the aggregate unassigned message rate from excess publishers, while *k* is increased to account for the smallest number of additional brokers needed to accommodate *ua_rate* in a manner that again conforms to the *max-min* and *correlation-awareness* principles (see Appendix A for details). Iterations continue until all publishers are assigned to brokers that can accommodate them, or the system runs out of brokers.

[8]This is performed in parallel by all the brokers with violated SLOs.

In spite of its efficacy, we acknowledge that this iterative process has disadvantages. First, its measurement-based nature implies that convergence can take time. We quantify this overhead in Section 2.6.5. Second, applications may experience temporary service degradation while we iterate. This is unavoidable in any measurement-based approach and, as discussed in Section 2.4.1, is a consequence of the difficulties in constructing a sufficiently general and accurate model.

2.4.3 Correlation-aware Allocation

As pointed out in Section 2.4.1, measuring correlation across a topic's publishers is challenging. As a result, we adopt a pragmatic approach where users provide correlation information based on application semantics commonly available in IoT settings. Our approach is inspired by the concept of partition keys used in stream processing in Azure IoT hub [30, p. 123], which allows users to explicitly identify correlated data streams to facilitate efficient stream processing by jointly considering streams within the same partition. Partitions that give rise to temporal correlation among publishers are common in many IoT deployments. For example, sensors in close proximity to each others will often trigger around the same time when detecting the same physical phenomenon. Publishers associated with those sensors, will then produce correlated message arrival patterns to the messaging system.

We adapt the concept of partition keys in SRTM, and provide APIs to users that let them label correlated publishers with an identical **correlation group key**.

Returning now to the end of Step (2) of the workload distribution process, the Load Distributor 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 number of publishers to be assigned to a broker (based on $rcap$ and $rate/k$), the Load Distributor selects publishers from each correlation group in proportion to the number of publishers in the group (see Appendix A.1 for details and justifications).

2.5 SRTM Implementation

As mentioned earlier, SRTM is based on NSQ. In this section, we first introduce NSQ's architecture, and then present our approach to adding SRTM's functionality together with some challenges we had to address. While discussions on implementation challenges are closely tied to NSQ, several of the issues we encountered are broadly applicable to systems that need to achieve predictable performance while relying on high concurrency platforms (for scalability) such as Go runtime.

2.5.1 NSQ Architecture Overview

As shown in Fig. 2.4, NSQ comprises a set of goroutines. Messages from publishers arrive on separate TCP connections each handled by an `IOLoop` goroutine. Messages from publishers of a given topic are then passed to another goroutine, `Topic`. The `Topic` goroutine has a dedicated buffer into which `IOLoop` goroutines move messages. Once a `Topic` goroutine is scheduled for execution, it runs to completion by pulling messages from its buffer and forwarding them to `MsgPump` goroutines until the buffer becomes empty. The `MsgPump` goroutines sends messages to subscribers through separate TCP connections. The Go runtime employs a work-stealing scheduler to schedule all the goroutines on multicore platforms. The lightweight goroutines and the scheduler are important to the scalability of NSQ that may need to handle a large number of publishers through a single broker.

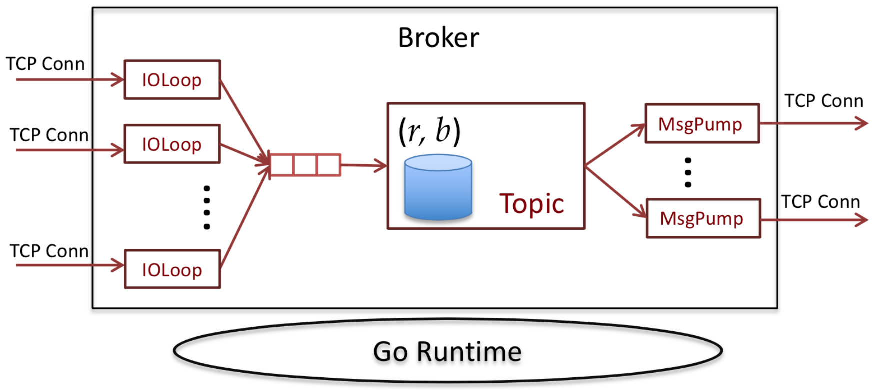


Figure 2.4: Goroutines in a NSQ broker

2.5.2 Rate Limiting

SRTM enforces rate limits through a token bucket that resides in the `Topic` goroutine. At runtime, each `Topic` goroutine keeps track of the state of its token bucket. When a `Topic` goroutine is scheduled, it pull messages from its incoming message buffer only if its token bucket has tokens. Otherwise, messages wait in the message buffer until tokens become available.

While the logic of a token bucket is well understood, it is non-trivial to implement its precise temporal behavior in a high concurrency environment such as Go runtime. Specifically, the token bucket determines whether a message is conformant based on its state (number of tokens) *and* the message arrival time. The straightforward approach to measure arrival times is to timestamp messages when the `IOLoop` goroutine reads them from the TCP receive buffer. However, because of the cooperative scheduling of goroutines, the time at which the `IOLoop` goroutine is scheduled to read messages can exhibit significant variations from when messages are first received in the TCP buffer. As the token bucket logic updates its token count based on how much time has elapsed since the message arrived, a late arrival timestamp could cause the token bucket to mistakenly delay messages that are actually conformant. Of particular concern is the increase in the scheduling delay of the `IOLoop` goroutine with the number of goroutines in the system. A large number of publishers (and consequently `IOLoop` goroutines) can then easily introduce large temporal errors in the token bucket's behavior.

To eliminate those errors, SRTM employs TCP-layer timestamping that timestamps the arrival of each TCP packet (sk_buff) inside the Linux kernel (in the tcp_v4_rcv function). Whenever an `IOLoop` reads data, the arrival timestamps (at the TCP layer) 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 mapping between received messages and TCP packets and assigns each message the timestamp from the correct TCP packet. As arrival timestamps at the TCP-layer are independent of goroutine scheduling, SRTM is able to enforce rate limits with higher temporal accuracy.

2.5.3 Handling Garbage Collection

Garbage Collection (GC) in the Go runtime can have a significant impact on the tail latency of message processing in NSQ. When GC is triggered, Go runtime uses marker goroutines to mark memory allocations, which can consume up to 25% of CPU time [49]. As a result, we found significant increases in the 99^{th} percentile in the message processing tail latency when GC is triggered.

As GC is usually triggered on demand in Go [10], 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, Go runtime forces GC if there is no GC in a 2 minutes (by default) interval. We disabled this feature so that in SRTM, GC is solely triggered on demand. Although these optimizations do not completely eliminate the impact of GC, they effectively mitigate it for the 99^{th} percentile tail latency in our experiments.

2.5.4 Adapting to Workload Changes

As discussed in Section 2.4.2, SRTM performs load distribution upon the arrival of a topic. At runtime, however, the traffic of the topic may shift among publishers, and therefore brokers, even if the topic's traffic remains conformant to its global (r, b) traffic envelope. The resulting mis-matched sub-token bucket configurations (r_l, b_l) could introduce DRL penalties at individual brokers. To avoid this, SRTM employs a TB Adaptor to dynamically adjust sub-token bucket configurations in response to traffic shifts at run time. Specially, every broker periodically reports to the TB Adaptor the rate and burst statistics of its topics. In our implementation, this is based on a per-topic 10s history window that tracks the average message rate and maximum backlog over that window in 1s increments. The TB Adaptor adjust token rates in proportion to average message rates, and token bucket sizes in proportion to maximum backlogs. To balance responsiveness and stability, SRTM administrators can adjust the length of the history window and the frequency of updates.

Note that the TB Adaptor only adjusts sub-token bucket parameters to avoid DRL penalties caused by shift in traffic among publishers. This does not address problems that may arise when a shift in traffic overloads a broker. Handling such scenarios calls for the ability to migrate publishers among brokers. Although this feature has been implemented (and is used in the profiling phase), its introduction as a runtime mechanism is left as future work.

2.6 Evaluation

This section presents an empirical evaluation of SRTM, and more precisely of the different design principles on which it relies. Section 2.6.1 starts with experiments designed to illustrate the impact of DRL on tail latency, while Sections 2.6.2 to 2.6.4 proceed with quantifying the relative benefits derived from each one of SRTM's three principles. This is realized through a progression of designs that incorporate SRTM's principles one at the time. Finally, Section 2.6.5 explores a more pragmatic aspect, namely, the amount of time SRTM takes to converge to a stable load distribution after the arrival of a new topic.

Testbed. The evaluation is carried out on a testbed consisting of 7 physical hosts. Fig. 2.5 offers an overview of the testbed. Hosts boast two 8-core Intel Xeon E5-2630 processors, 8 GB of memory, and are connected by 40 Gbps Ethernet links. Hosts $1, 2$ and $3, 4$ are dedicated to publishers and subscribers, respectively. There are 6 brokers in total, deployed over Hosts 5 and 6. Each broker has 2 dedicated CPU cores, which is also the default CPU configuration of Amazon-MQ instances [1]. The Load Distributor and TB Adaptor are deployed at Host 7.

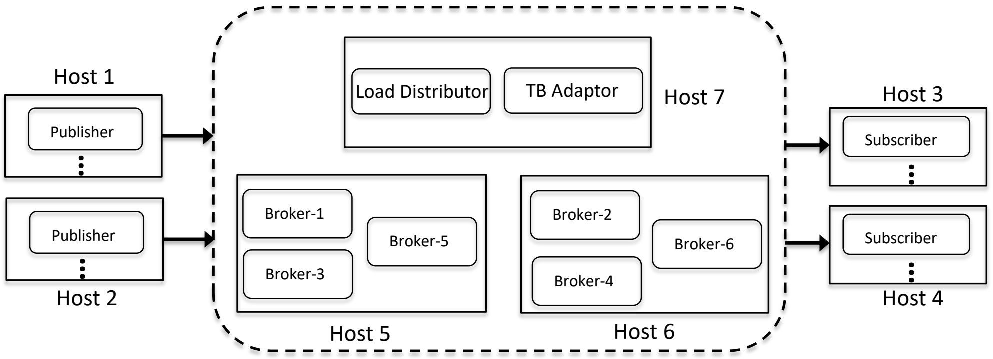
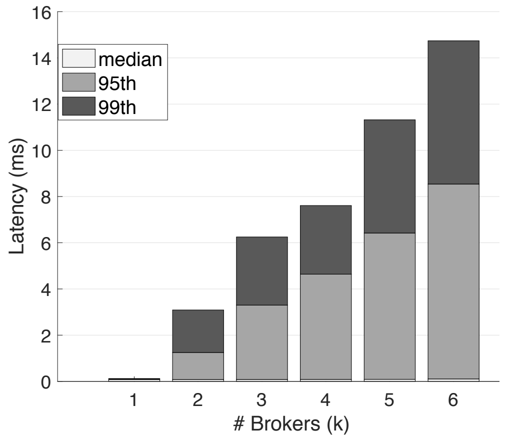


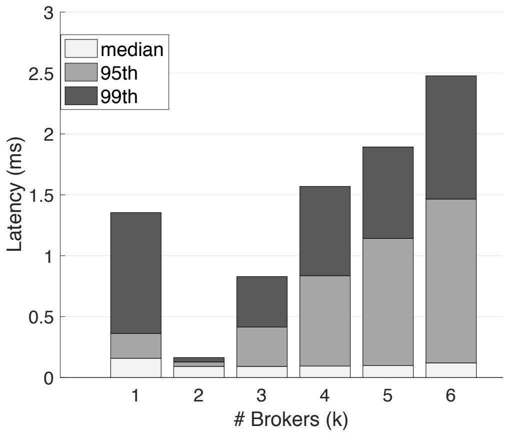
Figure 2.5: Testbed Setup

Workload. We generate a messaging workload that seeks to mimic IoT traffic. In a typical IoT setup, messages for a topic originate from multiple publishers, with publishers corresponding to a single sensor or a gateway aggregating a group of sensors. Messages can be *time-triggered* or *event-triggered*. Time-triggered messages are usually generated periodically. For example, intelligent transportation systems may require periodic traffic updates from sensors across a city [6, 21]. Conversely, event-triggered messages are generated upon detecting specific environmental changes. For example, building thermostats may trigger when temperature drops below (or exceeds) a certain threshold.

We emulate time-triggered traffic with publishers sending messages periodically. Because we were unable to secure real-world traces of event-triggered messages, we approximated the resulting traffic using a Poisson process (randomly occurring events). Both periodic and Poisson publishers may generate messages in a *batch*, with the *batch size* determining 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.



(a) 1,000 publishers



(b) 6,000 publishers

Figure 2.6: DRL penalty for Poisson workloads
(10 msg/s per publisher – batch size of 1).

We selected a 99th percentile end-to-end message latency of $1ms$ as the SLO[9] across all experiments. Whenever a given design is unable to meet the $1ms$ SLO, we report the performance of the best configuration. Additionally, topics' token buckets were configured using a profiling phase that gathered a representative traffic trace. The trace was used to perform an offline simulation[10] 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.

2.6.1 Illustrating the DRL Penalty

This section explores the impact of the DRL penalty on tail latency through two experiments (see Fig. 2.6) that differ in their message workload. In both experiments, the publishers of a new topic have access to the 6 brokers, which for consistency are initially all idle. The first

[9]It reflects the typical end-to-end latency in our testbed. A real-world deployment would likely have to account for larger network delays.

[10]For Poisson publishers, the approach behind Eq. (2.1) could instead be used.

experiment illustrates the DRL penalty when the topic's load is low, *i.e.,* a single broker is nominally able to accommodate the topic. The second experiment considers a heavier load for which a trade-off arises between access to more capacity when spreading the load across brokers vs. the resulting increase in DRL penalty. Both experiments rely on publishers that generate single messages (batch size of 1) according to a Poisson process of rate of 10 msg/s.

The first experiment involves a topic with $1,000$ (Poisson) publishers (a workload of $10,000$ msg/s) that are distributed across 1 to 6 brokers, with the topic's token bucket correspondingly split among them. Fig. 2.6a shows the median, 95th, and 99th percentile of the end-to-end latency for the 6 configurations. It illustrates the increase of the 99th percentile latency with the number of brokers across which the topic is split (it reaches $14.7ms$ for $k = 6$). This is because in low load configurations, as is the case here, the benefit from access to more broker capacity is small and does not offset the DRL penalty.

The second experiment (Fig. 2.6b) is similar except that it now uses $6,000$ publishers. The higher message rate overloads a single broker, as illustrated by the case $k = 1$ that exhibits a large tail latency. Splitting publishers between two brokers reduces the tail latency, as the decrease in message processing latency (from the lower broker load) exceeds the increase in DRL penalty (from splitting the token bucket). Distributing the topic across more brokers is, however, of no benefit, as the increase in DRL penalty again exceeds the decrease in message processing latency. This highlights the trade-off that Fig. 2.2 alluded to.

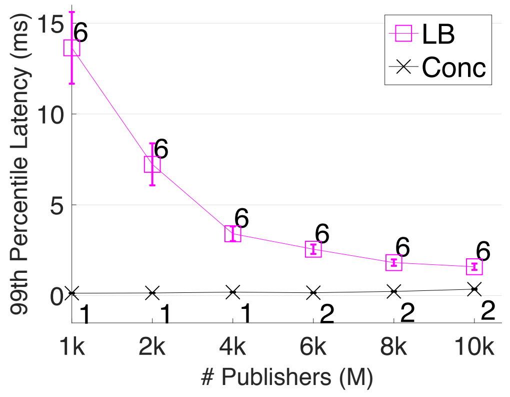
Next, we compare SRTM against a series of baseline solutions that incrementally incorporate the design principles behind SRTM, namely, *concentration*, *max-min*, and *correlation-awareness*. The experiments allow us to isolate the contributions of each principle, while assessing their overall impact when combined in SRTM.

