Utility-driven Performance Management over a Shared Resource Pool

Garyfallia Kourou

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Associate Professor *Kostas Magoutis*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).

University of Crete
Computer Science Department

Utility-driven Performance Management over a Shared Resource Pool

Thesis submitted by
Garyfallia Kourou
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: Γαρυφαλλια Κουρου
Garyfallia Kourou

Committee approvals: KONSTANTINOS MAGOUTIS 07/06/2023 22:14
Kostas Magoutis
Associate Professor, Thesis Supervisor

ANGELOS BILAS ANGELOS BILAS Jun 8 2023 8:37 AM
Angelos Bilas
Professor, Committee Member

DIMITRIOS PLEXOUSAKIS Digitally signed by DIMITRIOS PLEXOUSAKIS Date: 2023.06.08 11:05:50 +03'00'
Dimitris Plexousakis
Professor, Committee Member

Departmental approval: Polyvios Pratikakis Digitally signed by Polyvios Pratikakis Date: 2023.06.15 11:09:53 +03'00'
Polyvios Pratikakis
Associate Professor, Director of Graduate Studies

Heraklion, March 2023

Utility-driven performance management over a shared resource pool

Abstract

Autonomic computing systems aim for self-adaptation and self-management, typically using a decision-making process to comply with user-defined service goals. In this thesis, we propose a scheme to effectively provide utility-driven fair sharing of resources with explicit quality-of-service (QoS) targets. We describe Sprinkler, a two-level architecture for achieving performance QoS guarantees for latency-sensitive clients over shared resource pools. Sprinkler utilizes a higher-level controller to achieve fair sharing of aggregate system throughput while a lower-level scheduler simultaneously achieves explicit per-client latency targets in concurrent access to a shared pool of resources. The lower-level scheduler isolates each client from others that may exceed their load specifications (average throughput and/or maximum burst size) as decided by the upper-level controller. A distinctive feature of Sprinkler is that controller allocations are advisory, namely a client may decide to exceed them hoping to leverage excess capacity from reduced demand of other clients. Such speculative, non-complying clients can benefit from spare capacity, if it is indeed available, but only hurt themselves if not, as Sprinkler protects compliant clients from violating their latency targets under overload conditions. Sprinkler makes clients aware of their latency metrics at any point in time, so that they can decide if they can afford risking latency violations. We implement a prototype of Sprinkler and demonstrate its effectiveness in fairly regulating per-client throughput while providing latency guarantees in experiments with synthetically generated workloads under workload variations.

Διαχείριση απόδοσης με βάση συναρτήσεις χρησιμότητας πάνω από κοινόχρηστους πόρους

Περίληψη

Τα αυτόνομα υπολογιστικά συστήματα αποσκοπούν στην αυτόματη διαχείριση και προσαρμογή των λειτουργιών τους, μέσω μιας διαδικασίας λήψης αποφάσεων για την επίτευξη στόχων επιπέδου υπηρεσίας πελατών. Στην παρούσα διατριβή προτείνεται ένας μηχανισμός δίκαιης μοιρασιάς πόρων συστήματος, με βάση συναρτήσεις χρησιμότητας, για την επίτευξη στόχων απόδοσης κατά την ταυτόχρονη προσπέλαση κοινόχρηστων πόρων. Ο μηχανισμός αυτός, τον οποίο ονομάζουμε Sprinkler, ακολουθεί μια αρχιτεκτονική δύο επιπέδων. Στο υψηλότερο επίπεδο, το Sprinkler χρησιμοποιεί έναν ελεγκτή για να επιτύχει δίκαιη κατανομή της συνολικής ρυθμοαπόδοσης του συστήματος μέσω εκχωρήσεων ανά πελάτη, ενώ ένας προγραμματιστής χαμηλότερου επιπέδου επιτυγχάνει ταυτόχρονα στόχους καθυστέρησης ανά πελάτη. Ο προγραμματιστής χαμηλότερου επιπέδου απομονώνει κάθε πελάτη από άλλους που μπορεί να υπερβαίνουν τις προδιαγραφές φόρτου τους (μέση εκχώρηση ρυθμοαπόδοσης ή μέγιστο μέγεθος έκρηξης (burst size)) όπως ορίζεται από τον ελεγκτή ανώτερου επιπέδου. Ένα ιδιαίτερο γνώρισμα του Sprinkler είναι ότι οι εκχωρήσεις είναι συμβουλευτικές, δηλαδή ένας πελάτης μπορεί να τις υπερβεί ελπίζοντας να αξιοποιήσει πλεονάζουσα χωρητικότητα. Τέτοιες διερευνητικές πολιτικές μπορούν να ωφελήσουν τους πελάτες αν υπάρχει πλεονάζουσα χωρητικότητα, χωρίς να βλάπτουν άλλους πελάτες καθώς το Sprinkler προστατεύει εφαρμογές που τηρούν τις εκχωρήσεις τους από την παραβίαση στόχων καθυστέρησης υπό συνθήκες υπερφόρτωσης. Το Sprinkler περιοδικά ενημερώνει τις εφαρμογές για τις μετρήσεις χρόνου απόκρισής τους, ώστε να μπορούν να αποφασίσουν εάν έχουν περιθώριο παραβιάσεων με βάση τον στόχο ποιότητας υπηρεσιών τους. Υλοποιήσαμε ένα πρωτότυπο του Sprinkler και το αξιολογήσαμε σε πειράματα με συνθετικά παραγόμενο φόρτο εργασίας, υποδεικνύοντας την αποτελεσματικότητά του στη δίκαιη ρύθμιση της απόδοσης ανά πελάτη, παρέχοντας ταυτόχρονα εγγυήσεις καθυστέρησης υπό διακυμάνσεις φόρτου εργασίας.

Acknowledgements

First and foremost, I would like to thank my supervisor Prof. Kostas Magoutis for his constant support and valuable guidance he has provided throughout my thesis. The way he encouraged me helped me expand my skills and knowledge allowing me to grow professionally.

I would also like to express my gratitude to my co-supervisor Dr. Marina Bitsaki for her guidance and assistance that helped shape my thesis providing me with meaningful piece of insight.

Additionally, I would like to thank my collaborator Giorgos Kelantonakis for his help and meaningful contribution in completing this work. He was always willing to share his knowledge and assistance in any way possible throughout the research project.

I would also like to thank my committee members - Angelos Bilas and Dimitris Plexousakis - for providing helpful feedback on my thesis.

Furthermore, I would like to thankfully acknowledge the support of funding by the Greek Research Technology Development and Innovation Action "RESEARCH-CREATE- INNOVATE", Operational Programme on Competitiveness, Entrepreneurship and Innovation (ΕΠΑνΕΚ) 2014-2020, Grant T2EΔK-02848 (SmartCityBus).

Finally, I would like to thank my family for their love and support during this process. Without their encouragement and motivation, it would be impossible for me to complete my studies.

στους γονείς και τα αδέλφια μου

Contents

Table of Contents **1**

1 Introduction **3**

2 Background **5**
1 Goal-oriented autonomic systems 5
2 Utility functions 6
3 Quality of service 8
4 Serverless computing 8

3 Related work **11**

4 Experience with weighted fair sharing algorithms **15**

5 Fair throughput allocation with latency guarantees **21**
1 Design 21
2 Implementation 24

6 Evaluation **27**
1 Performance isolation and use of excess capacity 27
2 Fair throughput regulation 29
3 Adaptation to workload variations 30
4 Application-specific rate-control policy 31

7 Conclusions **35**

8 Future work **37**

Bibliography **39**

Chapter 1

Introduction

Shared resource pools, such as consolidated storage arrays, large core-count multiprocessors, etc., are prevalent in data-center environments due to their simpler management and statistical multiplexing benefits. Concurrent access to such resource pools by multiple applications raises the issue of how to ensure fair sharing of resources, isolating well-behaving applications from others that may be hoarding resources, and differentiating between applications under resource shortage. Applications often express their performance goals either via explicit metrics, such as a specific level of throughput and/or latency, or via utility functions expressing the value assigned by the application owner to different performance levels. Another approach to fair resource allocation has focused to providing shares (weights) of aggregate system throughput to specific clients.

Several fair sharing mechanisms offering weighted bandwidth allocation have been proposed over the years [5, 4, 6, 9, 24]. Although such systems often bound latency as well [25], they cannot be used to *independently* control the bandwidth and latency of clients. These algorithms offer one parameter (weight) to adjust both: lowering latency can only be done by increasing the bandwidth allocation of a client. As such they cannot meet both constraints independently.

In this thesis we describe Sprinkler, a system that aims to achieve explicit latency targets for applications and fair sharing of the aggregate throughput over shared resources as allowed by available capacity. Sprinkler achieves this by interleaving a throughput-regulating controller over a state-of-the-art latency-oriented scheduler [8]. The aim of Sprinkler is to achieve high system utilization and high throughput allocation for applications as far as resources allow, in a fair manner [2], while guaranteeing explicit latency targets for each application. Sprinkler additionally isolates well-behaved applications from others that exceed their average throughput allocation as set by the upper-level controller.

A distinctive feature of Sprinkler is that controller allocations are *advisory*, namely an application may decide to exceed them hoping to leverage a possible simultaneous drop in demand by other applications, thus increasing overall

system efficiency. Such a feature is not available with systems that *explicitly* rate-control sources to enforce latency targets [7, 16]. The latter are typically not work-conserving (i.e., an application experiencing higher throughput demand may not benefit from a simultaneous reduction in throughput use by another application). In Sprinkler, if no excess capacity is available, such speculative, non-complying applications can only hurt themselves, as Sprinkler protects compliant applications from violating their latency targets under overload conditions.

This thesis proposes a scheme to effectively provide utility-driven fair sharing of resources with explicit quality-of-service (QoS) targets. We implement a prototype of Sprinkler and demonstrate its effectiveness in fairly regulating per-client throughput while providing latency guarantees in experiments with synthetically generated workloads under workload variations.

The contributions of this thesis are:

- A novel two-level QoS architecture (Sprinkler) for fair sharing of aggregate throughput among latency-sensitive applications over a shared pool of resources, providing each application with an advisory rate allocation for which a latency-target is guaranteed, while allowing it to control its request-rate to explore higher efficiency
- An evaluation of a prototype implementation of Sprinkler with synthetically generated request streams, exhibiting its key benefits, namely fair throughput allocation, latency guarantees, isolation under overload, and application-specific rate control for higher efficiency

Chapter 2

Background

In this section we introduce basic principles of resource allocation and decision-making mechanisms.

1 Goal-oriented autonomic systems

Autonomic computing, an architecture initiated by IBM, [3, 21, 30] is the capability of a system to automatically adapt itself to changing environmental conditions in order to achieve *system goals*, such as performance goals. This architecture consists of a number of interacting components where each is responsible for providing or consuming useful information to achieve self-management. Specifically, a self-adaptive system is based on the architecture called MAPE, a feedback loop model, which includes monitoring, analysis, planning and execution functions. It operates with the guidance of a central Controller that makes adaptive decisions using a set of *policies*.

Policies are generally organized in a top-down hierarchy to describe the decision-making process by specifying a set of rules or/and constraints that decisions must be based on to achieve the desired outcome. The goal is to control system behavior by mapping the rules and constraints into specific system configuration. Policies are divided into two types to describe system goals, high-level and low-level policies. High-level policies define how the system objective is met by guiding decision-making. They specify user goals by representing the quality of service (QoS) they are willing to experience. On the other hand, low-level policies define how the system should work by providing information control to meet required specifications and specific compliance objectives. That is, low-level policies specify how the users should interact with the system in order to satisfy high-level goals.

To effectively achieve the goals of the system, the feedback loop starts collecting information of relevant data (performance metrics that can be associated with specified-goals) that reflect the current state of the system. The Controller then analyzes the collected data using the specified policies to detect the state the system is in and if something needs to change. Next, based on policies and the

knowledge about what is happening in the environment, the Controller decides how to adapt the system in order to reach the desirable state by implementing the decision by configuring the system environment.

2 Utility functions

The utility function is one of the most widely used methods in economic theory for measuring the happiness or satisfaction that someone gets from using (or consuming) a product or service. Specifically, the utility function strategy is used to determine which level of satisfaction in a group of possible outcomes is most preferred by associating a single value or score to each.

Prior works [10, 19, 22, 23, 26], have focused on utility-based approaches for achieving self-optimization in autonomic computing systems. Utility functions are used as QoS indicators to help decision makers make the best decision in order to determine the most valuable feasible state of the system. Utility functions can be defined by the users of the service system to describe the desired specification levels of system behavior. The idea behind expressing system requirements using utility functions is to assess the objectives of a system from a business perspective. Since each performance metric (throughput, latency, etc.) has an effect on system QoS, a utility function can be associated with a system metric to analyze the impact it can have on system performance either from an economic point of view (e.g. revenue and costs) or in terms of level of satisfaction (e.g. specific level of throughput and/or latency).

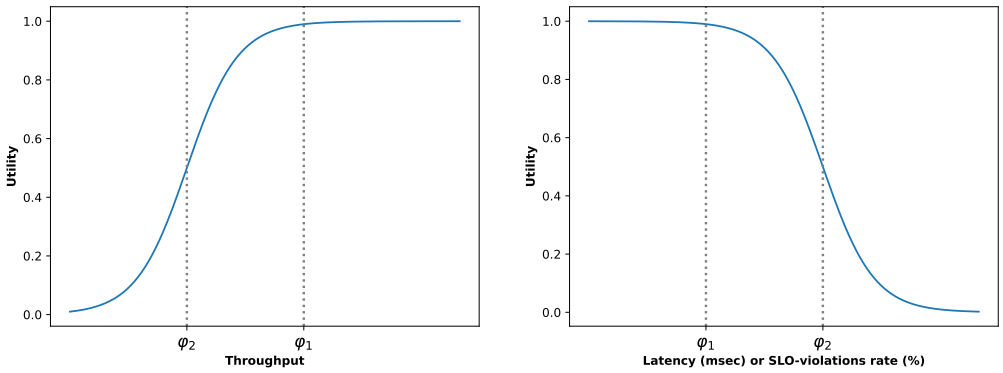
Strunk et al. [22], use the utility functions as return on investment (ROI) metric to automatically provisioning a storage system. They present a utility-driven provisioning tool that evaluates a storage system configuration by associating it with a utility value to describe the suitability of the configuration according to the objectives of administrator. Gupta et al. [10], present a dynamic multi-tier pricing scheme for optimal resource allocation in serverless computing, taking into account delay-sensitive characteristics. Walsh et al. [26], use utility functions to develop a two-level architecture for dynamically allocating servers within a data center with multiple transaction classes in response to changes in average response time. Tesauro et al. [23], use reinforcement learning for resource allocation among multiple applications based on optimizing the sum of utility for each application. Różańska and Horn [19], propose a utility model for application deployment configuration management in cloud computing. To capture application client preferences, they combine a two modelled utility functions into an overall to analyze both the cost and the user satisfaction.

Basic concept and mathematical notation

Utility functions are expressed as a function of random variables $X_i, i = 1, 2, ..N$, representing service characteristics. Random variables take on multiple values

corresponding to the specified preferences of consumers. The utility function computes the utility of obtaining a preference and the utility gained from a preference x is denoted with $f(X_i = x)$.

Generally, there are several common utility functions used to model how individuals value alternative preferences, such as linear, exponential, logarithmic. In this thesis, we use sigmoid functions [19, 26] to describe the desirable properties of a service system. Specifically, in (Fig. 2.1) we illustrate two types of sigmoid functions that we use to describe client-specific level of throughput, latency and SLO-violations.



(a) S-shaped sigmoid function (b) Reversed S-shaped sigmoid function

Figure 2.1: Sigmoid functions as utility functions

We use S-shaped sigmoid function (Fig. 2.1 (a)) to express client satisfaction for a specific level of throughput, implying the higher the value, the higher the client satisfaction.

$$f_1(X_i/\phi_1, \phi_2) = 1 - [1 + \mathrm{e}^{(\mathrm{X_i} - \phi 2)\cdot \ln \frac{1-\epsilon}{\epsilon} \cdot \frac{1}{\phi_1 - \phi_2}}]^{-1} \tag{2.1}$$

We use the Reversed S-shaped sigmoid function (Fig. 2.1 (b)) to describe client satisfaction for a specific level of latency or SLO-violations rate, indicating a preference in low values.

$$f_2(X_i/\phi_1, \phi_2) = [1 + \mathrm{e}^{(\mathrm{X_i} - \phi 2)\cdot \ln \frac{\epsilon}{1-\epsilon} \cdot \frac{1}{\phi_1 - \phi_2}}]^{-1} \tag{2.2}$$

Sigmoid function is a mathematical function that can takes any real number as input and maps it to a value between 0 and 1. The sigmoid function has an S-shaped characteristic whose shape is determined by two parameters, ϕ_1 and ϕ_2. Shape parameters determine the slope of the function and indicate what range of preferences maps close to 0 or 1.

In (Fig. 2.1), we observe that a preference value equal to zero maps to zero, a preference value equal to ϕ_2 maps to 0.5, and a preference value greater than or equal ϕ_1 maps to 1.

3 Quality of service

Quality of Service (QoS) is a technique used for managing the service level experienced by a workflow with specified requirements. QoS is a measurement that helps ensure a specific service level by allocating resources among workloads. Aspects of the service can include multiple performance metrics, such as response time, throughput etc., availability, reliability. In this thesis, we focus on performance metrics, throughput and latency.

The measurement of quality of service is an essential process as depends entirely on the context and constraints defined by a client. The actual steps required to improve service quality depend on monitoring and control period. To improve understanding of client requirements, Controller periodically should take control actions to manage the service received by each client. Generally, collecting real-time monitoring data helps a control manager immediately evaluate and react to current events. Thus, the amount of time of monitoring is an important part of effective management. Another key part of control management is the analysis and use of information during implementation. Therefore, an attention should be paid to statistical methods in the decision-making process.

In this thesis, we use exponentially weighted moving average (EWMA), a widely used technical analysis to monitor attribute data using the entire history of data. Specifically, EWMA weights samples to indicate how important recent samples are. In Equation 2.3 we denote the current time as t, the current observation as X_i and the weighted factor by α.

$$EWMA_t = \alpha \cdot X_i + (1 - \alpha) \cdot EWMA_{t-1} \tag{2.3}$$

In this thesis, we assume that Controller periodically (every 1 second) receives metric observations by monitor and calculates the average of the metric values based on the exponentially weighted moving average (EWMA). Until reaching the end of a control period, controller just records the observations and calculates the EWMA. To calculate the EWMA we set the weight α equal to 0.125. Upon reaching the end of a control period, it makes changes based on a specified fairness criterion. Specifically, at the end of each control period (every 10 seconds), the controller changes the utilities taking into account the utility function of each client, which is estimated based on EWMA measurements of throughput and latency.

4 Serverless computing

Serverless computing is a cloud computing service which offers on-demand resources. Serverless computing is a popular technology that simplifies the process of deployment and resource management. It only allocates resources when the applications are used and charges the clients for the time the applications run.