2.6.2 The Benefits of Concentration

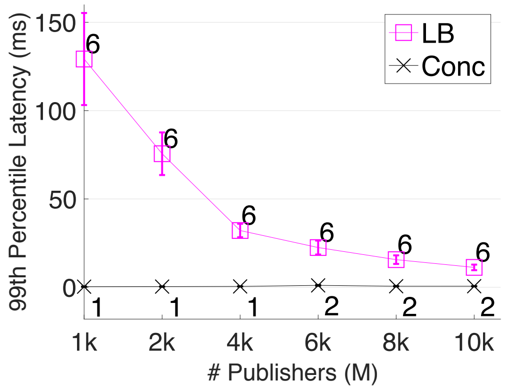
In this section, we evaluate the impact of *concentration* by comparing a baseline approach, **Load Balancing (LB)**, that evenly distributes a topic's publishers across all available brokers, to an alternative, **Conc**, that incorporates the *concentration* principle. Conc distributes publishers across brokers so as to equalize the *total* load on each broker, but unlike LB that carries this distribution out across *all* brokers, it limits itself to the *smallest* possible number of brokers the topic requires. This number is first estimated based on the topic's message rate and brokers' *rcap* values, and then validated using a measurement-based approach as described in Section 2.4.2.

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 again consider Poisson and periodic publishers. We vary the burstiness of the message generation process of each publisher by changing the size of the message batch they generate (1 or 10). Periodic publishers are independent of each other, with a randomly selected phase for their period. Experiment start again with idle brokers and are repeated 10 times. The results are shown in Fig. 2.7 with the mean and standard deviation of the 99$^{\text{th}}$ 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.

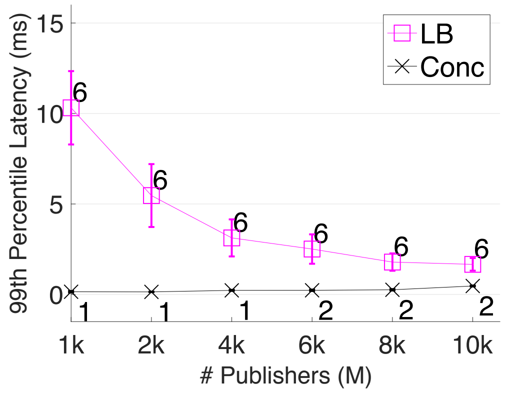
Figs. 2.7a and 2.7b report the results for Poisson publishers and batch sizes of 1 and 10, respectively, with Figs. 2.7c and 2.7d devoted to periodic publishers. Results are qualitatively consistent across scenarios, and illustrate the benefit of the *concentration* principle (Conc meets the topic's SLO for all configurations, while LB consistently fails to). The figures also highlight two relatively intuitive factors.



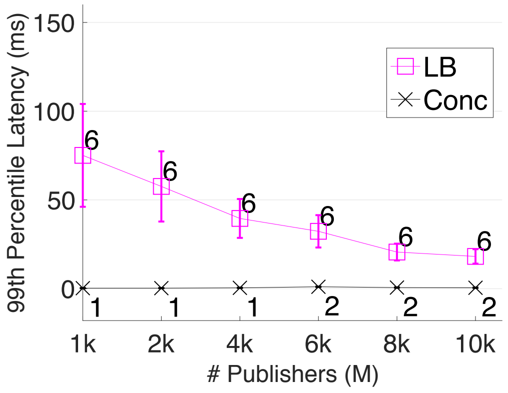
(a) Poisson, Batch=1,



(b) Poisson, Batch=10,



(c) Periodic, Batch=1,



(d) Periodic, Batch=10,

Figure 2.7: Impact of Concentration on 99^{th} percentile Latency. (Batch: batch size per publisher; average message rate per publisher: 10 msg/s.)

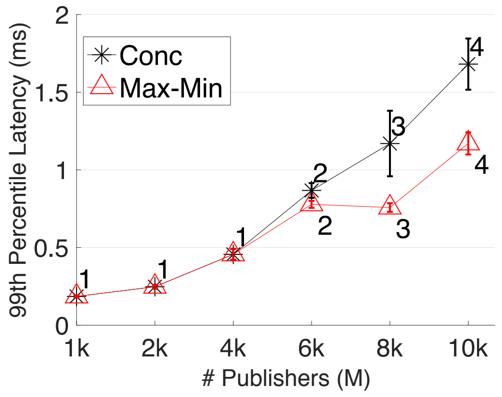
The first is the impact of the aggregate message rate on the DRL penalty. Specifically, as the number of publishers, and therefore the topic's aggregate message rate, increases, the penalty that LB incurs decreases. A return to Eq. (2.1) readily explains why. A higher token rate means a smaller token generation time (service time), and consequently a shorter delay as is well-known from basic queueing theory. 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 6 ways (as LB requires) 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 can again be explained by looking at Eq. (2.1). 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 waits for 10 tokens). It is this "amplification" factor that is behind the significantly worse performance of LB for batch arrivals.

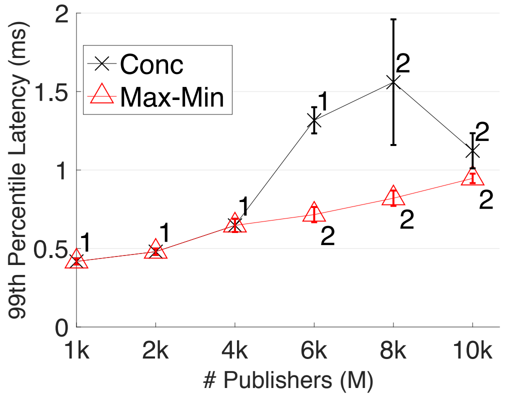
2.6.3 The Benefits of Max-Min

To evaluate the impact of the *max-min* principle, we reuse the scenarios of the previous section, but assume that a new topic finds brokers with varying pre-existing message loads (as shown in the caption of Fig. 2.8). We also introduce a new baseline algorithm, **Max-Min**, that extends Conc by incorporating the *max-min* principle. As 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.

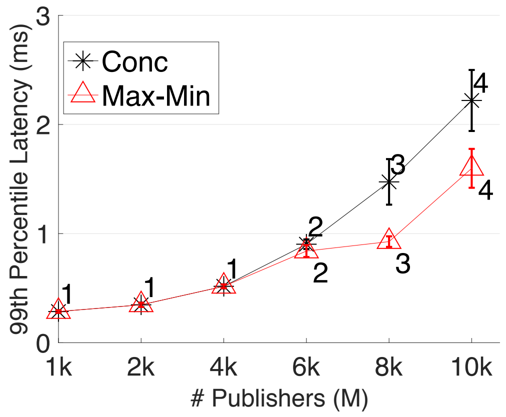
Figs. 2.8a and 2.8b compare Conc and Max-Min for a new topic with Poisson publishers, while Figs. 2.8c and 2.8d report similar results for periodic publishers. The presence of existing workloads on the 6 brokers affects the new topic's distribution across brokers as a function of its own workload. When it is low, it can fit 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 be split over multiple brokers. For example, when the topic boasts 8k publishers, it now needs to be distributed across brokers 1, 2, and 3. Conc equalizes the total load of the three brokers, while Max-Min instead



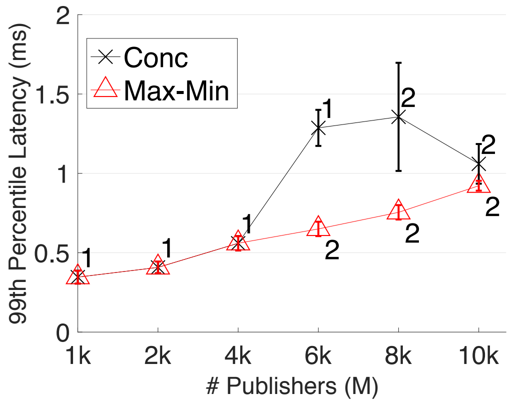
(a) Poisson, Batch=1



(b) Poisson, Batch=10



(c) Periodic, Batch=1



(d) Periodic, Batch=10

Figure 2.8: Impact of Max-Min on 99$^{\text{th}}$ percentile Latency.
Initial (Poisson) load on brokers 1-6 (in kmsg/s): 10, 40, 50, 60, 70, 80 (w/ batch size of 10 messages).

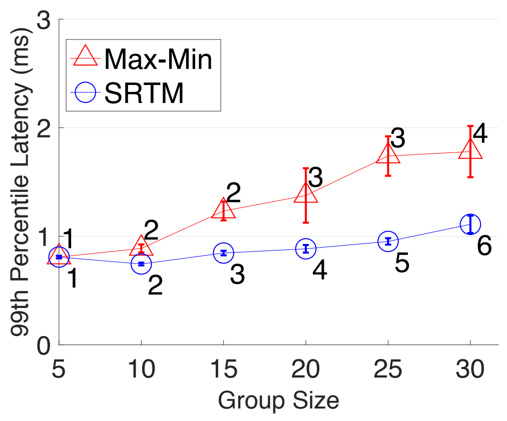
seeks to maximize the topic's load on broker 3 that, because it has the heaviest existing load, receives the smallest share. This results in a 99$^{\text{th}}$ percentile latency of 1.50ms for the publishers assigned to broker 3 under Conc, while it is 0.87ms under Max-Min, and the difference primarily arises from the larger DRL penalty under Conc.

As the two sets of figures show, Poisson and periodic publishers yield similar outcomes, but the experiments also reveal an interesting yet ultimately intuitive behavior when it comes to the impact of burstiness in the topic's arrival process. In particular, Figs. 2.8c and 2.8d display performance for a new topic with publishers that generate bursts of 10 messages, and both Conc and Max-Min perform as well if not better as when publishers generate bursts of size 1. This initially counter-intuitive behavior is because, in this scenario, the performance of the new topic is dominated by the processing of the `IOLoop` goroutines associated with its publishers (recall the overview of Section 2.5.1). Under a bursty arrival process, a publisher's `IOLoop` is scheduled less frequently, which results in a smaller number of simultaneously active `IOLoop`s that need to be serviced by the Go runtime scheduler. Such reduced scheduling overhead results in a broker being able to handle a higher overall message load under both Max-Min and Conc.

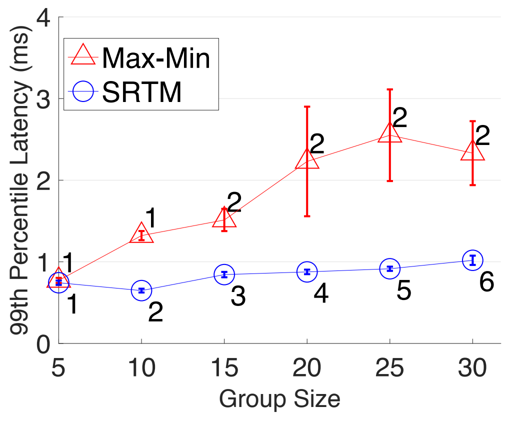
Another interesting behavior that Figs. 2.8c and 2.8d reveal is the improvement in performance of Conc when the new topic goes from 8k to 10k publishers. The reason is again differences in DRL penalty. With 8k publishers, the "left-over" number of publishers assigned to the second broker is smaller than with 10k publishers, and consequently the resulting DRL penalty is higher. This illustrates the primary weakness of relying on concentration only, as it can result in residual assignments to the last broker that produce very high DRL penalties. Again, this is the primary motivation behind the Max-Min principle.

2.6.4 The Benefits of Correlation Awareness

As discussed in Section 2.4.3, correlation in how publishers generate messages is captured through correlation groups. We then compare Max-Min to **SRTM** that incorporates 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 so as to



(a) Poisson Topic



(b) Periodic Topic

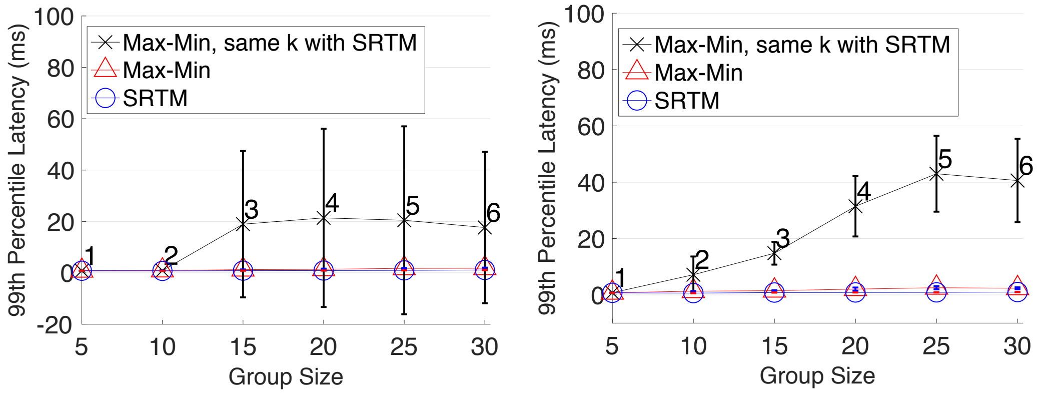
Figure 2.9: Impact of Correlation-aware Allocation.
(Each publisher has a batch size of 10 and an average message rate of 10 msg/s; Topic has 1,000 publishers).

reduce arrival burstiness in proportion to the reduction in bucket size). Towards isolating the impact of correlation, we again assume that a new topic arrives to a set of idle brokers.

Additionally, because correlation groupings can be coarse, *e.g.,* reflecting physical proximity rather than precise synchronization, we consider two scenarios. The first assumes that publishers within the same group are perfectly correlated, *i.e.,* their message generation times are precisely synchronized, while the second relaxes this assumption by introducing variations in the times at which publishers in the same group generate messages.

Scenario 1: Accurate correlation

In this set of experiments, publishers marked as belonging to the same group are perfectly 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

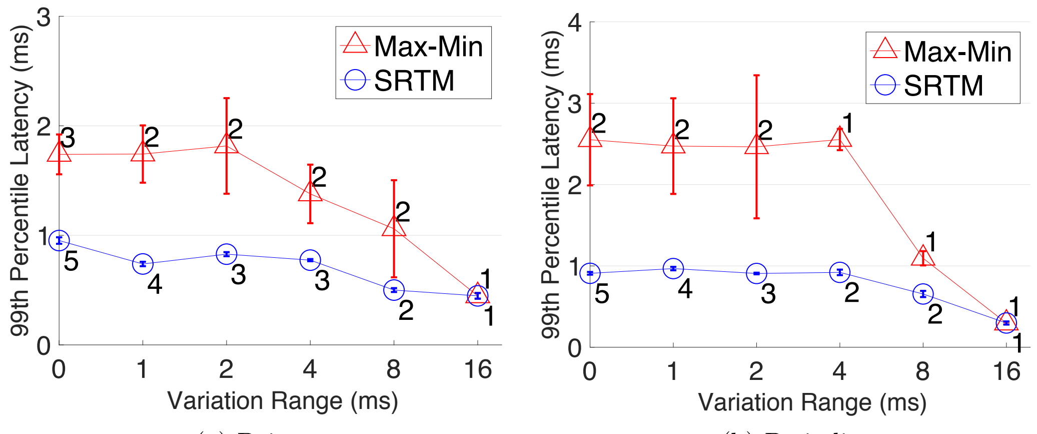


(a) Poisson Topic (b) Periodic Topic

Figure 2.10: Impact of Correlation-aware Allocation.
(Max-Min uses the same number of brokers as SRTM).

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. 2.9a and 2.9b 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. This allows it to meet the target SLO, even for groups of size 30. 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. 2.10 that also reports the performance of Max-Min when it tries to use the same number of brokers as SRTM. As anticipated, this makes its performance even worse.



(a) Poisson (b) Periodic

Figure 2.11: 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).

Scenario 2: Noisy correlation

This next set of experiment is explores the extent to which the benefits of a correlation-aware distribution of publishers remain when correlation information is inaccurate. Specifically, the message generation times of publishers within the same group are now (evenly) spread over an interval instead of perfectly concurrent. As the interval size increases, correlation between publishers weakens.

Results are reported in Fig. 2.11 that compares SRTM and Max-Min, again for both Poisson and periodic publishers. The experiments relied on the same type of publishers as in the previous section with a group size set to 25, and the interval across which messages from the same group were distributed ranged from $0ms$ to $16ms$ ($0ms$ corresponds to the configuration of the previous section). The results are again consistent for both Poisson and periodic publishers and demonstrate that, at least when the level of noise is small (of the order of a few percent 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

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 2.1: 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.

the size of the interval over which publishers' messages arrive, namely tail latency as well as the number of brokers across which publishers are distributed decrease for both SRTM and Max-Min. This behavior is a direct consequence of the lower burstiness associated with the increased "spreading" of message arrivals.

2.6.5 Load Distribution Latency

As described in Section 2.4.2, SRTM uses only the broker's average message load to estimate the residual capacity *rcap* available on each broker. This simple approach is because the complex internal architecture of NSQ makes accurately modeling the impact of higher order arrival statistics challenging if not impossible. As a result, SRTM resorts to a measurement-based solution to "fine-tune" its initial (imprecise) capacity estimates and the resulting publishers to brokers allocations. This fine-tuning is carried out in the online-fitting component of Step (3) of the load distribution mechanism. As previously mentioned, a potential disadvantage is that such an approach can take time to converge. Gaining insight into this *load distribution latency* is the purpose of this section.

Before reporting the results of those experiments, we however highlight that significant improvements are possible if topics' workloads are limited to a few well-understood traffic profiles for which customized models can be developed (as may be the case for specific applications). Such specialization is, however, beyond the scope of this paper, and instead we proceed next to quantify the load distribution latency in one illustrative configuration.

Specifically, we consider scenarios involving a new topic arriving to a set of empty brokers. This latter assumption arguably simplifies the load distribution process, as the brokers are homogeneous in their spare capacity. Nevertheless, the basic steps of SRTM's load distribution remain present. The topic's message load stands at a relatively low level of $10k\ msg/s$ ($1,000$ publishers, each with a message rate of 10 msg/s and a batch size of 10), and we vary its burstiness by changing the publishers' group size (publishers in a group are perfectly synchronized). When burstiness is low (small group size), a single broker is able to accommodate the new topic, but as burstiness increases, so does the number of brokers needed, with 6 brokers eventually required when the group size reaches 30.

Because the topic's workload (message rate) is low and brokers are initially empty, the first *rcap*-based assignment always starts with $k = 1$ (a single broker is deemed to have enough capacity). Depending on the topic's burstiness, additional brokers may, however, be needed (resulting in an increase in k). This is reflected in the different rows of Table 2.1, where each row corresponds to a different level of burstiness, and within a given row, a column (k value) represents one iteration of Step (3) of the load distribution.

An iteration starts with an assignment of publishers to a broker[11] and 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

[11]In the first iteration, all publishers are assigned to broker 1.

accommodate[12]. Entries in the table give the number of search rounds. Each round lasts 60 secs to ensure a representative traffic sample and are the primary contributors to load distribution 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 2.1 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 in excess of 15 minutes. This is clearly long, though not unreasonable when dealing with, say, an IoT deployment that may last for weeks or months. Additionally and as mentioned earlier, if variations in traffic profile parameters can be constrained, better models for estimating $rcap$ are feasible and would reduce the number of iterations.

2.7 Related Work

Distributed rate limiting (DRL) is supported by many cloud services through software platforms such as Cloud Bouncer [7], Tyk [24] and Doorman [9]. The existing platforms provide coordination protocols to adapt rate limiting parameters based on traffic of different workload partitions. Earlier research on distributed rate limiting generally focused on fairness and accuracy of distributed rate limiting. Raghavan et al. [59] supported DRL through

[12]In our experiments, the search stops as soon as latency is within 20% below the SLO target of $1ms$.

distributed traffic policing in which each rate limiter drops packets with a probability proportional to the excess global traffic demand. Stanojevic et al. [66, 65] proposed unified models for load balancing and DRL in which DRL is used to achieve fairness in resource allocation. Retro [53] is a resource management framework that employs distributed rate limiting to achieve fairness and isolation in resource usage among tenants. However, these previous work on DRL has not addressed the impact of DRL on latency.

To reduce operation or energy cost, there has been a significant body of work on *concentrating* workloads at the fewest servers subject to SLOs. WorkloadCompactor [79] concentrates workloads (with latency constraints) by shaping each co-located traffic stream with optimized token bucket (r, b) parameters. STeP [67] realizes concentration by co-locating tenants with compatible resource usage patterns, while several other systems [69, 38, 36, 48] improved *concentration* by co-locating latency-sensitive workloads (VMs) with data-intensive (batch-processing) workloads (VMs). Notably, while these earlier work achieved concentration by mitigating its impact on processing delays, we exploit concentration to reduce DRL penalty, i.e., access delays of token sub-buckets enforcing distributed rate limiting.

Load balancing is commonly adopted in distributed cloud services. The common objective of load balancing is to balance the *aggregate* load or performance at each server. Load balancers may be implemented at the data plane (e.g., Maglev [37] and Cheetah [28]) or the control plane (e.g., Pesto [41] and Romano [57]). While load balancing is an effective approach to achieve fair processing delays, we observe that it is insufficient to balance only the aggregate load in the presence of DRL. Instead, we propose the max-min principle to balance *per-topic* load to avoid excessive DRL penalty.

Correlation-aware workload assignment has been explored in the context of cloud computing. This approach generally requires capturing correlations in resource usage (e.g., CPU and network) based on workload traces. To reduce CPU contention, STeP [67] co-located anti-correlated databases based on FFT covariance, while Verma et al. [70] proposed to consolidate anti-correlated applications based on Pearson correlation. To optimize power consumption, Carpo [72] consolidated loosely-correlated flows (based on Pearson correlation) in data center networks. While these previous works leverage correlation to reduce operation and energy cost, we exploit correlation to reduce DRL penalty. In addition, we allow users to specify correlation groups based on application semantics. This practical approach takes advantage of the characteristics of IoT domain to avoid the practical challenge of measuring correlation in highly concurrent messaging systems.

2.8 Conclusion

The paper's main contributions are in identifying and explicating the DRL penalty that unavoidably arises when rate limiting is distributed across servers, and in designing and developing a system, SRTM, capable of mitigating this penalty with a focus on its use by IoT applications.

The SRTM system 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.

SRTM is fully operational, but relies on a measurement-based solution to accurately match arbitrary workload to system resources. This matching can in some cases be time-consuming.

A promising extension for configurations involving more specialized workloads involves developing an accurate model-based resource matching solution that would overcome this limitation.

Chapter 3

VATC: Real-Time Communication at Infrastructure Layer

3.1 Introduction

As computer hardware, e.g., multi-core CPU, has increased in power, so has the use of virtualization technology in data centers and clouds. This two-prong progress has fostered Infrastructure as a Service (IaaS). In such a virtualized environment, applications/services are deployed in Virtual Machines (VMs), so that the network I/O performance of the virtualized hosts, which build the infrastructure layer, becomes a critical component of meeting the communication requirements of distributed real-time applications. Examples of such applications include shipboard computing [68], where distributed mission-critical and safety-critical tasks are deployed in multiple servers, and are subject to end-to-end deadlines. Similarly, end-to-end latency constraints are also present in enterprise data centers and industrial automation systems that are increasingly deployed in virtualized environments. The main challenge in such settings is that VMs running latency-sensitive (soft real-time) applications are likely to be deployed in the same host as VMs that run bandwidth-intensive (bulk) applications. Therefore, network flows in the same virtualized host have to share CPU and NIC resources. While NIC sharing mechanisms are reasonably well understood, CPU consumption for network traffic processing involves a complex range of interactions that are harder to predict and control. This makes meeting latency requirements for distributed real-time applications in the presence of competing non-real-time applications challenging.

In non-virtualized hosts running standard Linux, a queueing discipline (QDisc) layer implements traffic control functionalities, including traffic classification, prioritization and rate limiting. Several different queueing disciplines are provided in the Linux kernel. In particular, disciplines such as Prio [15] and FQ_CoDel [45] are able to prioritize network traffic. When combining with Hierarchical Token Bucket (HTB), these queueing disciplines can achieve differentiated and isolated **NIC sharing** among latency-sensitive and non real-time network flows. These queueing disciplines are also used in virtualized hosts based on Xen [29], a widely-used open-source virtualization platform. From now on, following Xen's terminology, we use the term *domain* in place of VM. Xen employs a manager domain called domain 0 (dom0) to manage the other domains (guest domains). Dom0 is also responsible for processing network traffic on behalf of the guest domains. By default dom0 runs a Linux kernel. Virtualization, however, may introduce priority inversions within the transmission (TX) and reception (RX) routines. Those limitations result in mixed **CPU sharing** between network flows, which can not be solved by the standard Linux queueing disciplines. Therefore, there is the potential that latency-sensitive traffic suffers from unpredictable delay in dom0.

This paper proposes virtualization-aware traffic control for network traffic in virtualized hosts, and implements the proposed approach in Xen. Specifically, the paper makes the following contributions: (1) it identifies the impact of QDisc traffic control mechanisms and the virtualization-related components in the network path of Xen; (2) it identifies limitations of Xen's network architecture, which leave CPU sharing between network flows unprotected; and (3) it introduces *VATC*, a *Virtualization-Aware Traffic Control* scheme that achieves differentiated, isolated and fair resource sharing on modern multi-core CPUs, thereby offering greater latency control and predictability for latency-sensitive applications.

3.2 Motivations

As mentioned earlier, Xen uses a Linux-based manager domain (dom0) to handle network traffic from and to guest domains. Xen's network stack is thus similar to that of the standard Linux distribution, but with additional virtualization-related components. Understanding to what extent virtualized platforms can offer latency guarantees, therefore, calls for exploring how Linux policies and mechanisms, including queueing disciplines, the sharing of transmission and reception queues, and the frequency with which interrupts (notifications) are generated and serviced, interact in a virtualized environment. A first contribution of this paper is, therefore, to offer a careful study of such interactions and how they affect latency under different traffic configurations.

In this section, we first review standard Linux packet transmission and reception routines, which have been stable since version 2.6. Then we study the architecture of dom0, but more focus on the additional virtualization-related components. We select Linux 3.18, identify the limitations and explore the implications for latency guarantees. Addressing those issues is the main motivation behind the design of VATC. Note that in this section, although we only introduce Linux and dom0 on a single CPU core, the limitations that we identify exist on multi-core environments as well.

3.2.1 Network Stack in Standard Linux

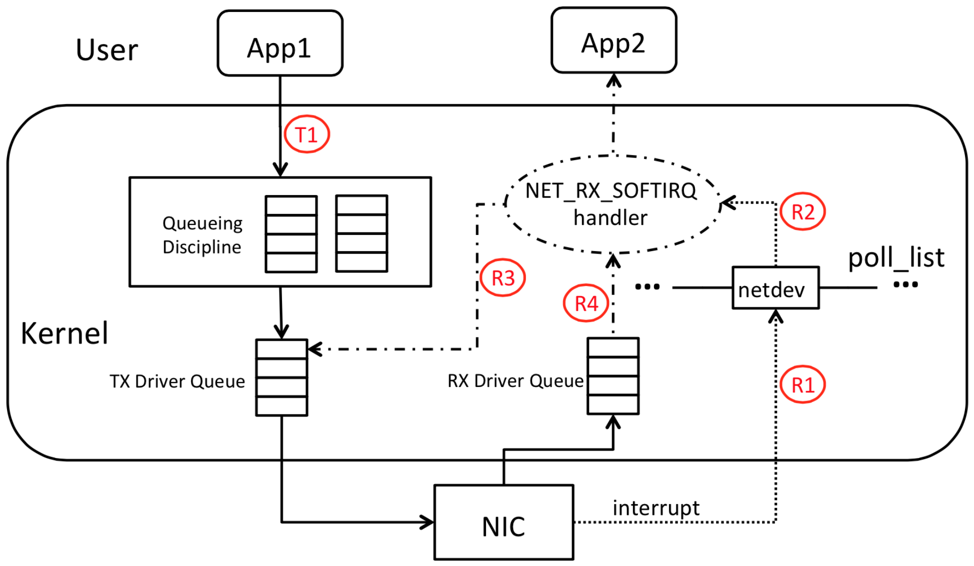


Figure 3.1: Transmission/Reception in Standard Linux

Transmission:
T1: packets are transferred from app to TX driver queue
Reception:
R1: interrupt handler inserts netdev into the poll_list and
R2: raises NET_RX_SOFTIRQ
R3: NET_RX_SOFTIRQ handler cleans up TX driver queue and
R4: delivers packets from RX driver queue to app

Transmission Routine in Standard Linux

Figure 3.1 shows network transmission and reception routines in standard Linux. In standard Linux, packets from applications are processed by the network stack in the Linux kernel. Because the virtualization-extensions of Linux only change the link layer, we omit session, transport and network layers in the figure. Packets are enqueued into the appropriate queueing discipline (QDisc) queue(s) in the link layer, which is where Linux implements traffic control. The TX driver queue, also known as the ring buffer, is a FIFO queue that works closely with the NIC.

Queueing Discipline: The QDisc layer implements traffic classification, prioritization and rate limit. These settings can achieve differentiated and isolated NIC sharing among network flows. In Linux, QDisc settings are configured through the TC command. By default, Linux uses pfifo_fast as the queueing discipline for traffic control.

Depending on the QDisc configuration, Linux can prioritize packets and reduce queueing delay for latency-sensitive applications. Prio [15] is a queueing discipline that has one queue per priority. It works in cooperation with packet filters, which distribute packets from different flows (applications) into different queues. When the dequeue function of Prio is called, the order in which packets are dequeued from queues goes from high-priority to low-priority. Hence, assigning latency-sensitive applications to the highest priority queue can ensure shorter queueing delays. FQ_Codel [45] [55] is another queueing discipline that works to reduce queueing delay. FQ_Codel has one queue per flow, with a quantum for each queue. Once the quantum is reached, the corresponding queue is classified as a negative deficit queue, which has low-priority. This policy thus offers short queueing delays to latency-sensitive applications with low throughput.

In addition to these differentiation settings, QDisc layer provides hierarchical token bucket (HTB) to throttle and shape each network flow. This setting realizes isolated NIC sharing, which protects a NIC from being flooded by (intentionally or accidentally) misbehaving flows. When a differentiated queueing discipline is used, it is essential that high-priority flows are respectively throttled by HTBs. Otherwise low-priority flows easily suffer from extreme starvation due to misbehaving high-priority traffic.

TX driver queue: Packets remain pending in the TX driver queue until the next DMA transfer to the NIC. Congestion in the TX driver queue can, therefore, have a critical influence on packet transmission delays. Congestion arises when too many large packets are forwarded to the NIC and the hardware is not capable of processing them fast enough.