Serverless computing differs from the traditional concept of cloud computing in which clients rent and configure a virtual machine environment based on the needs

of their application [13]. It provides backend services and allows programmers to write and deploy code in a high-level language without having to worry about server management. Instead, cloud providers are responsible to manage, maintain and provision the infrastructure required to execute code. This architecture enables developers to focus more on the code by adding features to their application and better configuring its functionality. Serverless computing, also referred to as FaaS (Functions as a Service), uses an event-driven computing execution model where applications are composed by decoupled functions. Each function corresponds to a specific task and when an associated event (or HTTP request) is occurred, it is triggered. Then, a container instance is provided to the invoked function to be executed by the cloud provider.

Despite all the advantages serverless computing provides, such as on-demand scalability, no charge for idle resources etc., there are no resource management algorithms available to provide QoS guarantees. In this thesis, we study QoS techniques to provide resource isolation and service differentiation requirements. In this work, we also present a prototype of our QoS approach that could be adopted in a serverless computing platform to efficiently manage a container pool.

Chapter 3

Related work

Recent work has surveyed general decision-making mechanisms for storage QoS [15], including approaches to fair sharing of shared storage systems. Here we discuss previous research works that have been proposed in this context and that aim for fair sharing and/or explicit QoS guarantees (latency target, throughput reservations, etc.).

mClock [9] is an algorithm for I/O resource allocation in a hypervisor that supports proportional-share fairness subject to minimum reservations and maximum limits on the I/O allocations. mClock logically interleaves a constraint-based scheduler and a weight-based scheduler in a fine-grained manner. The mClock scheduler alternates between phases during which one of these schedulers is active to maintain the desired allocation. An important difference of Sprinkler from mClock is that Sprinkler aims to enforce per-application latency targets by overlaying its utility-driven throughput-regulating Controller over the pClock arrival-curve scheduler described next, whereas latency is not an explicit goal in mClock. Sprinkler does not support throughput limits.

pClock [8] is an arrival-curve based algorithm that achieves latency targets for individual workloads taking into account throughput and burst specifications. In pClock, application requirements are represented in terms of the average throughput, desired latency and maximum burst size. pClock isolates workloads to protect from applications exceeding their specifications and allows use of spare capacity without penalizing applications for prior use of such capacity. pClock decouples latency and throughput requirements but does not support reservations and limits on application throughput. In this work, we use pClock as a lower-level scheduler, working in conjunction with a higher-level Controller that provides application throughput specifications as input to pClock. In this way, pClock acts as an isolation mechanism for Sprinkler, who overlays utility-driven fair throughput sharing over it.

PARDA [7] enforces proportional-share fairness among distributed hosts accessing a storage array, without assuming support from the array itself. PARDA uses latency measurements to detect overload, and adjusts issue-queue lengths to

provide fairness, similar to aspects of flow control in FAST TCP [28] with local I/O scheduling at each host using SFQ(D) [12]. PARDA can provide differential quality of service for unmodified virtual machines while maintaining high efficiency. Sprinkler is similar to PARDA in that its controller applies a flow control scheme to adjust demand to capacity. Sprinkler differs from PARDA in that it enforces *per-flow* latency targets (rather than an aggregate average) via its lower-level pClock scheduler. In addition, Sprinkler's rate control is advisory to give speculative applications the ability to pursue higher throughput at the risk of violating their latency targets without hurting other (compliant) applications.

PSLO [16] defines a framework to support 99.9th percentile latency and throughput service-level objectives (SLOs) under consolidated VM environment by precisely coordinating the level of IO concurrency and arrival rate for each VM issue queue in a single consolidated platform (e.g., a Xen hypervisor). PSLO features a control loop to regulate issue rates so as to utilize available IO capacity while achieving stringent latency targets. PSLO leverages detailed knowledge of latency history for each flow (VM) to determine whether higher rates are possible in that flow. Sprinkler follows the same principle, except that it exposes it as an application-specific policy, rather than applying it across all issue queues as in the case of PSLO. In that sense, Sprinkler assumes a gray-box (providing certain information to facilitate application-level control [1]), rather than a black box model, for the underlying resource management platform.

Pisces [20] achieves per-tenant weighted fair sharing of system resources across the entire shared service, even when partitions belonging to different tenants are co-located and when demand for different partitions is skewed or time-varying. Pisces decomposes the fair sharing problem into four complementary mechanisms— partition placement, weight allocation, replica selection, and weighted fair queuing— operating on different time-scales to provide system-wide max-min fairness. The Pisces storage prototype achieves fair sharing, strong performance isolation, and robustness to skew and shifts in tenant demand.

IOFlow [24] is an architecture that uses a logically centralized control plane to enable high-level flow policies for differentiated shared access to storage resources. IOFlow adds a queuing abstraction at data-plane stages and exposes this to the controller. The controller can then translate policies into queuing rules at individual stages. It can also choose among multiple stages for policy enforcement.

Façade [18] is a virtual store controller that interposes between hosts and storage devices in the network, and throttles individual I/O requests from multiple clients so that devices do not saturate. Façade satisfies performance objectives while making efficient use of the storage resources-even in the presence of failures and bursty workloads with stringent performance requirements. Sprinkler has similar goals to Façade but differs in its approach in that it layers its throughput controller over a latency-oriented QoS scheduler, whereas Façade intermixes its ad-hoc throughput controller with an earliest-deadline first (EDF) scheduling policy. Sprinkler improves over Façade by incorporating a better approach to latency-aware scheduling (pClock [8], whose key features are described earlier), and a

cleaner separation between the throughput control and latency-aware scheduling layers. In contrast to Façade, Sprinkler facilitates throughput control for individual flows, allowing better handling of service differentiation and fairness issues.

Triage [14] proposes an online feedback loop with an adaptive controller that throttles storage access requests to ensure that the available system throughput is shared among workloads according to their throughput goals and their relative importance for isolation and differentiation. While Triage generally results to reasonable average latency, it does not explicitly target per-application latency targets, one of the goals in Sprinkler.

LaSS [27] uses a fair-share allocation approach to guarantee a minimum of allocated resources to each function in the presence of overload in a serverless function-as-a-service (FaaS) execution environment. It utilizes resource reclamation methods based on container deflation and termination to reassign resources from over-provisioned functions to under-provisioned ones.

Weighted fair sharing schemes include WFQ [5] and YFQ [4] among others. SFQ [6] is a weighted fair sharing algorithm that is computationally efficient and achieves fairness regardless of variation in server capacity. Such schemes have been used to implement proportional-share fairness, typically with fixed share proportions or *weights* (e.g., 1:2:4). In a more general context, fairness refers to resource allocation where the utilities (a quantitative level of client goals and satisfaction) achieved by different clients conform to criteria such as utilitarian, max-min or proportional fairness [2]. In this work we use a simple fairness scheme but plan to experiment with other notions of utility-based fairness (max-min, proportional) in future work.

Chapter 4

Experience with weighted fair sharing algorithms

In this section, we explore weighted fair sharing algorithms related to QoS-based resource allocation that provide fairness proportional to the weights. The concept of weighted fair sharing algorithms is to achieve desired quality of service goals providing high throughput and low latency. Such scheduling algorithms, associate two tags a start tag and a finish tag with each request and schedule requests in the increasing order of the start tags or finish tags of the requests using virtual clocks. These algorithms provide an abstraction of having its own dedicated server with a guaranteed minimum level of performance.

We implement YFQ [4] which is a time-sharing algorithm that uses client-specified weights as resource reservations to provide QoS. It assigns start tag and finish tags with each reservation and schedule requests in the increasing order of the finish tags of the reservations. Specifically, we consider a public pool and assume that each workload is associated with a weight representing the amount of resources can be consumed. We use weights as control knobs to achieve performance targets (throughput and latency) by changing weight settings. In case of a saturated system, we aim to provide an optimal solution based on fairness rules (constraints) to satisfy the performance targets.

Evaluation

The following experiments are conducted using the *max-min fair allocation* [2], a fair sharing technique that allocates resources first to the client with the smallest demand (expected load), and subject to that the second smallest, and so on. In these experiments we assume latency targets to study the behavior of the system in terms of stability under different weight settings and input load. If latency exceeds the desired latency bound (0.5 sec), then clients become dissatisfied, which means zero utility value. Below, we present the results obtained with two and three clients.

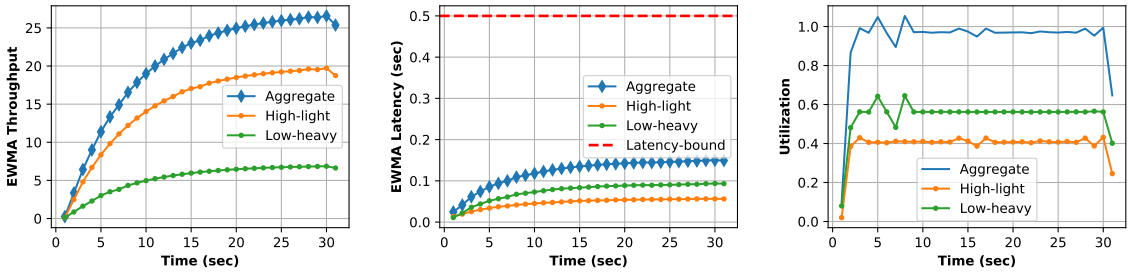
Case 1: Feasible system close to capacity - Unequal load

In this example, we define unequal load and different weights between clients under capacity. We set a weight of 0.4 for the high-light client and 0.6 to the low-heavy. Looking at Table 4.1 we predict a feasible system with 96% of server utilization, and we expect bounded latency for both clients.