There is typically a limit to the size of TX driver queue, which controls the number of pending packets. However, this control is insufficient to prevent congestion when the bulk of the NIC traffic consists of large packets. This limitation has been addressed in recent Linux kernels (after 3.3), by the introduction of a Byte Queue Limit (BQL) [43] policy, which limits the number of *bytes* in the TX driver queue of the NIC. In cooperation with the QDisc layer, BQL can greatly reduce the queueing delay in the TX driver queue, even in the presence of large packets. With BQL, the size of the TX driver queue is limited dynamically, based on the traffic mode and throughput. Once the queue size hits the limit, the QDisc layer holds or drops subsequent packets.

In most NIC drivers, when packets are successfully sent by the NIC, a TX completion interrupt is triggered. The interrupt handler puts a netdev (a software data structure representing the NIC driver) device into the poll_list, which is a per_CPU data structure in Linux. At the end of the interrupt handler, a software interrupt, called NET_RX_SOFTIRQ, is raised, whose handler services the poll_list. The NET_RX_SOFTIRQ handler processes the network devices in the poll_list in a round-robin order, with a quantum of 64 packets. When the netdev device is fetched, the NET_RX_SOFTIRQ handler invokes the NAPI poll() method of the NIC driver. Depending on the NIC driver, the NAPI poll() method may perform different actions. In the NIC driver used in our experiments, the NAPI poll() method cleans up the TX driver queue and receives packets from RX driver queue. Other NICs have separate TX and RX interrupts.

The TX completion interrupt handler cleans up the TX driver queue, while the RX interrupt handler raises NET_RX_SOFTIRQ. The NAPI poll() method of these NIC drivers only does packet reception. Once the queue size is under the BQL limit, the interrupt handler notifies the QDisc layer to resume releasing packets to the TX driver queue. The interval between TX completion interrupts (the *interrupt throttle* rate) can be configured. In clusters and data centers, where low-latency communication is vital [46], users tend to configure a small interval. However, too frequent interrupts can generate heavy CPU workloads and adversely impact progress of the packet transmission and reception routines. Conversely, if the interrupt interval is too large, the TX driver queue may become congested because it is not refreshed often enough. In this case, packets remain pending in the QDisc layer and can experience long queueing delays there. Several NIC driver vendors offer dynamic interrupt throttle rates, which adjust the interval value on the fly based on whether the traffic is low-latency or bulk.

Reception Routine in Standard Linux

Figure 3.1 also shows network reception in standard Linux, in which packet arrivals trigger hardware interrupts, and the interrupt handler then puts the netdev (the same network device mentioned above) into the poll_list and raises NET_RX_SOFTIRQ. When the NET_RX_SOFTIRQ handler fetches the netdev and invokes the corresponding NAPI poll() method, packets are delivered from the RX driver queue to the upper layer. The NET_RX_SOFTIRQ handler function ends when either no device in the poll_list has packets pending, or it has serviced over 300 packets or has run for > 2 jiffies.

3.2.2 Network Stack Modifications in Xen

Recall that Xen relies on a manager domain, dom0 (domain 0), to handle network traffic. The network stack in dom0 is similar to that of standard Linux. Figure 3.2 shows the transmission and reception routines in dom0.

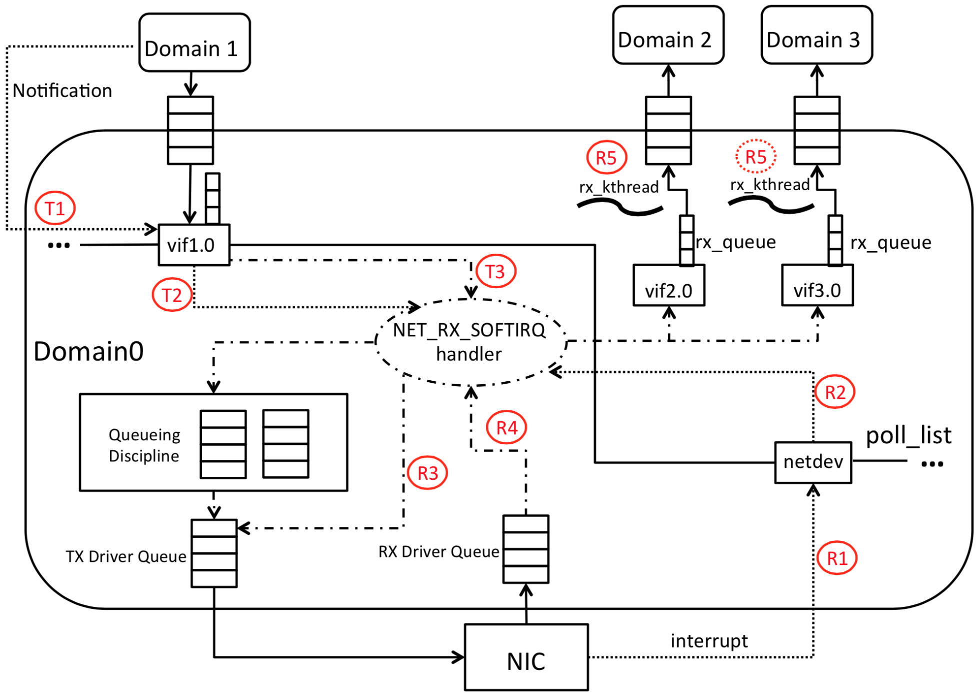


Figure 3.2: Transmission/Reception in Dom0

Transmission:
T1: notification handler inserts vif into the poll_list and
T2: raises NET_RX_SOFTIRQ
T3: NET_RX_SOFTIRQ handler delivers packets from vif(s) to TX driver queue
Reception:
R1: interrupt handler inserts netdev into the poll_list and
R2: raises NET_RX_SOFTIRQ
R3: NET_RX_SOFTIRQ handler cleans up TX driver queue and
R4: delivers packets from RX driver queue to vif(s), and triggers rx_kthread(s)
R5: rx_kthread(s) delivers packets from rx_queue(s) to guest domain(s)

Each guest domain has a corresponding vif device (a virtualization-related component) in dom0. There is a shared buffer between each pair of guest domain and vif device. Besides, each vif has a separate rx_queue and a dedicated kernel thread (rx_kthread) for reception.

Transmission Routine in Dom0: When a guest domain has a packet to transmit it first notifies dom0. The notification handler then inserts the corresponding vif device into the poll_list, and raises a NET_RX_SOFTIRQ. When the handler is scheduled, it processes all the devices in the poll_list in the same way as in standard Linux. After the NET_RX_SOFTIRQ handler function ends, other pending softirqs are processed. If a notification handler raises the NET_RX_SOFTIRQ before that processing finishes, the NET_RX_SOFTIRQ handler function is invoked again after other pending softirqs have been processed. In situations where the NET_RX_SOFTIRQ is frequently raised, it is therefore possible for softirq processing to run continuously for an extended period of time.

Reception Routine in Dom0: When a packet arrives, the hardware interrupt handler inserts the netdev device into the poll_list and raises a NET_RX_SOFTIRQ. The handler then delivers packets from the RX driver queue to the rx_queue of the destination vif device. Each vif device has a corresponding reception kernel thread (rx_kthread). When packets are inserted into an rx_queue, the corresponding rx_kthread is also triggered. Packets are then forwarded from the rx_queue to the guest domain when that rx_kthread is scheduled. This must wait, however, until after the softirq processing finishes, which can cause delays, as was discussed earlier.

3.2.3 Traffic Control Limitation in Xen

In either standard Linux or dom0 of Xen, network traffic processing requires NIC and CPU resources. Hence, when latency-sensitive network flows face non real-time (data-intensive) competitors, the traffic control mechanisms should be able to differentiate and isolate (rate limit) both NIC and CPU sharing. However, Xen traffic control mechanisms, which rely on QDisc settings, can only differentiate and isolate NIC sharing, while leave CPU sharing unprotected. We next summarize limitations of Xen network architecture. Those limitations, when combined with contention for CPU resource, can introduce unexpected delays in dom0.

Limitation 1: Priority Inversion between Transmissions:
When both latency-sensitive domains and other interfering domains are transmitting packets, their vif devices are all inserted into the poll_list and serviced in a round-robin order. A vif device holding latency-sensitive packets can, therefore, be delayed by other vif devices. Note that the default quantum for each network device in the poll_list is 64 (packets). Reducing the quantum can relieve this priority inversion. However, as we shall see next, there is another limitation that this approach cannot resolve.

Limitation 2: Priority Inversion between Transmission and Reception:
When the NET_RX_SOFTIRQ handler forwards a packet to the vif device, it wakes up the corresponding rx_kthread to do the follow-up tasks. However, the rx_kthread can only be scheduled after the softirq processing finishes. In CPU-bound situations with many (non real-time) domains sending packets at a high enough rate, NET_RX_SOFTIRQ can be raised frequently (by the notification handler), so that the handler continuously services the

poll_list, which can delay the running of rx_kthreads for a long time. This priority inversion arises between transmission and reception. Simply reducing the quantum for each network device in poll_list cannot resolve this priority inversion, because the duration for which softirq processing runs doesn't depend on the quantum value.

It is easy to find that existing queueing disciplines that Xen inherits from Linux cannot address these limitations, because those QDisc settings are dedicated to NIC sharing while the latency problems of dom0 are in the virtualization-related components that require CPU resource. Therefore, we propose *VATC*, a *Virtualization-Aware Traffic Control* scheme in which the network streams in Xen are prioritized and rate limited across all network components, thereby achieving differentiated and isolated resource sharing in dom0.

3.3 Design and Implementation

The simplest way to mitigate priority inversions among transmission flows is to extend priority awareness to the vif devices in the poll_list. However, priority inversions between transmission and reception are due to interference between softirq processing and rx_kthread. As a result, rather than implementing one priority-aware vif scheduler for packet transmission in the virtualization-related components, VATC is designed to provide fine-grained kernel-thread-based traffic control.

In Linux, both the scheduling policy and the priority of kernel threads can be configured by users. SCHED_FIFO is a preemptive fixed-priority scheduling policy, under which a high-priority thread can preempt a running low-priority thread. VATC builds on this concept by assigning the network traffic of high-priority domains to high-priority kernel threads, and the network traffic of low-priority domains to low-priority kernel threads. Besides, similar to the wisdoms of differentiated and isolated NIC sharing (e.g., Prio combined with HTB), VATC provides a rate limiting mechanism that throttles the CPU utilization of high-priority threads. Without this mechanism, low-priority traffic is vulnerable to CPU starvation.

In the rest of this section, we first introduce how VATC achieves latency (CPU utilization) differentiation by thread prioritization. On top of this design, we then introduce the rate limiting mechanism which realizes isolation. Both differentiation and rate limiting are firstly presented in a single CPU-core environment. But at last, we present how VATC scales to multi-core CPU, and introduce additional load distribution challenges.

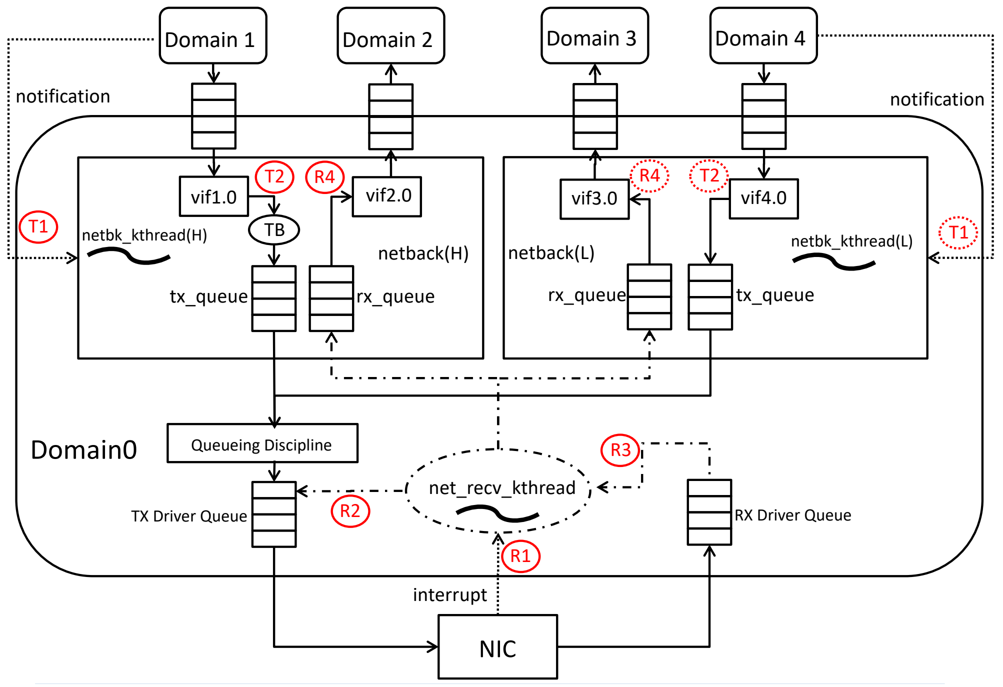


Figure 3.3: VATC: Virtualization-aware Traffic Control

Transmission:
T1: notification handler(s) trigger netbk_kthread(s)
T2: netbk_kthread(s) deliver packets from vif(s) to TX driver queue
Reception:
R1: interrupt handler triggers net_recv_kthread
R2: net_recv_kthread cleans up TX driver queue and
R3: delivers packets from RX driver queue to rx_queue(s) and triggers netbk_kthread(s)
R4: netbk_kthread(s) deliver packets from rx_queue(s) to guest domain(s)

3.3.1 VATC: Latency Differentiation

Figure 3.3 illustrates the overall structure of VATC. It introduces multiple (software) netback devices, and correspondingly multiple kernel threads (netbk_kthreads) to handle packet transmissions and receptions to/from different domains. These netbk_kthreads are configured with different priorities. Guest domains with the same priority (same latency requirement) share the same netback device and netbk_kthread. All the netbk_kthreads are scheduled under a SCHED_FIFO policy. The number of netback devices (netbk_kthreads) can be configured based on the number of priority levels needed.

For clarity, Figure 3.3 uses two priority levels. Domain 1 and domain 2 are running real-time (latency-sensitive) applications and are assigned to a high-priority thread. Domain 3 and domain 4 are low-priority domains running bandwidth-intensive (non real-time) applications. The high-priority netbk_kthread(H) handles the network traffic of the high-priority domains, and the low-priority netbk_kthread(L) handles traffic of the low-priority domains. The net_recv_kthread, which is triggered by TX completion and the RX interrupt handler, has the highest priority. It cleans up all the packets that have been transmitted from the TX driver queue and processes packets in the RX driver queue. Because both transmission and reception are handled by kernel threads, we remove the poll_list and software interrupt handling from VATC. Next, we review packet transmission and reception in VATC, as well as their interactions.

Packet Transmission in VATC: When a high-priority domain has packets to send, it notifies dom0. The notification handler in dom0 then triggers the corresponding high-priority netbk_kthread, which can preempt lower-priority threads. Packets from the high-priority domain are first enqueued [13] in the tx_queue of the corresponding netback device. The thread then checks whether the BQL limit of the TX driver queue has been reached. If it has, i.e., the TX driver queue is congested, it suspends itself until the net_recv_kthread cleans up the TX driver queue and refreshes the queue size. After cleaning up the TX driver queue, the net_recv_kthread notifies the suspended netbk_kthread to resume (if the new queue size is under the BQL limit). If multiple netbk_kthreads are suspended, they will resume one by one, based on their priorities. Note that each netbk_kthread can process packets in the tx_queue in FIFO order because the source domains of these packets have the same priority. Packets go through the QDisc layer [14], and are finally put into the TX driver queue. Once there are no more packets from any high-priority domain to enqueue and the tx_queue is empty, the high-priority thread stops, allowing a lower-priority thread to run.

VATC removes softirqs because they can lead to priority inversion when a high-priority thread has to process softirqs raised by low-priority threads. VATC therefore handles packet TX/RX entirely in the netbk_kthreads and the net_recv_kthread.

Packet Reception in VATC: When packets arrive at a virtualized host, the RX hardware interrupt handler wakes up the net_recv_kthread, instead of the NET_RX_SOFTIRQ, to process them. Once the net_recv_kthread is scheduled, it picks up packets from the RX driver queue and forwards them to the rx_queue of the destination netback device. For the rx_queue of the netback device, each enqueue operation wakes up the corresponding

[13]Before the enqueue operation, packets have to go through a token bucket (the "TB" in Figure 3.3) in order to enforce rate limiting. Details of this feature will be introduced at Section 3.3.2.

[14]Users can still configure QDisc schemes to control NIC utilization.

netbk_kthread, which will be scheduled after the net_recv_kthread finishes its work. If multiple netbk_kthreads are woken up by the net_recv_kthread, they will be scheduled based on their priorities. When a netbk_kthread is scheduled, it delivers packets from the corresponding rx_queue to the destination domains.

Interference between Transmission and Reception: In original dom0, the rx_kthread (for reception) can be preempted by the NET_RX_SOFTIRQ handler (for transmission). In VATC, the transmission/reception of real-time traffic is handled by high-priority kernel thread(s). Hence, the interference from either the transmission or reception of non-real-time traffic (handled by lower-priority kernel threads) is greatly reduced.

3.3.2 VATC: Rate Limiting

In the last section, we present how VATC achieves latency differentiation by prioritized kernel threads. However, unlimited high-priority network flows (threads) could easily starve low-priority flows (threads). In this section, we introduce the rate limiting feature of VATC, which eliminates CPU starvation between priorities.

Because traversing Linux (software) network stack is a per-packet operation, limiting packet rate becomes the key to throttle CPU utilization [11]. In VATC, because the network architecture of dom0 has been changed, we use the following experiment to re-validate the impact of packet rate on CPU utilization.

In this experiment, we dedicate one CPU core to dom0. In guest domains (at other cores), we use different UDP packet sizes to generate network streams that saturate the CPU of dom0. In Figure 3.4, we show both the maximum throughput (left y-axis) and packet rate (right y-axis) of this one-core dom0. As packet size increases, the throughput grows accordingly, while the maximum packet rate keeps relatively constant. These results validate that packet rate determines the bottleneck of dom0's CPU, which is consistent with the experience in original Linux box. Therefore, to throttle CPU consumption of network flows, we have to use packet-based rate limiters, instead of byte-based ones (e.g., HTB in QDisc layer).

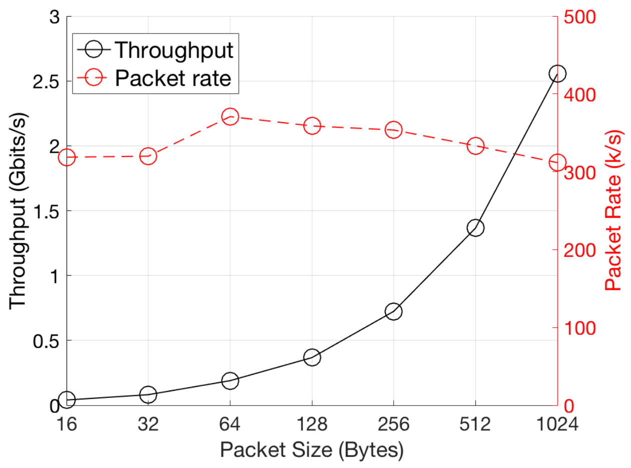


Figure 3.4: Impact of packet rate on CPU utilization

In original dom0, users can configure a rate limiter for each guest domain [27]. However, its implementation has two limitations:

- This rate limiter is byte-based, which can not effectively throttle CPU consumption of network flows.
- Different from token bucket, this rate limiter lacks an independent parameter for the allowance of burst, which results in loose traffic envelop.

To mitigate these two limitations, VATC rate limiting is implemented by adding one **packet-based token bucket** (TB) to each high-priority vif device. In a high-priority netback device, traffic from every vif has to go through the corresponding token bucket (as the "T2" step in Figure 3.3 shows), where each packet requires a token. In the absence of tokens, an arriving packet must wait for one. The (r, b) parameters of a packet-based token bucket are determined by the corresponding guest domain's SLO contract.

By limiting the packet rate of high-priority domains (vifs), CPU starvation of low-priority traffic can be eliminated. In VATC, the packet-based token buckets only apply to high-priority domains. With high-priority traffic throttled, low-priority traffic is free to use all leftover CPU resource. Note that the packet-based token buckets only aim to throttle CPU consumption, while users can simultaneously leverage HTBs in the QDisc layer to control NIC utilization.

3.3.3 VATC: Multi-core Support

As modern multi-core CPUs are widely used in virtualized hosts, it becomes common that dom0 executes on multiple cores. In this section, we introduce how VATC scales to multi-core dom0, and explore how network traffic is distributed in this environment.

In VATC, because most of network transmission and reception operations are handled by netback devices, scaling to multiple cores only requires duplicating netback devices accordingly. If a dom0 has N CPU cores, then VATC creates N netback devices for every priority, with each netback device dedicated to one core. Besides, VATC creates one net_recv_kthread per core to improve the scalability of packet reception from a NIC. With this multi-core settings, VATC keeps the number of (prioritized) netback devices on each core equal to the number of priority levels, thereby latency differentiation still preserved. Meanwhile, rate limiting can still be guaranteed by the per-vif packet-based token buckets without any changes.

As netback devices are duplicated, VATC now faces a new challenge: within a priority, how to fairly distribute traffic load across the N (per-core) netback devices. Considering the architecture of VATC, the solution relies on how vif devices are fairly assigned to netback devices.

VATC netback device is developed based on the xen-netback driver of dom0-3.10 (Linux-3.10). In this driver, vifs are assigned to netback devices with a Least-Number-First (LNF) policy: upon the creation of a new domain, its vif is permanently assigned to the netback with the least number of vifs. However, this assignment has two limitations that may introduce unfairness:

- It ignores the fact that vif devices may have different network traffic loads.
- The assignment is a one-shot decision. Therefore, given a group of guest domains, different sequences of domain creation may lead to assignments with extremely different fairness [15].

Therefore, we propose a vif rebalancing mechanism for load-distribution fairness. Within each priority, we maintain a sorted list of vifs with decreasing order of traffic loads. Upon the creation and shutdown of a domain (which lead to potential unfairness), we update the sorted list by inserting or deleting the vif. After the updating, we rebalance the vif assignments by running the **ReBalance** algorithm.

[15]This is commonly found in online bin-packing algorithms

Algorithm 1 ReBalance: Rebalance vif assignments

```
/*$netback_{1...N}$: the per-core netback devices to be assigned to*/
/*$vif_{1...M}$: the sorted list of vifs after updating*/
function ReBalance($netback_{1...N}$, $vif_{1...M}$)
    for $i = 1$; $i \le N$; $i++$ do //Initialization
        $netback_i.load = 0$
    end for
    for $i = 1$; $i \le M$; $i++$ do
        $netback_{min}$ = the netback with the least load
        if $netback_{min} \neq$ the current assignment of $vif_i$ then
            Migrate $vif_i$ to $netback_{min}$
        end if
        $netback_{min}.load = netback_{min}.load + vif_i.load$
    end for
end function
```

ReBalance is a simple greedy algorithm which adjusts vif assignments one-by-one. When selecting a netback device for a vif (line 8), we use a Least-Load-First (LLF) policy. In most of virtualized hosts, the creation and shutdown of guest domains are rare events. Therefore, this rebalance is generally triggered with limited times. During a rebalance, migrating a vif (line 10) only consumes less than 10 nano seconds based on our observation. Hence, total overhead of this rebalance mechanism is limited.

3.4 Evaluation

In this section, we respectively evaluate the three features of VATC: 1) latency differentiation; 2) rate limiting; and 3) multi-core support.

3.4.1 Evaluation: Latency Differentiation

As outlined in Section 3.2, various factors can delay soft real-time traffic in Xen. In this section, we explore a number of scenarios where such delays can arise, and both quantify their magnitude and analyze their causes. We evaluate latency and latency predictability for delay-sensitive traffic under our implementation of VATC, and under existing Xen traffic control mechanisms[16], i.e., Prio and FQ_CoDel in original dom0.

The evaluation is carried out on a testbed consisting of six physical machines, hosts 0 to 5. Host 0 is an Intel i7-980 six core machine with Xen 4.3 installed, on which dom0 is a 64-bit CentOS built on Linux kernel 3.18.0. Host 0 acts as the host server. Five other physical machines, hosts 1 to 5, run standard Linux. All machines are equipped with Intel 82567 Gigabit NICs and are connected by a TP-LINK TL-SG108 Gigabit switch. Because both Prio and FQ_CoDel are fine-grained packet schedulers, it is recommended [5, 78] that the TCP Segmentation Offload (TSO) and Generic Segmentation Offloading (GSO) of the NIC be disabled, which we do. This ensures that large packets with a size greater than the MTU (1,500 bytes in our system) are segmented in the kernel instead of in the NIC, and avoids long head-of-the-line blocking delays in the TX driver queue. Our NIC driver uses the NAPI poll() method, which is invoked by the NET_RX_SOFTIRQ handler, to clean up the TX driver queue. Figure 3.5 offers a schematic overview of the testbed.

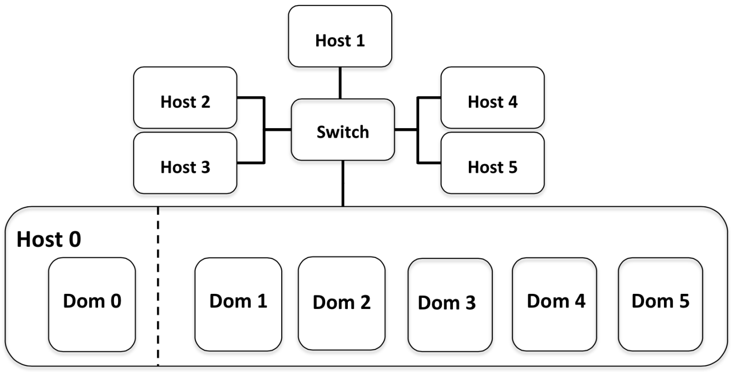


Figure 3.5: Testbed Setup

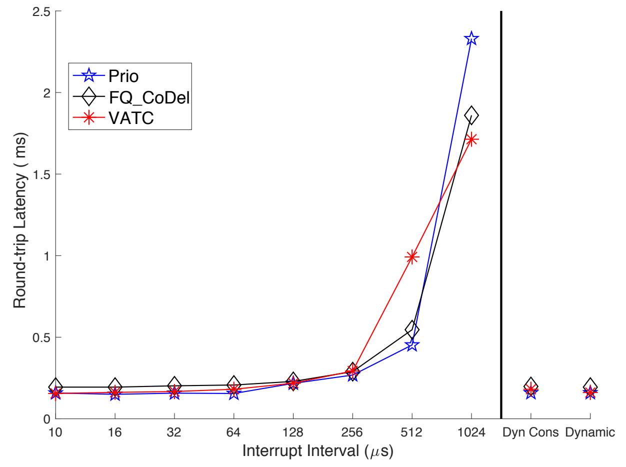
[16]FIFO is the default traffic control scheme in Linux, but as expected it performs poorly when it comes to latency guarantees. As a result, we only compare the latency of VATC to that of Prio and FQ_Codel.

In host 0, dom0 is given one dedicated physical CPU core. This is common practice to handle communication and interrupts [34, 77], and is also recommended by the Xen community to improve I/O performance [73]. We boot up five guest domains, domain 1 to domain 5 on host 0. Each of them is pinned to a separate physical CPU core to avoid influences from the VM scheduler. In our setup, domain 1 is the latency-sensitive domain and domains 2 to 5 are interfering domains. Hence under VATC, traffic from/to domain 1 is handled by a high-priority kernel thread in dom0, while traffic belonging to domains 2 to 5 is handled by a low-priority one.

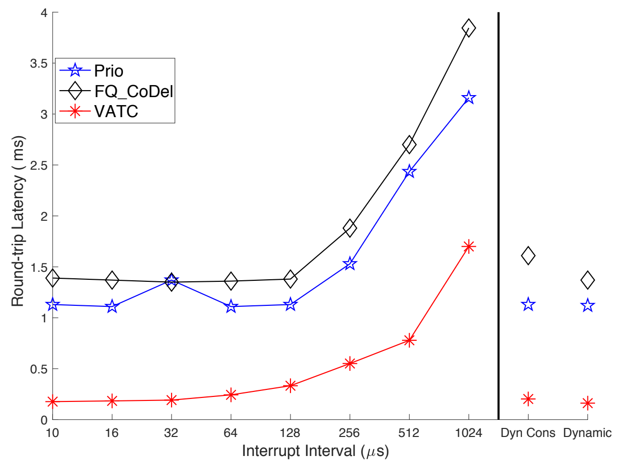
The round-trip latency between domain 1 and host 1 is measured as follows. Domain 1 pings (with ICMP packets) host 1 every 10 ms, and host 1 replies back. This traffic pattern seeks to emulate the behavior of common periodic real-time applications. Each experiment records latency values for $1,000$ ICMP request/response pairs. We report both median and tail latency (95th percentile). Tail latency is important to many soft real-time applications because it reflects latency predictability. In domain 2 to domain 5, we run the stream test of Netperf [61] to simulate non-real-time applications.

The Intel NIC in our hosts supports interrupt intervals from 10μs to 10ms. The Intel NIC driver also provides two adaptive modes, dynamic conservative (50μs to 250μs) and dynamic (14μs to 250μs). Both modes dynamically adjust the interrupt interval based on the type of network traffic, bulk or interactive. The dynamic conservative mode is the default mode of the Intel NIC driver. We evaluate both modes and a range of static values.