Client		Weight	Arrival rate	Service demand	Latency target	Expected utilization
Color	Label	w_i	r_i (req/sec)	D_i (msec)	l_i (msec)	$r_i \cdot D_i$ (%)
Orange	High-light	0.4	20	20	500	40
Green	Low-heavy	0.6	7	80	500	56

Table 4.1: Workload specifications, expected utilization in units of processing time per wall-clock time

In Fig. 4.1, we depict the results of this experiment, which is validated as predicted. Each client consumes resources according to their demands (the measured utilization agrees with the expected, as shown in Fig. 4.1c).



(a) EWMA Throughput (b) EWMA Latency (c) Utilization

Figure 4.1: Feasible system close to capacity - Unequal load

Case 2: Feasible system close to capacity - Equal load

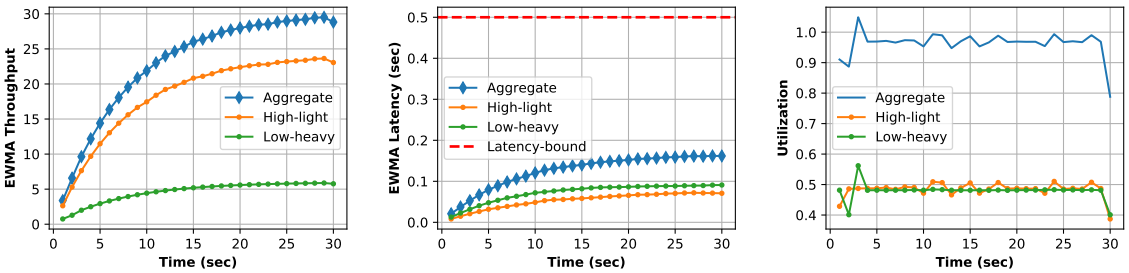
The following example differs from the previous one (exp. 4) in terms of load, however stills under capacity. We define equal load while keeping the same weight settings.

Client		Weight	Arrival rate	Service demand	Latency target	Expected utilization
Color	Label	w_i	r_i (req/sec)	D_i (msec)	l_i (msec)	$r_i \cdot D_i$ (%)
Orange	High-light	0.4	24	20	500	48
Green	Low-heavy	0.6	6	80	500	48

Table 4.2: Workload specifications, expected utilization in units of processing time per wall-clock time

As it can be observed in Fig. 4.2, system is feasible (clients are satisfied, both achieve stable latency). In Fig. 4.2c we observe that the resource demand of the

low-heavy client is less than the amount of resources could be consumed based on the weights provided, as a result the high-light client benefits from the excess capacity. This occurs as the result of needs of clients rather of control.



(a) EWMA Throughput (b) EWMA Latency (c) Utilization

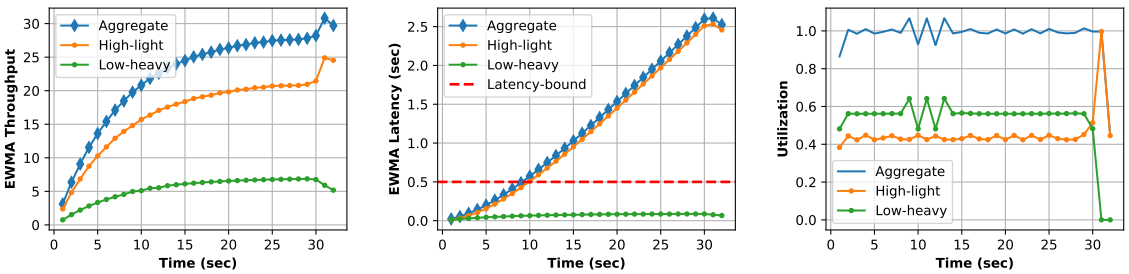
Figure 4.2: Feasible system close to capacity - Equal load

Case 3: Infeasible system at capacity - Unequal load

In this example, we study a system with unequal load with the same weight settings, but above capacity. The expected server utilization is 104%, and we predict an infeasible system as demand exceeds capacity.

Client		Weight	Arrival rate	Service demand	Latency target	Expected utilization
Color	Label	w_i	r_i (req/sec)	D_i (msec)	l_i (msec)	$r_i \cdot D_i$ (%)
Orange	High-light	0.4	24	20	500	48
Green	Low-heavy	0.6	7	80	500	56

Table 4.3: Workload specifications, expected utilization in units of processing time per wall-clock time



(a) EWMA Throughput (b) EWMA Latency (c) Utilization

Figure 4.3: Infeasible system at capacity - Unequal load

In Fig. 4.3 depicts what we predicted, the system is infeasible and only provides bounded latency to the low-heavy client, which is the client with the higher

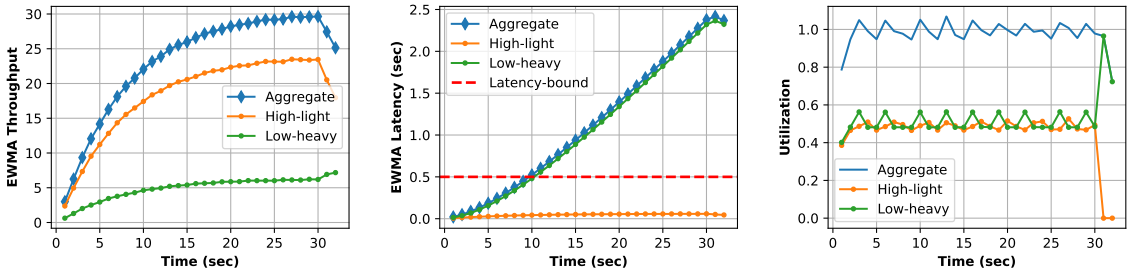
demand. We observe an inefficient allocation of resources as the system is unable to successfully provide a guaranteed level to the weaker client (high-light). This implies that fairness is needed in order to protect the client with the lower demand.

Case 4: Infeasible system at capacity - Equal load

In the following experiment, we are going to protect the weak client and ensure a fair share of resources based on max-min principles [2]. We use the same workload specification with the previous example (exp. 4) except weight settings. To successfully protect the weak client, we set a weight of 0.5 for each client (Table 4.4).

Client		Weight	Arrival rate	Service demand	Latency target	Expected utilization
Color	Label	w_i	r_i (req/sec)	D_i (msec)	l_i (msec)	$r_i \cdot D_i$ (%)
Orange	High-light	0.5	24	20	500	48
Green	Low-heavy	0.5	7	80	500	56

Table 4.4: Workload specifications, expected utilization in units of processing time per wall-clock time



(a) EWMA Throughput (b) EWMA Latency (c) Utilization

Figure 4.4: Infeasible system at capacity - Equal load

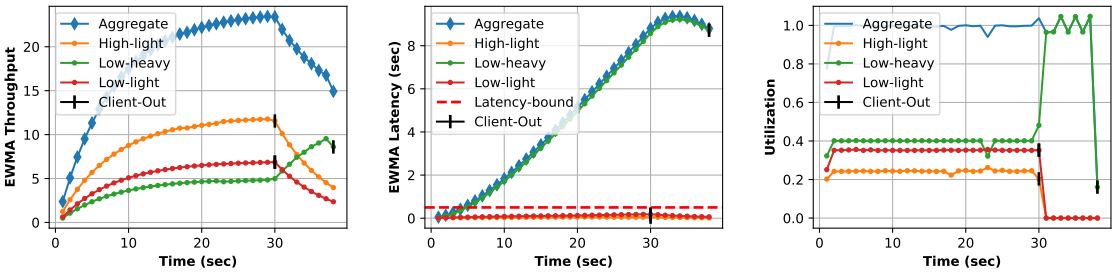
In Fig. 4.4, we observe that the system successfully provides stable latency to the low-heavy client (weak client), as we predicted.

Case 5: Infeasible system at capacity - Favor two low demand workloads

In this example, we run an experiment with three clients in a system above capacity to study fairness among multiple clients. We define unequal load and different weights between clients above capacity. The expected server utilization is 123%, and we predict an infeasible system as demand exceeds capacity. We choose to protect the two clients with the low expected load, high-light and low-light shown in Table 4.5.

Client Color	Client Label	Weight w_i	Arrival rate r_i (req/sec)	Service demand D_i (msec)	Latency target l_i (msec)	Expected utilization $r_i \cdot D_i$ (%)
Orange	High-light	0.25	12	20	500	24
Green	Low-heavy	0.4	8	80	500	64
Red	Low-light	0.35	7	50	500	35

Table 4.5: Workload specifications, expected utilization in units of processing time per wall-clock time



(a) EWMA Throughput (b) EWMA Latency (c) Utilization

Figure 4.5: Favor two low demand workloads

Discussion

Experimental results indicate that in a system operating below capacity, clients can achieve a stable latency without control and fairness. In such a case, weight settings are not key control parameters in resource allocation (i.e., there is ample capacity to stabilize the latencies of all clients). On the other hand, control and fairness is needed to a system that operates above capacity in order to protect weak clients whose latency may diverge. However, achieving explicit latency guarantees is a challenging task and further study is needed to decouple latency targets from throughput allocations alone. This is the subject of the next chapter.

Chapter 5

Fair throughput allocation with latency guarantees

In this section, we present Sprinkler, a two-level architecture for achieving performance QoS guarantees for latency- sensitive clients over shared resource pools. Sprinkler aims to enforce per-application latency targets by overlaying its utility-driven throughput-regulating Controller over the pClock arrival-curve scheduler described earlier at Chapter 4.