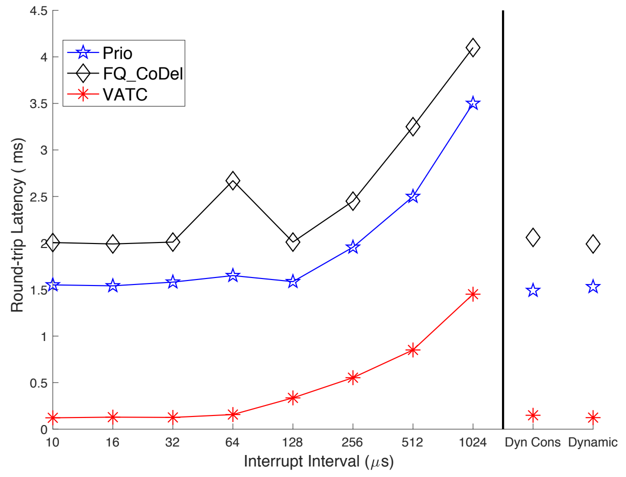
Latency of high-priority (latency-sensitive) traffic is measured in scenarios with CPU contention in dom0. CPU contention can occur when low-priority domains are sending many small packets. In those scenarios, NET_RX_SOFTIRQ handler is frequently triggered. This impact can be compounded by setting the interrupt handler interval to a small value, as the



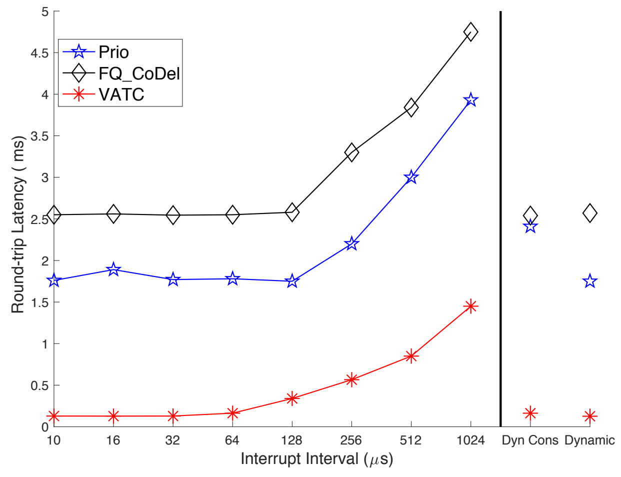
(a) One small UDP interfering stream



(b) Two small UDP interfering streams



(c) Three small UDP interfering streams

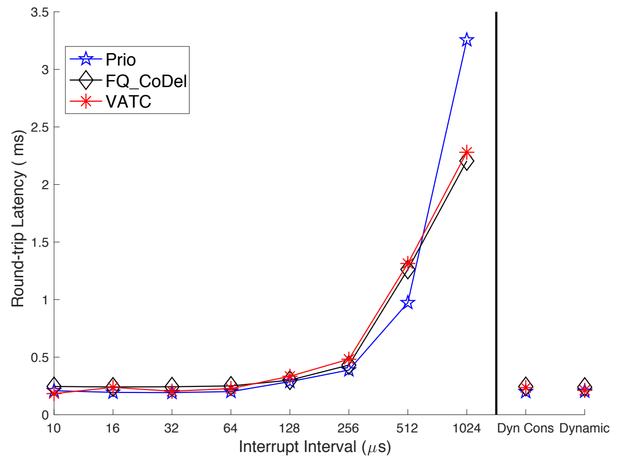


(d) Four small UDP interfering streams

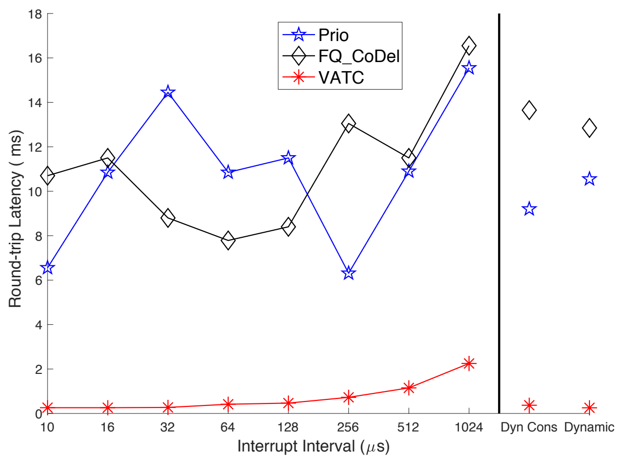
Figure 3.6: Increasing number of interfering streams: Median high-priority latency

bottom-half processing of the interrupt handler can then overload dom0. As we shall see, in these scenarios, the two *Limitations* identified in Section 3.2, introduce long queueing delays in virtualization-related network components.

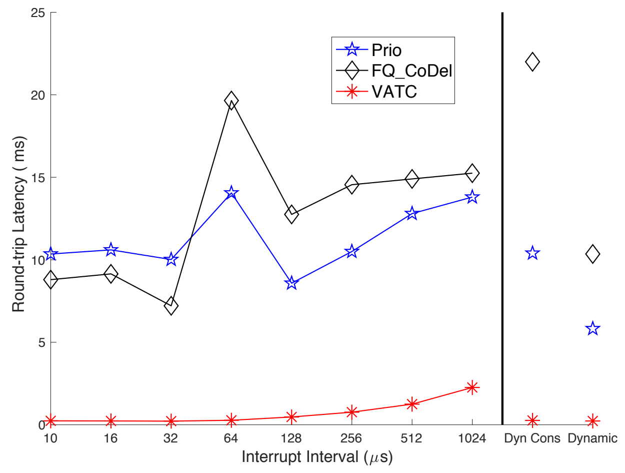
In the following experiment, we evaluate the latency of high-priority traffic in the presence of interfering low-priority streams consisting of small (1 byte) UDP packets. Figure 3.6 and Figure 3.7 show the round-trip latency (median and 95th percentile) of ICMP packets from the high-priority domain for different numbers of interfering low-priority UDP streams



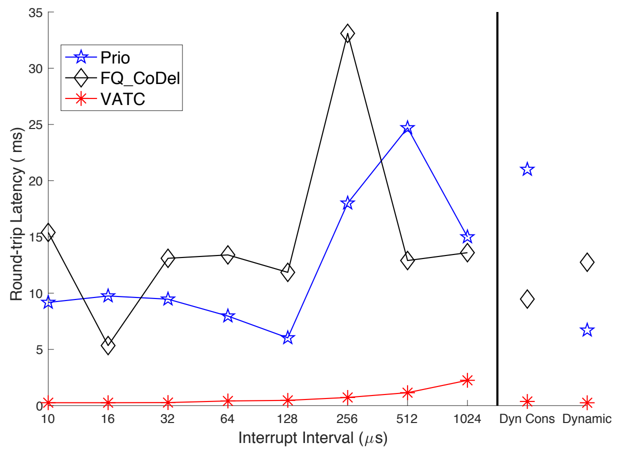
(a) One small UDP interfering streams



(b) Two small UDP interfering streams



(c) Three small UDP interfering streams



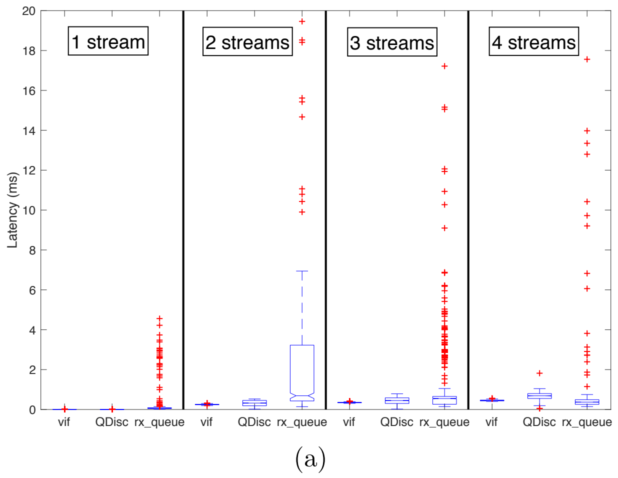
(d) Four small UDP interfering streams

Figure 3.7: Increasing number of interfering streams: Tail (95th percentile) high-priority latency

and different interrupt intervals (from 10μs to 1024μs and using the dynamic and dynamic conservative modes).

Impact of Interrupt Interval

In order to isolate the impact of different interrupt intervals, we focus on the case of one interfering stream (Figures 3.6a and 3.7a). We note that because packets are small, the dynamic conservative mode and the dynamic mode tend to default to setting the interrupt interval to the lower bound of their range, i.e., 50μs and 14μs, respectively. In these two figures, it is easy to find that latency under all three traffic control mechanisms grows as interrupt interval increases. This is because TX/RX interrupts are triggered less frequently with longer interrupt intervals.



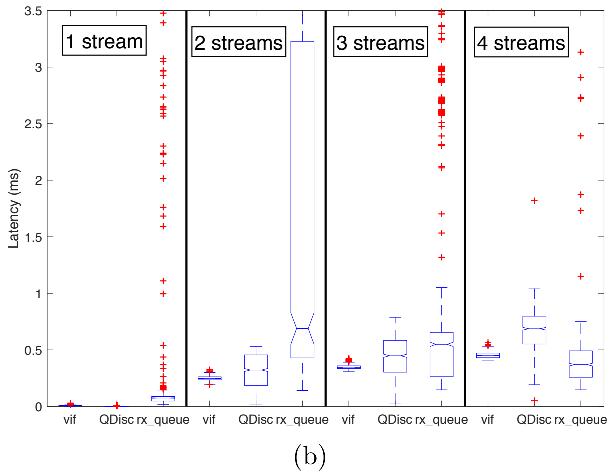


Figure 3.8: Increasing interfering traffic and # streams: high-priority latency (10μs interrupt interval, (b) presents a subset (values between 0 ms and 3.5 ms) of the results of (a))

Impact of the Number of Streams

The growth in the number of cores in a single host means that multiple guest domains can coexist within one host. Supporting latency differentiation with more than one domain is, therefore, an important scalability concern. In this part, we focus on the latency of the high-priority (ICMP) stream as the number of interfering domains increases.

Figures 3.6b, 3.6c, and 3.6d show the median latency of high-priority packets with 2, 3, or 4 interfering low-priority UDP streams, while Figures 3.7b, 3.7c, and 3.7d show tail latency for the same configurations.

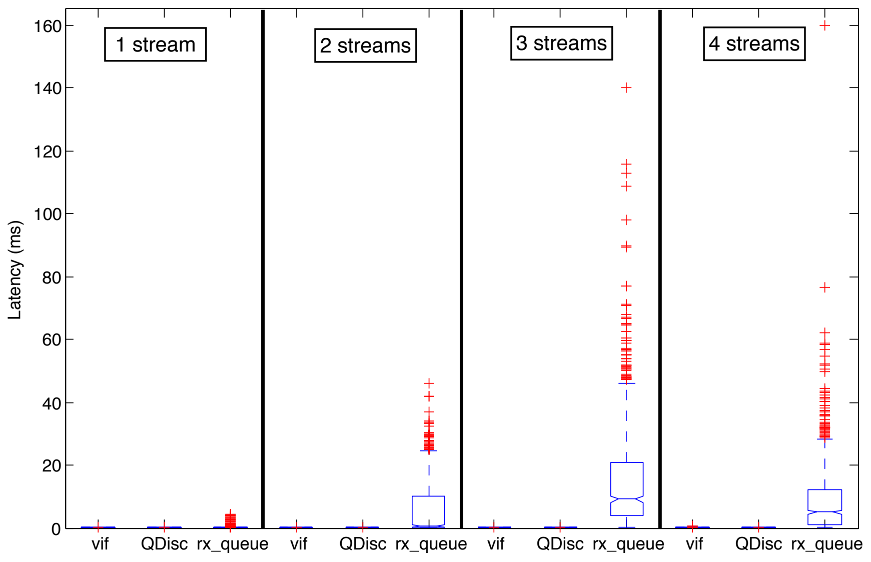


Figure 3.9: Constant interfering traffic, increasing # streams: high-priority latency (10μs interrupt interval)

We note from the figures that VATC's latency performance is unaffected by the number of interfering streams, as the high-priority netbk_kthread handling the real-time traffic cannot be preempted by the lower priority ones. In other words, VATC successfully mitigates *Limitation* 1 and *Limitation* 2.

In contrast, the high-priority stream under Prio and FQ_CoDel experiences increases in latency (median and tail) with the number of interfering streams. We performed a detailed analysis of how different dom0 components contribute to this latency increase (of high-priority packets). Under Prio and FQ_CoDel configurations, queueing delay may arise in:

- vif: when the NET_RX_SOFTIRQ handler is servicing other vif devices in the poll_list, high-priority packets are pending in the corresponding vif device;

- QDisc: when there is congestion in the TX driver queue, packets are kept waiting in the QDisc layer;

- rx_queue: After ICMP response packets are forwarded to the rx_queue, the corresponding rx_kthread must wait for the softirq processing to finish before it can be scheduled.

Figure 3.8 details delays at these three components under Prio (the results under FQ_CoDel are similar). As the number of interfering domains increases, so does the number of vif devices inserted in the poll_list, which contributes a small but steady increase in delay (at vif). The delay in the QDisc layer also increases, as there are now more pending packets in the TX driver queue, and the bottom-half processing of the TX completion interrupt is delayed since in our NIC driver the TX completion interrupt handler simply inserts the netdev device into the poll_list and leaves the bottom-half processing (clean up of the TX driver queue) to be executed by the NET_RX_SOFTIRQ handler. As the number of competing vif devices (in the poll_list) increases, the netdev device is serviced less frequently. This results in higher congestion in the TX driver queue and, therefore, longer queueing delays in the QDisc layer. In some other NIC drivers that clean up TX driver queue in hardware interrupt handler, the vif devices in poll_list won't delay the clean-up of TX driver queue, thus the queueing delay in QDisc layer may be reduced.

Both of those contributions to higher latency can be attributed to *Limitation 1*, but *Limitation 2* can be seen to have an even more pronounced effect. Two interfering streams (see Figure 3.8b) significantly affect the delay in rx_queue. This is caused by the NET_RX_SOFTIRQ handler repeatedly servicing the poll_list when the softirq is raised frequently by notification and interrupt handlers[17]. This can then result in the rx_kthread being delayed for an unpredictably long time as illustrated in Figure 3.7b which captures the tail of the delay distribution. For purposes of illustration, the 95th latency percentile is 40 times higher in

[17]NIC drivers that clean up the TX driver queue in the hardware interrupt handler without, therefore, raising NET_RX_SOFTIRQ might reduce this workload. However, because the dominant contribution to the softirqs is the notification handler, we do not expect this would be of much benefit.

Prio than in VATC. Interestingly though, this trend somewhat reverses as the number of interfering streams increases further beyond 2, because the NET_RX_SOFTIRQ is raised less frequently as more low-priority streams are added. In our experiments, a large fraction of the softirqs are raised by notification handlers from guest domains. Because dom0's CPU is overloaded, not all low-priority packets can be serviced in time, and a backlog of packets builds-up in the buffers between vif devices and the corresponding (low-priority) guest domains. This backlog prevents the corresponding guest domains from putting more packets into the buffer, and thus no new notifications are issued to dom0 until the buffer is refreshed.

The next experiment explores further the impact of notification frequency on latency performance. The results are shown in Figure 3.9, which parallels Figure 3.8 but keeps the total throughput of the interfering traffic constant and evenly distributed across streams (as opposed to each stream contributing their own independent traffic volume). This largely eliminates the possibility of congestion in the buffer between each vif and guest domain. Consequently, the notification frequency from guest domains is much higher than in the previous experiments. For example, with 4 interfering streams, we measure a notification frequency that is *100 times larger* than with the same number of streams each contributing their own traffic. This difference is largely responsible for the significant increase in latency seen between Figures 3.8 and 3.9 (the worst delay observed in the experiment of Figure 3.9 was 160 ms!). Of note in Figure 3.9 is the fact that while latency initially experiences significant increases as more streams are added, adding a fourth stream appears to contribute to a slight decrease. We were not able to pinpoint the exact sources of the decrease, but conjecture that it may be partially due to some streams now not always having new packets, which would in turn lower the notification rate.

Summary: *Limitation* 1 and *Limitation* 2 are both present under Prio and FQ_CoDel, which result in latency-sensitive traffic unprotected in dom0. VATC overcomes these limitations by dedicating a netback device and a prioritized kernel thread to each priority level, so that real-time streams are guaranteed with CPU resource, and are protected from interfering traffic.

3.4.2 Evaluation: Rate Limiting

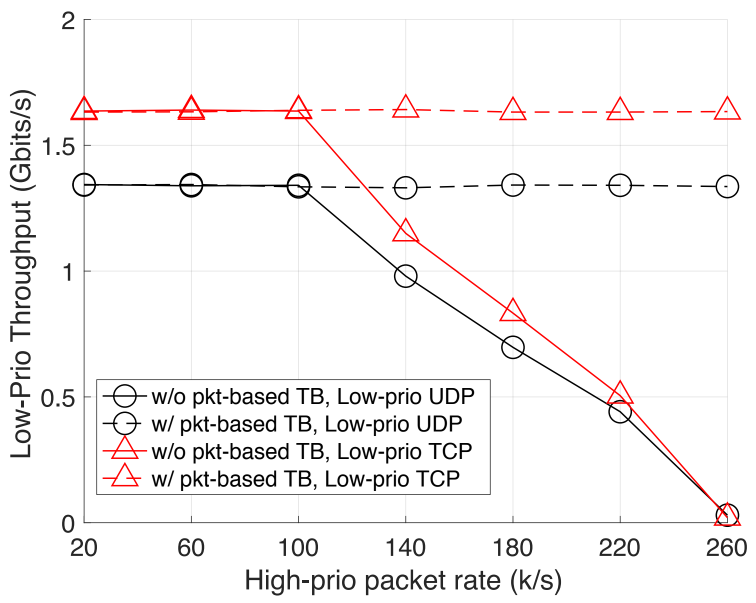


Figure 3.10: Impact of packet-based token bucket

To avoid starvation between priorities, VATC introduces rate limiting by adding packet-based token buckets to high-priority domains.

In this section, we evaluate the impact of this feature. In the following experiment, we create two high-priority and one low-priority guest domains. In each of the high-priority domains, we run iperf3 [12] to generate UDP network traffic with fixed throughput at 50

Mbits/s. Meanwhile, we change the packet size so that different packet rate is generated. In the low-priority domain, we run iperf3 to measure its UDP/ TCP throughput.

Figure 3.10 presents the experimental results. The x-axis shows the different total packet rate generated by the two high-priority domains. Under the pressure of the high-priority traffic, we evaluate the UDP and TCP throughput of the low-priority domain. Without the packet-based token buckets, the (UDP/TCP) throughput of the low-priority domain dramatically drops when the high-priority packet rate is over 100 k/s. This degradation is caused by CPU starvation. On the other hand, if we add a (20 k/s, 100) packet-based token bucket for each high-priority domain (vif), then the CPU resource (of dom0) consumed by the high-priority traffic is accordingly throttled, thereby the low-priority traffic getting enough CPU cycles. Hence, we can find that the UDP/TCP throughput of the low-priority domain is successfully protected.

3.4.3 Evaluation: Multi-core Support

Because the scalability improvement of multi-core has been well-studied, we focus on the load-distribution fairness of the static LNF vif assignment (introduced in Section 3.3.3) and VATC vif rebalancing mechanism in this section. In the following experiment, we dedicate core-0 and core-1 (N = 2) to dom0, and create 8 high-priority guest domains (dom-1 to 8) with packet rate at 10, 20, 30, 40, 50, 60, 70 and 80 k/s respectively.

Depending on the sequence of domain creation, the LNF method may assign vifs (traffic loads) with extremely different fairness. In the best case, the LNF method can assign traffic load of 180 k/s to core-0 and the other half to core-1 [18]. In the worst case, the LNF method

[18]This result occurs if the sequence of domain creation is, e.g., (dom-1, dom-3, dom-2, dom-4, dom-7, dom-5, dom-8, dom-6).

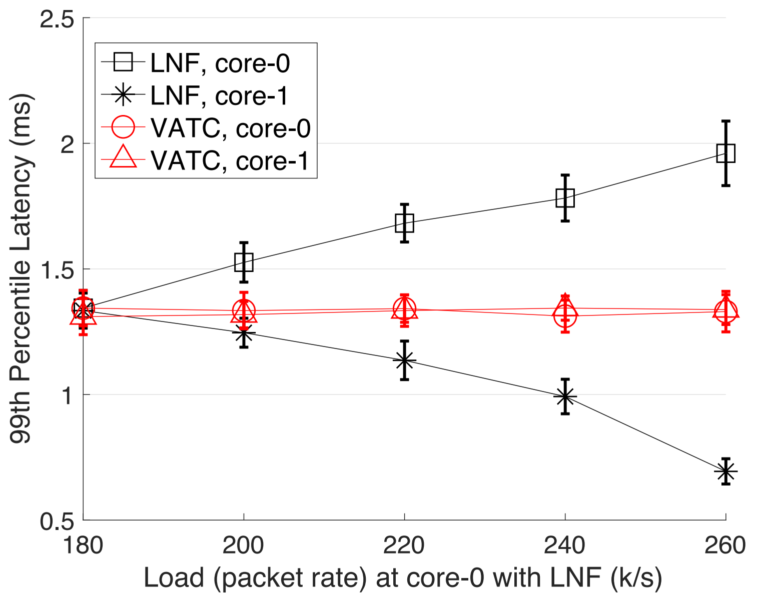


Figure 3.11: Load distribution fairness: LNF and VATC rebalancing

can assign traffic load of 260 k/s to core-0 and the remaining part to core-1 [19]. In Figure 3.11, the x-axis includes different cases (sequences of domain creation) from the best one to the worst one. In each case, we respectively evaluate the 99th percentile latency of the network traffic assigned to core-0 or core-1. Each evaluation is repeated 10 times, so we show both average latency (by markers) and standard deviation (by error bars).

When using the static LNF vif assignment, the network traffic at different cores may show significantly different latency due to unfair load distribution. In the worst case, the latency of the network traffic assigned to core-0 is over 1.5 ms higher than the other one. On the other hand, because of our vif rebalancing mechanism, VATC is able to achieve fair vif assignment such that the two cores show similar performance.

[19]This result occurs if the sequence of domain creation is, e.g., (dom-5, dom-1, dom-6, dom-2, dom-7, dom-3, dom-8, dom-4).

3.5 Related Work

As soft real-time applications are widely deployed in virtualized platforms, protecting latency-sensitive traffic has become an important topic.

The network I/O control in VMware vSphere [25] can reserve I/O resources (e.g. network bandwidth) for business-critical traffic based on user-defined network resource pools [71]. In Windows Server 2012 R2 [26], Hyper-V QoS [16, 20] also provides bandwidth management to network traffic. In environments with network-contention, these can effectively enhance the performance of latency-sensitive VMs. However, because they focus on managing bandwidth, they may not effectively handle the priority inversions caused by CPU contentions as we observed in Xen, which is the focus of VATC. Therefore existing approaches to bandwidth management and VATC are complementary solutions for network- and CPU-contention scenarios, respectively.

KVM [14] is another virtualization platform based on Linux. It creates multiple vhost threads to handle traffic from different guest VMs. However, different vhost threads are not assigned priorities corresponding to the priorities of the VMs. In addition, because vhost threads service traffic as in standard Linux, KVM may experience similar priority inversion problems. For example, the vhost thread servicing real-time traffic can be preempted by threads for non-real-time traffic or softirq handlers.

Xu et al. [78] investigate optimizing the network stack of Xen's dom0 by fragmenting large packets into small ones so that BQL and FQ_CoDel can work more efficiently to reduce queueing delay. In addition to those network stack modifications, Xu et al. [78] also optimize the VCPU scheduler and the network switch to further reduce host-to-host latency in a data center setting. That work, however, does not consider queueing delays in the virtualization layer of dom0, i.e., the netback or vif devices, which can play a significant role.

RT-Xen [75, 51, 76] provides a real-time VCPU scheduling framework recently included in Xen 4.5. Xi et al. [74] develop RTCA, which implements a prioritization-aware packet scheduling in the netback device of dom0. RTCA is able to offer real-time guarantees to *local* inter-domain communications. VATC seeks to extend those guarantees to communications with *remote* hosts.

Another related topic is how to improve guest domains communication performance by allocating additional cores to each domain. Xu et al. [77] improve the I/O performance of a multi-VCPU guest domain by delegating all its I/O processing to a dedicated VCPU. Because of the availability of a dedicated VCPU, the guest domain can process interrupts more efficiently with limited CPU overhead. Similarly, Har'El [42] proposes an efficient and scalable paravirtual I/O system by implementing a fine-grained I/O scheduling and exitless request/reply notification model in KVM. Neither of these two systems seeks to prioritize network traffic with different real-time requirements. Their goal is to improve the average network performance in virtualized hosts (Xen or KVM).

Finally, other work has focused on NICs supporting SR-IOV [19], a pass-through mechanism to bypass the network virtualization layer and dom0 to reduce network latency and have specialized hardware support for network communication. These technologies have been supported by commercial virtualization platforms [8, 64]. In contrast, VATC does not require special hardware support. Radhakrishnan et al. [58] present SENIC, which implements rate limiters and transmit schedulers in hardware. While SENIC is designed to improve the scalability and performance of the low-level network stack, VATC focuses on mitigating priority inversion in the virtualization layer above the native network stack. SENIC and VATC are therefore complementary to each other.

3.6 Conclusion

With the development of ever more powerful and flexible virtualization platforms, distributed soft real-time applications are increasingly deployed in virtualized environments. Those deployments introduce new challenges when it comes to guaranteeing low and predictable latency. This paper evaluates network latency in Xen in the presence of diverse traffic patterns and system configurations, including the use of several existing Linux traffic control mechanisms. Our investigation reveals that some virtualization-related components of Xen can introduce priority inversions in network transmission and reception. Because these limitations involves CPU sharing between network streams, Linux QDisc traffic control mechanisms, which are dedicated to NIC sharing, can not be a feasible solution. Therefore, we propose VATC, a virtualization-aware traffic control framework, which addresses these limitations by a novel network I/O architecture with prioritized and rate limited kernel threads. Additionally, VATC scales to multi-core CPUs and provides a fair load-distribution mechanism. With VATC and QDisc respectively controlling CPU and NIC utilization, we can achieve a complete framework that provides differentiated and isolated traffic processing across all network components in Xen.

Chapter 4

Conclusion

4.1 Open Questions and Future Work

This dissertation seeks to achieve real-time communication at platform and infrastructure layers through SRTM and VATC respectively. Both systems feature rate limited differentiation while preserving scalability. Although the evaluation shows promising real-time communication performance, SRTM and VATC can be extended in the following aspects.

4.1.1 Adaptation to dynamics

SRTM load distributor is triggered at the time a new topic is created. However, decisions made at that one time may become inadequate because of load dynamics at (subsequent) run time.

SRTM currently handles a limited range of load dynamics, namely, shifts in load across publishers within a topic. And even that solution is limited in adjusting sub-token bucket parameters. A more complete solution would handle scenarios where a load shift among publishers is potentially overloading one of the brokers to which the topics was originally assigned. Similarly, such a solution should also handle requests for load increases by a given topic, i.e., a renegotiation of its token bucket parameters. Additionally, the departure of a topic may free up capacity so that another topic currently split across multiple brokers could be consolidated onto fewer brokers, and in the process lower its DRL penalty. To satisfy these demands, SRTM needs a fine-grained adaptor that reacts to all load dynamics while making decisions at limited cost.

In VATC, the need for fine-grained adaptation to dynamics comes up when dom0 has multiple CPU cores. In the current version of VATC, the rebalancing of vif assignments is only triggered at the time a guest domain is created or shutdown. But at normal runtime, traffic variations of guest domains may still result in unfair load distribution across dom0's CPU cores.

One potential solution is relying on irqbalanced [13] to balance notifications (sent from guest domains) across CPU cores at runtime. In Xen, notifications are treated as hardware interrupts, so they have affinity configurations that specify which CPU cores handle them. The irqbalanced is a daemon that can dynamically re-configure the affinity of each interrupt and notification such that interrupt/notification handling is balanced across CPU cores. To complete such a fined-grained adaptation in VATC, whenever a notification's affinity is re-configured, the corresponding vif device has to be re-assigned accordingly. Implementing and comparing this solution to the current vif rebalancing mechanism could be a valuable extension to VATC.

4.1.2 Coordinate two layers

In this dissertation, SRTM and VATC achieve real-time communication *independently* at two layers. A potential extension is developing an interface that coordinate these two systems. For example, in cloud environments, messaging brokers can be deployed in guest domains. Through this interface, SRTM messaging brokers can expose topic-level SLOs and messaging load profiles to dom0 such that VATC can distribute and differentiate network (messaging) flows with more fine-grained granularity.

4.2 Closing Remarks

With the recent emergence of IoT, Industry 4.0 and 5G networks, it becomes a common trend that large-scale and latency-sensitive applications are deployed in cloud environments. As Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS) satisfy the scalability requirements of these cloud applications, the two-layer communication architecture simultaneously introduces a new challenge to the real-time requirements. This dissertation seeks to tackle this challenge and achieve real-time communication at two different layers.

At the platform layer, we build a Scalable Real-Time Messaging (SRTM) middleware (platform) that features (1) latency differentiation, (2) service isolation through rate limiting, and (3) scalability through load distribution among message brokers. The key contribution of SRTM is identifying (distributed) rate limiting's negative impact on latency (DRL penalty). To reduce this penalty, we explore three load distribution principles and develop a novel load distributor which guarantees latency SLOs of topics.

At the infrastructure layer, we develop VATC, a virtualization-aware traffic control framework in Xen virtualized hosts. In this project, we identify the limitations of Xen traffic control mechanisms (Linux queueing disciplines) that leave CPU sharing between network streams unprotected. To mitigate these limitations, VATC provides a novel network I/O architecture that achieves latency (CPU utilization) differentiation through prioritized packet processing. Additionally, VATC features service isolation through rate limiting, and scalability through fair load distribution across multi-core CPUs.

In our evaluation, experimental results show SLO guarantee or significant latency improvement for (soft) real-time applications. This dissertation work therefore makes a promising step towards the real-time communication in cloud environments.

References

[1] Amazon MQ. https://aws.amazon.com/amazon-mq/.

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

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

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

[5] Best Practices for Benchmarking CoDel and FQ CoDel. https://www.bufferbloat.net/projects/codel/wiki/Best_practices_for_benchmarking_Codel_and_FQ_Codel.

[6] CityPulse. http://www.ict-citypulse.eu/page/.

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

[8] Deploying Extremely Latency-Sensitive Applications in VMware vSphere 5.5. http://www.vmware.com/files/pdf/techpaper/latency-sensitive-perf-vsphere55.pdf.

[9] Doorman. https://github.com/youtube/doorman.

[10] Golang Package Debug: SetGCPercent. https://golang.org/pkg/runtime/debug.

[11] GSO: Generic Segmentation Offload. https://lwn.net/Articles/188489/.

[12] iPerf - The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/.

[13] Irqbalance. https://github.com/irqbalance/irqbalance.

[14] KVM: Kernel Based Virtual Machine. http://www.linux-kvm.org/page/Main_Page.

[15] Linux Advanced Routing and Traffic Control. http://www.lartc.org/.

[16] Microsoft Cloud Platform. http://www.microsoft.com/en-us/server-cloud/solutions/virtualization.aspx.

[17] NSQ. http://nsq.io/.

[18] Partitioned Queues and Topics. `https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-partitioning`.

[19] PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology. `http://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html`.

[20] Quality of Service (QoS) Overview. `https://technet.microsoft.com/en-us/library/hh831679.aspx`.

[21] SCATS: The Benchmark in Urban Traffic Control. `http://www.scats.com.au/`.

[22] The Go Programing Language. `https://golang.org/`.

[23] Transform and Protect Your API. `https://docs.microsoft.com/en-us/azure/api-management/transform-api`.

[24] Tyk: Rate Limiting. `https://tyk.io/docs/control-limit-traffic/rate-limiting/`.

[25] VMware vSphere. `http://www.vmware.com/products/vsphere`.

[26] Windows Server 2012 R2. `http://www.microsoft.com/en-us/server-cloud/products/windows-server-2012-r2/`.

[27] XL Network Configuration. `https://xenbits.xen.org/docs/4.2-testing/misc/xl-network-configuration.html`.

[28] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q. Maguire Jr., Panagiotis Papadimitratos, and Marco Chiesa. A high-speed load-balancer design with guaranteed per-connection-consistency. In *NSDI*, 2020.