1 Design

A high-level depiction of the Sprinkler two-level QoS architecture appears in Fig. 5.1. In Sprinkler, applications specify their latency targets (e.g., <150ms) and their appreciation for throughput (or equivalently, their willingness to pay for it) at their latency target through their utility function. Sprinkler achieves fair sharing of resources by providing the same level of satisfaction to all applications while allocating as high throughput as possible. We aim to experiment with other notions of utility-based fairness in future work, such as utilitarian criterion [2] which aims to maximize social welfare. Allocations under this principle are thought of as a fair-sharing point of the total available capacity.

Sprinkler features a Controller that is aware of applications (maintains session state for them) and communicates with them via a two-way API. Applications make their characteristics (utility, latency target, maximum burst size) known to the Controller, and expect that the Controller will notify them of an arrival-rate allocation, once per control period, within which it conservatively believes that they can achieve their latency targets in the next control period. The Sprinkler Controller periodically adjusts the arrival-rate allocations based on the observed system status to effectively manage resource utilization. Specifically, it uses utilization measurements (U, current system load) and a threshold parameter, denoted by $\mathcal{U}$. When $U < \mathcal{U}$ while all applications fully utilize their Controller-specified rates, the Controller decides to increase the advisory allocations for all applications in a fair

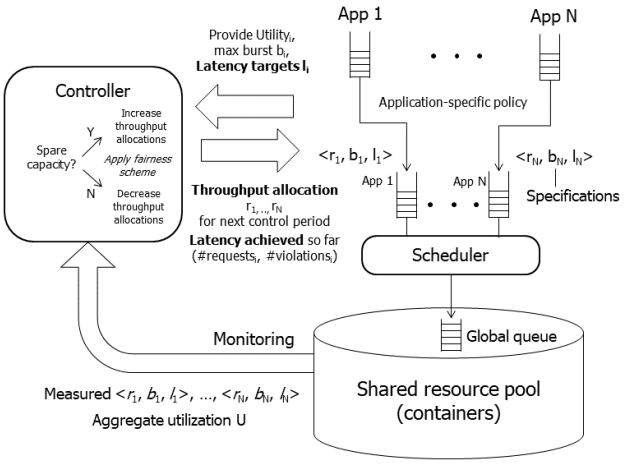


Figure 5.1: Sprinkler Architecture

(utility-wise) manner (a description follows below). To do so, Controller increases the utilities by a percentage of 20%. When $U > \mathcal{U}$, service-level objective (SLO) violations will likely occur, and so the Controller decreases the utilities to a fair share point of 0.5 in order to meet application requirements. If Controller still observes violations, decreases the utilities by a percentage of 10% to reduce advisory allocations. The assumption that utilization measurements for the resource pool are available allows the Sprinkler Controller to aim for controlling an aggregate metric (U) rather than several individual metrics (per-application latency targets) aiming to stay close to a feasible and efficient level (our prototype aims for about 80% utilized resource pools) using a standard PID controller [11]. This assumption is reasonable since most systems (such as compute or storage arrays) export, or can be extended to export, an aggregate utilization metric through their management interfaces. Systems assuming no support at all from the resource pool [18, 7] typically resort to controlling the issue-queue to the pool, an alternative for which however it is harder to set a reasonable goal (i.e., what aggregate latency to aim for). The Controller is typically conservative in its rate allocations for applications (i.e., based on conservative estimates using experimental data or simple performance models) and does not continuously adjust them to application-load variations. As such, applications that limit themselves to Controller allocations may miss opportunities for leveraging temporally available spare capacity and the overall efficiency of the system may be reduced.

An important distinctive feature of Sprinkler to address this problem, is that applications do not promise to limit themselves to the Controller-reported allocation but may individually decide to exceed them if they determine that they are ahead of their latency goals (in whatever percentile they decide to evaluate them, another important feature of Sprinkler). Applications can thus take the risk of sending at a higher rate than guaranteed at the risk of having SLO violations.

Based on the latency statistics they receive from Sprinkler, they are able to decide when to comply to the allocations suggested by the Controller, or not. When applications notice SLO violations, they decide to return to the advisory allocations. For as long as non-compliant applications do not learn of SLO violations, this is an indication that they succeed to benefit from the available spare capacity and continue to do so.

For performance isolation, Sprinkler utilizes a lower-level scheduler that takes as input per-flow load specifications (request rate[1], maximum burst size, latency target) and schedules requests in the resource pool in a way that will achieve latency targets as soon as the overall load is feasible. Since such schedulers already exist [8], Sprinkler leverages prior work in this space. Sprinkler makes applications aware of their individual latency metrics at any point in time (e.g., "recorded 10^3 latency-target violations out of 10^6 requests issued so far"), so that applications can decide if they are doing well enough (i.e., achieving <100ms in the 99.9-percentile of requests while it is sufficient to do so only in the 90%-percentile) to afford risking latency violations by exceeding their throughput allocations. If applications are barely making their latency targets they will opt for abiding by the controller's allocation for them. Applications that exceed their Controller-determined rates will be able to achieve higher throughput at their latency targets *only* if spare (excess) capacity is available in the resource pool at that time. Otherwise, such non-compliant applications will miss their latency targets. However, compliant applications will be protected (isolated). If many or all compliant applications exhibit latency violations, this is an indication that rate allocations exceed capacity and thus the Controller (triggered by the $U > \mathcal{U}$ condition at the same time) will reduce them to a new fair-sharing point taking into account their utilities.

In this work, we refer to fairness in resource allocation as the assignment of utilities (quantitative level of client goals and satisfaction) of different clients. We implement a proportional fairness scheme in order to maintain a balance among competing flows, assigning each application equal weight to determine same level of satisfaction. In particular, we fluctuate the utility values by the same proportion to provide a fair throughput allocation. Additionally, in case of system overload, we drive all applications to a fair point of utility to guarantee a fair share of throughput on equal terms.

In this work, we assume that each client's utility has three parts: the *throughput-specific* part which is an S-shaped sigmoid function (described earlier at Chapter 2), indicating the desire for a higher throughput allocation. The *slo-violations-specific* part, a reversed S-shaped sigmoid function (described earlier at Chapter 2), that implies a desire for facing as few violations as possible. The *latency-specific* part is a step function, namely it is zero for latency above a certain threshold l_{target}, and constant below that level (i.e., any latency below l_{target} is equally desirable). We assume that the total utility is the sum of the three parts [19]. Experimentation with different types of utility functions is a subject of future work.

[1]These are Controller-determined allocations, not actual application rates

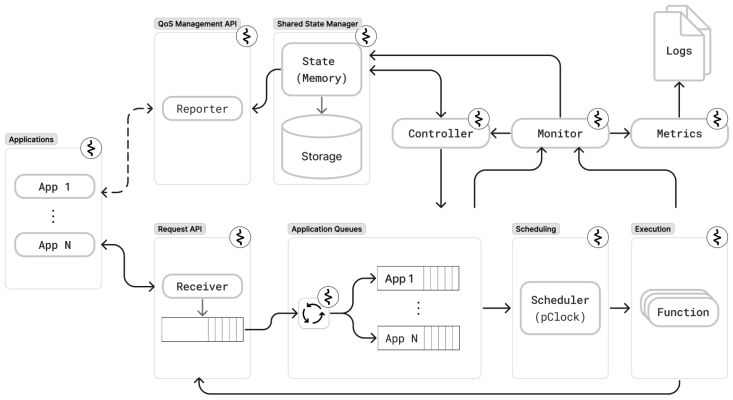


Figure 5.2: Sprinkler system implementation

2 Implementation

Our implementation, whose outline is shown in Fig. 5.2, follows staged event-driven architecture (SEDA) principles [29] and consists of a number of separate modules. Each module/stage is served by a separate thread, and the entire system is run as a single process. This minimizes the latency incurred in information sharing and module communication. The four stages of the Sprinkler execution pipeline are:

The Request API server (Fig. 5.2, lower left), responsible for receiving and pre-processing applications requests (in the case of our prototype, requests to execute specific functions). In case a response is needed to acknowledge receipt so that an application can proceed with sending new requests, the server responds with an acknowledgment as soon as a request is enqueued in the incoming buffer. The request will be processed by the execution module and eventually the actual response will be returned to the application (bottom arrow).

In the second stage of the pipeline, a consumer thread dequeues a request from the incoming buffer and demultiplexes to separate flows, termed Application Queues, with one such queue per application session. This way, a lower-level scheduler can apply fine-grained flow-specific policies. Each Application Queue maintains certain metadata in addition to buffered requests.

In the third stage in the execution pipeline, we implemented the pClock [8] scheduler to enforce isolation across latency-sensitive applications. To support pClock, Sprinkler maintains application profiles (controller-set request rate, maximum burst size, latency target) as shared state, and continuously maintain the minimum and maximum queue start tags, which are used to determine the start tag for incoming requests. pClock schedules requests to meet the latency targets requested by each application.

The fourth stage in the Sprinkler pipeline is request execution over a resource pool. Our prototype execution engine is an instance of the function-as-a-service (FaaS) execution model where each request triggers a function call with its arguments over an available core within the pool.

The four main stages of the Sprinkler execution pipeline are augmented by several management modules. The Monitor module is in charge of maintaining an overview of the state of the entire system at all times. Every request transition (such as arrival, enqueuing, execution start, etc.) is timestamped and transmitted to the monitor for bookeeping and to support reporting and analysis. Data from the monitor is gathered as metrics and stored in log files at the end of a run to allow post-processing. The key metrics are the arrival rate, response time, throughput and utilization. For all metrics, we use a smoothing average method (EWMA) to reduce noise (Chapter 3). The maximum burst size of each application is also independently measured for validation of the initial user-provided estimate. The Controller is in charge of partitioning available pool throughput between application sessions. Every second, the monitor notifies the controller of the current status (such as arrival rate and utilization), and the controller uses this information, as well as the maximum burst size and the utility functions, to determine whether to increase or decrease the arrival-rate allocations of applications during each control period. At each control period, it changes the settings of the application queues (e.g., adjusting the initial user-provided maximum burst size estimate, if needed) and exports the new information (allocated rates, total number of requests and total number of violations so far) to applications through the QoS management API. The current control period is 10 seconds, however in some experiments we set 60 seconds for easier visualization of results.

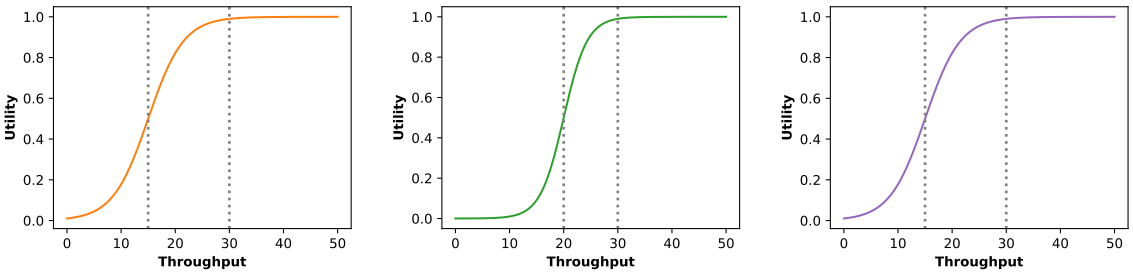
The Shared State Manager is responsible for storing application settings and important metrics on a per-application basis. The application settings are used upon queue initialization, and metrics (number of executed requests and SLO violations) are requested from apps through the QoS management API to learn about their progress. These two measures enable them to compute their latency percentiles and determine whether they can afford to increase their demands beyond what the controller recommended or maintain the recommended rates.

Chapter 6

Evaluation

We evaluate Sprinkler through a series of experiments that highlight the guarantees it provides to applications that are concurrently accessing a shared resource pool. Our main experimental testbed consists of four servers, each equipped with an Intel Xeon Bronze 3106 8-core 1.70GHz CPU, 16GB DDR4 2666MHz DIMMs, 256GB Intel D3-S4610 SSD and 2TB Ultrastar 7K2 HDD, running Ubuntu Linux 16.04.6 LTS, interconnected via a 10Gb/s Dell N4032 switch.

The following experiments are conducted using only throughput-utility functions in the decision-making process, and all applications feature the following sigmoid functions.



(a) App1 (b) App2 (c) App3

Figure 6.1: Utility functions

1 Performance isolation and use of excess capacity

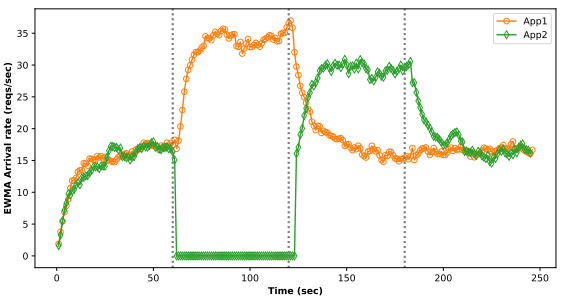
In the first experiment, we demonstrate how Sprinkler isolates an application that honors its specification from an application that does not under saturation, while it also allows the latter to use spare capacity when left unused by other applications.

The experiment evolves over 4 consecutive phases of 60 seconds each (Fig. 6.2 (a)-(c)). The allocations for each application set by the Sprinkler controller (also

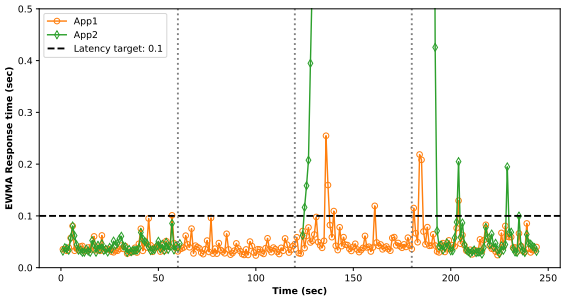
Client Color	Client Label	Max burst size b_i (reqs)	Initial arrival rate r_i (req/sec)	Service demand D_i (msec)	Latency target l_i (msec)	Expected utilization $r_i \cdot D_i$ (%)
Orange	App1	4	20	20	100	40
Green	App2	5	20	20	100	40

Table 6.1: Workload specifications, expected utilization in units of processing time per wall-clock time (Exp. 1)

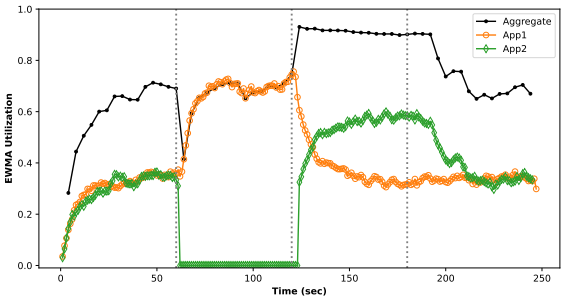
referred to as its specifications, used as input to pClock) dictate that each application sends at an average of 20 req/sec. The latency target of each application is 100ms. The complete workload specifications in this experiment are outlined in Table 6.1. Some of these parameters are set at our workload generator (Locust [17]) while others, such as the maximum burst size, are measured within the prototype. All reported metrics are measured and collected by the Sprinkler monitoring system.



(a) Arrival rate (EWMA) for applications 1 and 2



(b) Response time (EWMA) of applications 1 and 2



(c) Utilization (EWMA) for each application and aggregate

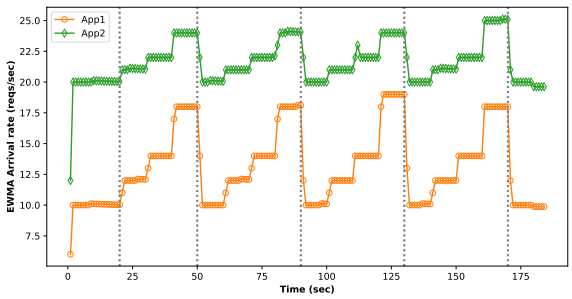
Figure 6.2: Fair regulation of throughput

Client Color	Client Label	Max burst size b_i (reqs)	Initial arrival rate r_i (req/sec)	Service demand D_i (msec)	Latency target l_i (msec)	Expected utilization $r_i \cdot D_i$ (%)
Orange	App1	6	10	20	100	20
Green	App2	6	20	20	100	40
Purple	App3	6	10	20	100	20

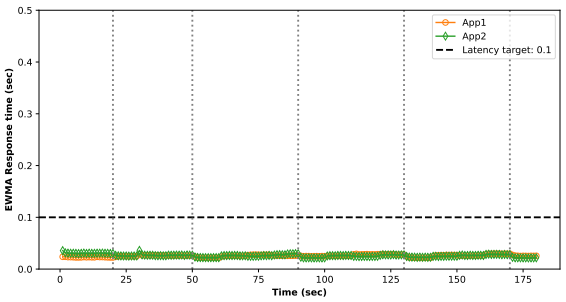
Table 6.2: Workload specifications, expected utilization in units of processing time per wall-clock time (Exp. 2, 3, & 4)

2 Fair throughput regulation

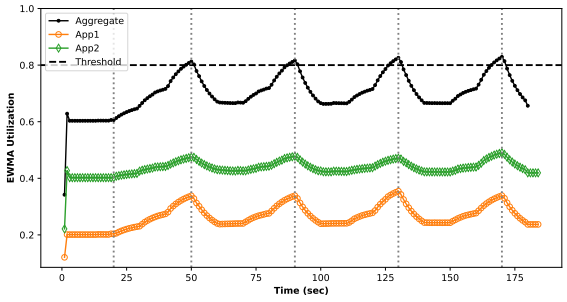
In the second experiment, we show how the Sprinkler controller is fairly regulating aggregate throughput between applications through modification of their allocations (and thus their specifications used as input to pClock) when the aggregate load or system capacity changes.



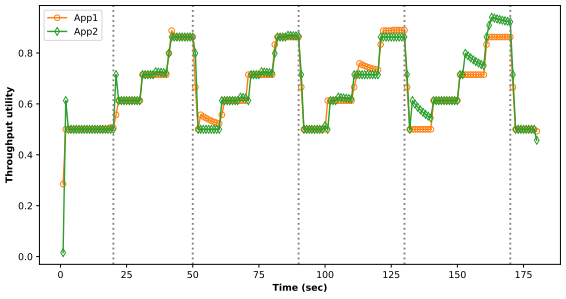
(a) Arrival rate (EWMA) for applications 1 and 2



(b) Response time (EWMA) of applications 1 and 2



(c) Utilization (EWMA) for each application and aggregate



(d) Throughput utility for applications 1 and 2

Figure 6.3: Fair throughput regulation

The experiment runs for three minutes to depict the policies that the controller applies to control allocations.

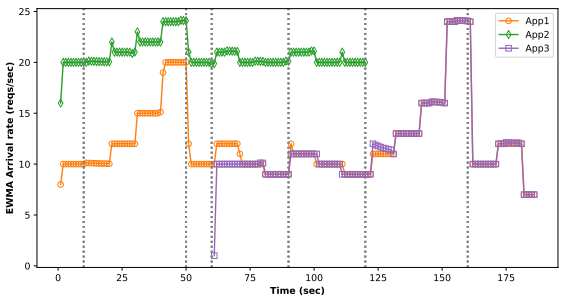
In the first 20sec, application 1 sends at an average of 10 req/sec and application 2 sends at an average of 20req/sec. The Sprinkler Controller every 10 seconds and up to 50 seconds, increases the utilities by percentage of 20% (Fig. 6.3d). In

the 50-th sec, Controller detects the aggregate utilization approaches to 80% and decreases application utilities to a fair sharing point of 0.5, reducing the utilization to about 70% (Fig. 6.4c). The remainder of this experiment depicts a repeated behavior that reflects controller adaptation to system capacity.