[29] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *SOSP*, 2003.

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

[31] A. W. Berger. Performance analysis of a rate-control throttle where tokens and jobs queue. *IEEE J. Select Areas Comm.*, 9(2):165–170, February 1991.

[32] A. W. Berger and W. Whitt. The impact of a job buffer in a token-bank rate-control throttle. *Stochastic Models*, 8(4):685–717, 1992.

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

[34] B.B. Brandenburg and J.H. Anderson. On the Implementation of Global Real-time Schedulers. In *RTSS*, 2009.

[35] L. Breslau, S. Jamin, and S. Shenker. Comments on the performance of measurement-based admission control algorithms. In *IEEE INFOCOM*, 2000.

[36] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *SOSP*, 2017.

[37] D. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J.D. Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *NSDI*, 2016.

[38] Andrey Goder, Alexey Spiridonov, and Yin Wang. Bistro: Scheduling data-parallel jobs against live production systems. In *USENIX ATC*, 2015.

[39] 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 *NSDI*, 2015.

[40] Kevin Grünberg and Wolfram Schenck. A case study on benchmarking iot cloud services. In Min Luo and Liang-Jie Zhang, editors, *Cloud Computing – CLOUD 2018*. Springer International Publishing, 2018.

[41] Ajay Gulati, Ganesha Shanmuganathan, Irfan Ahmad, Carl Waldspurger, and Mustafa Uysal. Pesto: online storage performance management in virtualized datacenters. In *ACM SoCC*, 2011.

[42] Nadav Har'El, Abel Gordon, Alex Landau, Muli Ben-Yehuda, Avishay Traeger, and Razya Ladelsky. Efficient and Scalable Paravirtual I/O System. In *USENIX ATC*, 2013.

[43] Herbert, Tom. bql: Byte Queue Limits. `http://lwn.net/Articles/454378/`.

[44] C. N. Hoefer and G. Karagiannis. Taxonomy of cloud computing services. In *IEEE Globecom Workshops*, 2010.

[45] Toke Høiland-Jørgensen. Technical Description of FQ Codel. `https://www.bufferbloat.net/projects/codel/wiki/Technical_description_of_FQ_CoDel`.

[46] Intel. Intel Network Driver Documentation. `https://www.kernel.org/doc/Documentation/networking/e1000e.txt`.

[47] Keon Jang, Justin Sherry, Hitesh Ballani, and Toby Moncaster. Silo: Predictable Message Latency in the Cloud. In *SIGCOMM*, 2015.

[48] Seyyed Ahmad Javadi, Amoghavarsha Suresh, Muhammad Wajahat, and Anshul Gandhi. Scavenger: A black-box batch workload resource manager for improving utilization in cloud environments. In *ACM SoCC*, 2019.

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

[50] 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*, 37(5):17–28, 2007.

[51] Jaewoo Lee, Sisu Xi, Sanjian Chen, Linh T.X. Phan, Christopher Gill, Insup Lee, Chenyang Lu, and Oleg Sokolsky. Realizing Compositional Scheduling Through Virtualization. In *RTAS*, 2012.

[52] C. Li, S. Xi, C. Lu, C. Gill, and R. Guerin. Prioritizing Soft Real-Time Network Traffic in Virtualized Hosts Based on Xen. In *RTAS*, 2015.

[53] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Retro: Targeted resource management in multi-tenant distributed systems. In *NSDI*, 2015.

[54] S. Mao and S.S. Panwar. A survey of envelope processes and their application in quality of service provisioning. *IEEE Communications Surveys*, 8(3), 3rd Quarter 2006.

[55] Nichols, Kathleen and Jacobson, Van. Controlling Queue Delay. `http://queue.acm.org/detail.cfm?id=2209336`.

[56] D. Orfanus, R. Indergaard, G. Prytz, and T. Wien. Ethercat-based platform for distributed control in high-performance industrial applications. In *IEEE Emerging Technologies Factory Automation (ETFA)*, 2013.

[57] Nohhyun Park, Irfan Ahmad, and David J. Lilja. Romano: Autonomous storage management using performance prediction in multi-tenant datacenters. In *ACM SoCC*, 2012.

[58] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. SENIC: Scalable NIC for End-Host Rate Limiting. In *NSDI*, 2014.

[59] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. Cloud Control with Distributed Rate Limiting. In *SIGCOMM*, 2007.

[60] Jun Rao. How to choose the number of topics/partitions in a Kafka cluster. `https://www.confluent.io/blog/`.

[61] Rick Jones. Netperf Manual. http://www.netperf.org/.

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

[63] M. Rubinovitch. The slow server problem. *Journal of Applied Probability*, 22(1):205–213, March 1985.

[64] Schnackenburg, Paul. Hyper-V Deep Dive: Networking Enhancements. http://virtualizationreview.com/articles/2013/03/06/hyper-v-dive-3-network.aspx.

[65] R. Stanojevic and R. Shorten. Generalized Distributed Rate Limiting. In *IEEE IWQoS*, 2009.

[66] R. Stanojevic and R. Shorten. Load balancing vs. distributed rate limiting: An unifying framework for cloud control. In *IEEE ICC*, 2009.

[67] 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 *ACM SoCC*, 2016.

[68] United States Navy. Total Ship Computing Environment(TSCE). peoships.crane.navy.mil/ddx/tsce.htm.

[69] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *EuroSys*, 2015.

[70] Akshat Verma, Gargi Dasgupta, Tapan Kumar Nayak, Pradipta De, and Ravi Kothari. Server workload analysis for power minimization using consolidation. In *USENIX ATC*, 2009.

[71] Vyenkatesh Deshpande. vSphere 5 New Networking Features - Enhanced NIOC. http://blogs.vmware.com/vsphere/2011/08/vsphere-5-new-networking-features-enhanced-nioc.html.

[72] X. Wang, Y. Yao, X. Wang, K. Lu, and Q. Cao. Carpo: Correlation-aware power optimization in data center networks. In *IEEE INFOCOM*, 2012.

[73] Xen Wiki. Xen Common Problems. http://wiki.xen.org/wiki/Xen_Common_Problems.

[74] Sisu Xi, Chong Li, Chenyang Lu, and Christopher Gill. Prioritizing Local Inter-Domain Communication in Xen. In *IWQoS*, 2013.

[75] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. RT-Xen: Towards Real-time Hypervisor Scheduling in Xen. In *EMSOFT*, 2011.

[76] Sisu Xi, Meng Xu, Chenyang Lu, Linh T. X. Phan, Christopher Gill, Oleg Sokolsky, and Insup Lee. Real-Time Multi-Core Virtual Machine Scheduling in Xen. In *EMSOFT*, 2014.

[77] Cong Xu, Sahan Gamage, Hui Lu, Ramana Kompella, and Dongyan Xu. vTurbo: Accelerating Virtual Machine I/O Processing Using Designated Turbo-Sliced Core. In *USENIX ATC*, 2013.

[78] Yunjing Xu, Michael Bailey, Brian Noble, and Farnam Jahanian. Small is Better: Avoiding Latency Traps in Virtualized Data Centers. In *ACM SoCC*, 2013.

[79] Timothy Zhu, Michael A. Kozuch, and Mor Harchol-Balter. WorkloadCompactor: Reducing datacenter cost while providing tail latency SLO guarantees. In *ACM SoCC*, 2017.

Appendix A

Iterative Workload Distribution

When a new topic arrives, SRTM Load Distributor runs the following **LoadDist** algorithm to generate a workload distribution that meets the topic's SLO.

Algorithm LoadDist: New Topic Workload Distribution

```
/*$rate$: total message rate of the new topic*/
function LoadDist($rate$, $broker_{1...N}$)
    for $i = 1$; $i \leq N$; $i++$ do // estimate residual capacity
        $broker_i.rcap$ = $mcap$ - $broker_i.existing\_rate$
    end for
    SORT($broker_{1...N}$) // Sort brokers in decreasing order of $rcap$
    $ua\_rate = rate$ //Initialize unassigned workload
    while $ua\_rate > 0$ do
        $k = \min \mathbf{k}$ subject to $\sum_{i=1}^{k} broker_i.rcap \geq ua\_rate$
        do
            for $i = 1$; $i \leq k$; $i++$ do
                if $broker_i$ has no previous assignment then
                    $quota = \min(broker_i.rcap, rate/k)$
                    if $ua\_rate < quota$ then
                        Migrate ($quota$ - $ua\_rate$) from other brokers
                    end if
                else //$broker_i$ has previous assignment
                    $quota = \min(broker_i.rcap, ua\_rate)$
                end if
                $broker_i \leftarrow$ PUB_SELECT($quota/rate$)
                $broker_i.rcap = broker_i.rcap$ - $quota$
                $ua\_rate = ua\_rate$ - $quota$
            end for
        while $ua\_rate > 0$
        for each $broker$ with new assignment do // Online-Fitting (profiling phase)
            $ex\_rate$ = FIT($broker$)
            if $ex\_rate > 0$ then
                $ua\_rate = ex\_rate$ //Return excess workload
                $broker.rcap = 0$
            end if
        end for
    end while
end function
```

As introduced in Section 2.4.2, **LoadDist** algorithm is a iterative workload distribution. From line 3 to 7, some initiations are done before the iterative process. In the "while loop" from line 8 to 32, the iterative process is conducted. In each iteration, it first (line 9) estimates the minimum number of brokers (k) that provide enough capacity to accommodate the new topic (**Step (1)**). In the next "do-while loop" (line 10 to 24), workload (publishers) is assigned to these k brokers, by taking *max-min* and *correlation-awareness* principles into account (**Step (2)**). After all the workload is assigned to the k brokers, we have to execute the online-fitting (line 25 to 31) to determine whether the current workload distribution is acceptable (**Step (3)**).

Next we introduce **LoadDist** by sequentially going through the main steps. At the beginning, we estimate each broker's residual capacity ($rcap$) at line 4, which is equal to the difference between the $mcap$ (the maximum message rate that a broker is able to accommodate without violating SLO) of a broker and its current existing message rate.

In the "while loop" from line 8 to 34, the iterative process is conducted. To follow the *concentration* principle, we first determine the **minimum** number (k) of brokers which is able to provide enough residual capacity to accommodate the unassigned workload (line 9).

In the next "do-while loop" (line 10 to 24), we assign workload to the k brokers, by taking the *max-min* and *correlation-awareness* principles into account. The assignment is done sequentially across the k brokers (the "for loop" from line 11 to 23). For each broker, we first determine the workload *quota* it should be assigned (line 12 to 19). In this routine, we have to consider two separate conditions. Firstly, if the broker has not been assigned any workload yet (line 12), then its *quota* would be the minimum value of the broker's residual capacity($broker_i.rcap$) and the fair share of the entire workload ($rate/k$). This *quota* ensures each broker gets at least $rate/k$ workload, unless limited by the residual capacity. Within this condition, if unassigned workload is less than the *quota* (line 14), then we have to migrate workload from other brokers (which currently have workload assignment larger than $rate/k$) in order to make this assignment reach the desired *quota* [20]. On the other hand, if the broker already has previous assignment (line 17), then we know *max-min* has been guaranteed. So the broker's *quota* only needs to consider its esidual capacity (accommodate as much workload as possible).

[20]Meanwhile, we make sure migration does not reduce any brokers' assignment below $rate/k$.

After a *quota* is determined, we assign publishers to the broker in a correlation-aware manner (line 20). As introduced in Section 2.4.3, SRTM provides APIs to users such that they can label correlated publishers with an identical **correlated group key**. Based on the keys, SRTM Load Distributor is able to classify publishers into different correlated groups upon the new topic arrives. In the PUB_SELECT function (line 20), we select the same proportion, which is equal to $quota/rate$, of publishers from each correlated group. These publishers are then assigned to $broker_i$. With this assignment approach, *correlation-awareness* principle is achieved (see Appendix A.1 for justification).

After the first pass of the "for loop", we can not guarantee that the sum of the k *quotas* is equal to the ua_rate. For example, if we select two brokers ($k = 2$), with $broker_1.rcap = 25\ k/s$ and $broker_2.rcap = 10\ k/s$, to handle a new topic with $rate = 30\ k/s$ (ua_rate is equal to $rate$ at the first iteration). Then after the first pass of the "for loop", $broker_1.quota = 15\ k/s$ and $broker_2.quota = 10\ k/s$, with $5\ k/s$ remaining ($ua\_rate > 0$). This is the reason that we need the outer "do-while" loop. With the extra check at line 24, we know whether the "for loop" has to be executed again, in which remaining workload is assigned.

After the assignments to the k brokers complete, we have to go through the online-fitting. During this step, each broker with new assignment has to run a FIT function. This function tests whether the newly-assigned workload results in SLO violation. If that is the case, then the function detects the maximum subset of the (newly-assigned) workload the broker is able to accommodate, and reports the excess workload to SRTM Load Distributor.

The FIT function is just a iterative binary search procedure. In each iteration, we enable a subset of the (newly-assigned) publishers (while disabling the complementary set) and measure the resulting latency. If the result exceeds the latency threshold θ (defined in SLO contract), we shrink the subset by half. Otherwise, we enlarge the subset. Assuming m publishers are assigned to a broker, then this binary search ends within at most $\log_2 m$ steps.

In order to shorten the depth of this binary search, we use $[0.8*\theta, \theta]$ as an "extended" latency threshold. This configuration relies on a fact that latency normally grows at increasing speed with load. Once latency has already approached the threshold, i.e., within $[0.8 * \theta, \theta]$, then accommodating even a little amount of extra load is very likely to dramatically increase latency, thereby violating SLO. So during the binary search, once finding a subset of publishers with resulting latency within this "extended" threshold, the binary search stops, and the complementary set of publishers (as the excess workload) are returned to SRTM Load Distributor.

After online-fitting, if there is no excess workload reported, then an adequate distribution has been made. Otherwise, we have to distribute the excess workload in the next iteration with a different k (re-calculated at line 9).

A.1 Achieving *correlation-awareness*

We leverage the PUB_SELECT function (line 20) to achieve a correlation-aware publisher assignment. In this function, we select the same proportion, which is equal to $quota/rate$, of publishers from each correlated group. These publishers are then assigned to the desired broker.

With this approach, we claim: for each assignment, the total workload of the selected publishers is equal to the *quota*, and message burst is decreased with the same proportion as the workload split. The proof is as follows:

Proof. Assume a topic has n publishers, and each publisher has 1 unit of workload. All the publishers belong to m correlated groups with sizes $= \{n_1, ..., n_m\}$ subject to $\sum_{i=1}^{m} n_i = n$. Accordingly, the message arrival process of the new topic can be formed by m independent bursts $= \{n_1, ..., n_m\}$. Given $broker_j$ with $quota = q_j$, then PUB_SELECT function selects publishers from each correlated group with proportion $\frac{q_j}{n}$, and constitutes a set of publishers P. The total workload of P is of the form

$$L(P) = \sum_{i=1}^{m} \frac{q_j}{n} * n_i = \frac{q_j}{n} * \sum_{i=1}^{m} n_i = \frac{q_j}{n} * n = q_j$$

, which means the PUB_SELECT function selects the right amount of workload. Besides, the message arrival process of P also have m bursts $= \{\frac{q_j}{n} * n_1, ..., \frac{q_j}{n} * n_m\}$. Hence, each original burst is split by $\frac{q_j}{n}$, which is the same proportion as the workload split, i.e., $\frac{L(P)}{n}$. $\square$

With this approach, each assignment splits the burst in a manner that parallels the decrease of the respective token bucket size, which realizes the *correlation-awareness* principle.

Real-Time Communication, Li, Ph.D. 2020