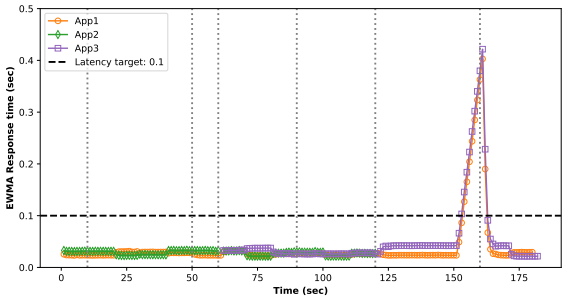
3 Adaptation to workload variations

In the third experiment, we demonstrate how the Sprinkler controller adapts to changes in the environment.

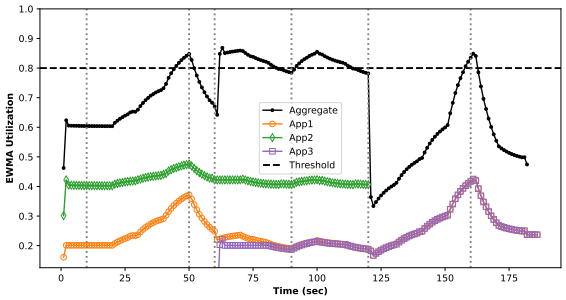
The next experiment runs for three minutes (Fig. 6.4 (a)-(d)) and exhibits Sprinkler's capability to adjust application request rates for efficiency and to achieve latency guarantees. The complete workload specifications are outlined in Table 6.2.



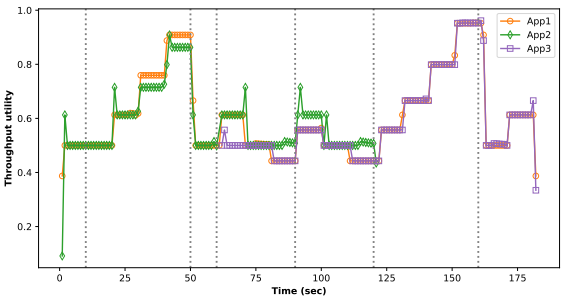
(a) Arrival rate (EWMA) for applications 1 and 2



(b) Response time (EWMA) of applications 1 and 2



(c) Utilization (EWMA) for each application and aggregate



(d) Throughput utility for applications 1 and 2

Figure 6.4: Adaptation to workload variations

In the first 10sec, applications 1 and 2 send at an average of their initial arrival rate (10req/sec for application 1 and 20req/sec for application 2). The Sprinkler Controller increases allocations for applications 1 and 2 based on their utilities up to 50 seconds, as spare capacity is available to harness as seen in aggregate pool utilization (Fig. 6.4c). In the 50-th sec, the aggregate utilization exceeds 80% and

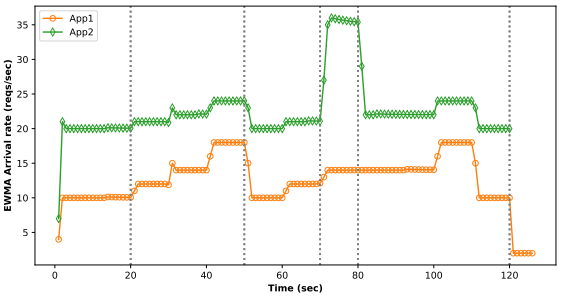
Controller decreases application utilities to a fair sharing point of 0.5 (Fig. 6.4d). In the 60-th sec, a third application (App3) enters and ramps up to sending at an average rate of 10 req/sec, while applications 1 and 2 keep their previously set advisory rates (observing 10 req/sec for application 1 and 20req/sec for application 2). As the aggregate utilization exceeds 80%, all applications driven to a fair point of 0.5 (Fig. 6.4d). At the 90-th sec the aggregate utilization is below to 80% and Controller increases application utilities, indicating an increase in advisory allocations. In the next minutes (90-120sec), Sprinkler adjusts allocations for all applications based on condition through monitoring system utilization and all applications respond by proportionally reducing rates to Sprinkler's advice (Fig. 6.4a). In 120-th sec, application 2 becomes temporarily idle (no requests issued, but not departing the system), releasing resources that can be used by the other two applications. Applications 1 and 3 are probing for additional throughput by speculatively increasing rates to about 24 req/sec, benefiting from the spare capacity until 160 seconds (Fig. 6.4a). In the 160-th sec, Controller returns applications to a fair sharing point of 0.5 (Fig. 6.4d) as the system utilization exceeds 80%.

4 Application-specific rate-control policy

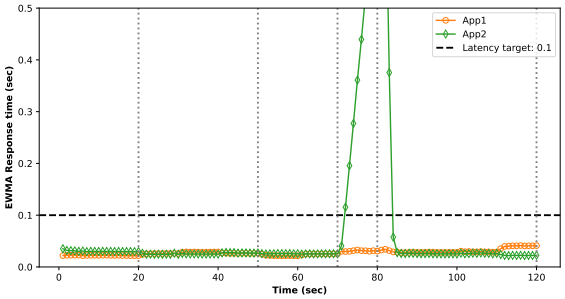
Next we focus on dynamic and application-specific policy for rate control to explore additional throughput based on an assessment of the latency achieved so far vs. their targets.

In this set of experiments we demonstrate application-specific rate control policy, according to which non-compliant applications may occasionally probe for additional throughput if justified by a risk assessment of their latency targets (SLOs). We show how such policies can at times benefit applications when spare capacity is available, and how they can affect and be affected by latency targets over time.

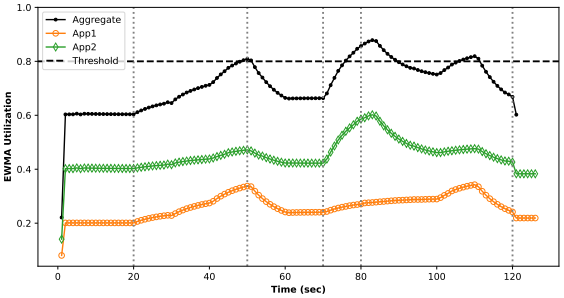
The first experiment (C_1) runs for two minutes (Fig. 6.5) during which two applications share a resource pool with a latency target (median or 50-percentile) of 100ms. In the first 20sec, both applications send at their initial rate allocations. Until 50 seconds, the Controller detects spare capacity available (as the total resource-pool utilization is reported less than 80%) and informs applications for their new advisory allocations, increasing their utilities by a percentage of 20% at each time decision (Fig. 6.5d). In the 50-th sec, Controller detects that aggregate utilization is above 80% (Fig. 6.5c) and decreases application utilities to a fair sharing point of 0.5 (Fig. 6.5d). In the 70-th sec, application 2 evaluates its latency profile, determining that nearly 100% of requests are below 100ms so far, so a significant number of violations can be afforded and still satisfy the latency-target at the median. It thus decides to exceed the (advisory) allocation by sending at an average of 35 req/sec. As seen in Figs. 6.5a and 6.5c, there is a throughput increase for the application, but it also pushes total utilization to about 85% yielding latency-target violations for the non-compliant application



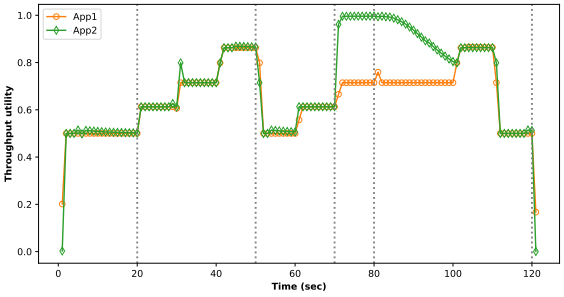
(a) Arrival rate (EWMA) for applications 1 and 2



(b) Response time (EWMA) of applications 1 and 2



(c) Utilization (EWMA) for each application and aggregate

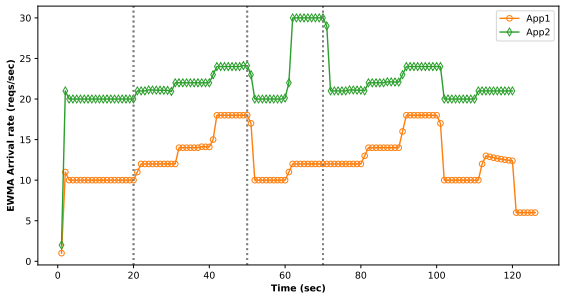


(d) Throughput utility for applications 1 and 2

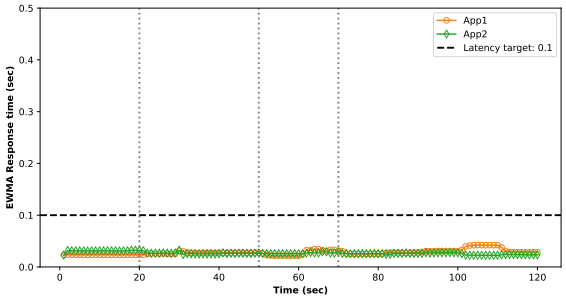
Figure 6.5: Application-specific rate control (C_1): Fail attempt

(Fig. 6.5b). However, the compliant application (Application 1) is protected (isolated) and does not register latency-target violations during this phase. In the remainder of this experiment (80-120sec), application 2 assesses that the latency-target violations risk hurting its overall SLO and thus decides to comply with the Controller-determined rate allocation, sending at an average of 20 req/sec. This restores its desired latency profile.

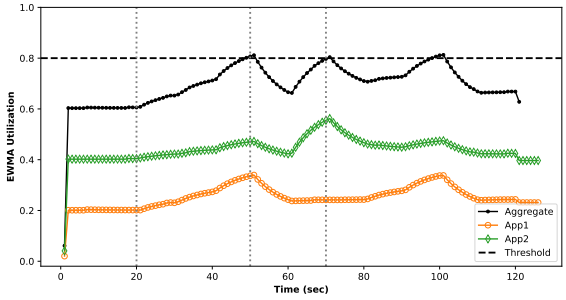
We setup another experiment (C_2, Fig. 6.6) that differs from the previous one in phase of 60-70sec where one of the applications becomes non-compliant. Just as in the previous experiment, in the 60-th second application 2 decides to increase the arrival rate to 30 req/sec while application 1 complies to the Controller-defined allocation (10 req/sec). Application 2 notes that it benefits from the available spare capacity as it does not observe SLO violations (Fig. 6.6b). However, in the 70-th sec decides to comply to the advisory allocation (20req/sec) decreasing the aggregate utilization (Fig. 6.6c).



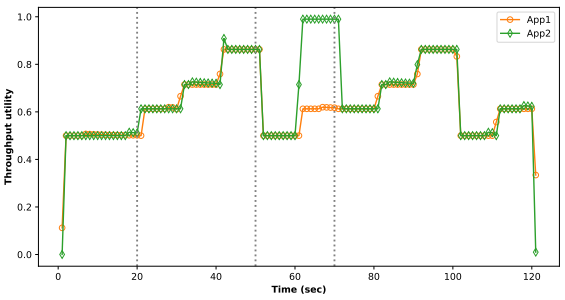
(a) Arrival rate (EWMA) for applications 1 and 2



(b) Response time (EWMA) of applications 1 and 2



(c) Utilization (EWMA) for each application and aggregate



(d) Throughput utility for applications 1 and 2

Figure 6.6: Application-specific rate control (C_2): Successful attempt

Chapter 7

Conclusions

In this thesis, we study and experimentally test utility-driven fair sharing mechanisms to effectively provide QoS goals in autonomic computing systems. Although many weighted fair queuing algorithms [5, 4, 6] have been proposed to provide throughput and low latency guarantees, the latency and throughput requirements cannot be independently controlled. Experimental results (Chapter 4) show that latency can be controlled by adjusting the weights, and low latency can only be achieved by increasing throughput. Instead, the QoS framework we propose (Chapter 5) achieves throughput-latency decoupling while providing fair throughput regulation and latency guarantees.

Sprinkler follows a two-layer architecture in which the throughput-oriented high-level layer determines and publishes conservative, fair-share request-rate allocations based on simple performance models and aiming for efficient use of the resource pool. The lower-level latency-oriented scheduler [8] achieves application-specific latency targets, assuming that input specifications are feasible, isolating well-behaved applications from those that exceed their specifications when the aggregate load exceeds capacity. Our evaluation of the Sprinkler prototype demonstrates that it effectively and fairly allocates aggregate resource-pool capacity in concurrent applications, maintaining their latency goals. Sprinkler allows application-specific control (based on advisory allocations and feedback on the latency achieved so far in a run). It combines isolation of compliant applications from those exceeding their allocations, and speculation by applications exceeding their SLO to increase efficiency, while maintaining aggregate use of the resource pool below capacity.

Chapter 8

Future work

In this thesis, we experimented with uniform clients with equal service demand and latency targets for initial testing of our prototype. Further experimental evaluation with different specifications (including bursty workloads) should be conducted to throroughly validate Sprinkler. Beyond gaining experience with different types of clients, in the future we plan to experiment with different utility functions and fairness schemes. We would also like to focus more on the theoretical analysis and study the impact of different configuration values on the system, such as utility variation percentages, desired system utilization level (threshold) and utility fair point, to speed up the rate of system convergence.

Furthermore, we would like to further develop controller and application-specific client policies. Specifically, we plan to consider cases where speculative applications exceed their advisory allocations in order to take advantage of spare resources, temporarily driving the resource pool above capacity. In such cases, the controller should not be applying a fairness action (reducing advisory allocations) since the speculative applications will eventually throttle themselves back to the fair allocation point based on their observed SLO violations. The controller should instead apply fairness actions only when it observes that the advisory throughput allocations are infeasible for the overall system, and the utilities to consider at that point should be based on the advisory allocations for each client.

Finally, we would like to consider a possible extension of Sprinkler to operate in a distributed environment in which global information and synchronization challenges will have to be addressed.

Bibliography

[1] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 43–56, New York, NY, USA, 2001. Association for Computing Machinery.

[2] Dimitris Bertsimas, Vivek F. Farias, and Nikolaos Trichakis. The Price of Fairness. *Operations Research*, 59(1):17–31, 2011.

[3] Y. Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi MG'Oller, and P. Pezze. *Engineering self-adaptive systems through feedback loops.* Jan 2009.

[4] John Bruno, Jose Brustoloni, Eran Gabber, Banu Ozden, and Abraham Silberschatz. Disk scheduling with quality of service guarantees. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems - Volume 2*, ICMCS '99, USA, 1999. IEEE Computer Society.

[5] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *SIGCOMM Comput. Commun. Rev.*, 19(4):1–12, aug 1989.

[6] P. Goyal, H.M. Vin, and Haichen Cheng. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Transactions on Networking*, 5(5):690–704, 1997.

[7] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *Procceedings of the 7th Conference on File and Storage Technologies*, FAST '09, pages 85–98, USA, 2009. USENIX Association.

[8] Ajay Gulati, Arif Merchant, and Peter J. Varman. pClock: An Arrival Curve Based Approach for QoS Guarantees in Shared Storage Systems. *SIGMETRICS Perform. Eval. Rev.*, 35(1):13–24, jun 2007.

[9] Ajay Gulati, Arif Merchant, and Peter J. Varman. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *Proceedings of the*

9th USENIX Conference on Operating Systems Design and Implementation, OSDI'10, pages 437–450, USA, 2010. USENIX Association.

[10] Vipul Gupta, Soham Phade, Thomas Courtade, and Kannan Ramchandran. Utility-based resource allocation and pricing for serverless computing, 2020.

[11] Joseph L. Hellerstein, Yixin Diao, Sujay S. Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. Wiley, 2004.

[12] Wei Jin, Jeffrey S. Chase, and Jasleen Kaur. Interposed proportional sharing for a storage service utility. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, pages 37–48, New York, NY, USA, 2004. Association for Computing Machinery.

[13] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing, 2019.

[14] Magnus Karlsson, Christos Karamanolis, and Xiaoyun Zhu. Triage: Performance differentiation for storage systems using adaptive control. *ACM Trans. Storage*, 1(4):457–480, nov 2005.

[15] Flora Karniavoura and Kostas Magoutis. Decision-Making Approaches for Performance QoS in Distributed Storage Systems: A Survey. *IEEE Transactions on Parallel and Distributed Systems*, 30(8):1906–1919, 2019.

[16] Ning Li, Hong Jiang, Dan Feng, and Zhan Shi. PSLO: Enforcing the Xth Percentile Latency and Throughput SLOs for Consolidated VM Storage. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[17] Locust. An open-source load testing tool. `https://locust.io/`. Accessed: 2023-02-01.

[18] Christopher R. Lumb, Arif Merchant, and Guillermo A. Alvarez. Façade: Virtual storage devices with performance guarantees. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 131–144, USA, 2003. USENIX Association.

[19] Marta Różańska and Geir Horn. Marginal metric utility for autonomic cloud application management. In *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, UCC '21, New York, NY, USA, 2022. Association for Computing Machinery.

[20] David Shue, Michael J. Freedman, and Anees Shaikh. Fairness and isolation in multi-tenant storage as optimization decomposition. *SIGOPS Oper. Syst. Rev.*, 47(1):16–21, jan 2013.

[21] David Sinreich. An architectural blueprint for autonomic computing. 2006.

[22] John D. Strunk, Eno Thereska, and Christos Faloutsos. Using utility to provision storage systems. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, San Jose, CA, February 2008. USENIX Association.

[23] G. Tesauro, R. Das, W.E. Walsh, and J.O. Kephart. Utility-function-driven resource allocation in autonomic systems. In *Second International Conference on Autonomic Computing (ICAC'05)*, pages 342–343, 2005.

[24] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. IOFlow: A Software-Defined Storage Architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 182–196, New York, NY, USA, 2013. Association for Computing Machinery.

[25] Jean Walrand. Notes on Token Buckets, GPS, and WFQ.

[26] W.E. Walsh, G. Tesauro, J.O. Kephart, and R. Das. Utility functions in autonomic systems. In *International Conference on Autonomic Computing, 2004. Proceedings.*, pages 70–77, 2004.

[27] Bin Wang, Ahmed Ali-Eldin, and Prashant Shenoy. LaSS: Running Latency Sensitive Serverless Computations at the Edge. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '21, pages 239–251, New York, NY, USA, 2021. Association for Computing Machinery.

[28] David X. Wei, Cheng Jin, Steven H. Low, and Sanjay Hegde. FAST TCP: Motivation, Architecture, Algorithms, Performance. *IEEE/ACM Transactions on Networking*, 14(6):1246–1259, 2006.

[29] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, oct 2001.

[30] S.R. White, J.E. Hanson, I. Whalley, D.M. Chess, and J.O. Kephart. An architectural approach to autonomic computing. In *International Conference on Autonomic Computing, 2004. Proceedings.*, pages 2–9, 2004.