LUND UNIVERSITY

Cloud Application Predictability through Integrated Load-Balancing and Service Time Control

Nylander, Tommi; Thelander Andrén, Marcus; Årzén, Karl-Erik; Maggio, Martina

Published in:
Proceedings of the 15th IEEE International Conference on Autonomic Computing

DOI:
10.1109/ICAC.2018.00015

2018

Document Version:
Peer reviewed version (aka post-print)

Link to publication

Citation for published version (APA):
Nylander, T., Thelander Andrén, M., Årzén, K.-E., & Maggio, M. (2018). Cloud Application Predictability through Integrated Load-Balancing and Service Time Control. In *Proceedings of the 15th IEEE International Conference on Autonomic Computing* IEEE Computer Society. https://doi.org/10.1109/ICAC.2018.00015

Total number of authors:
4

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

Cloud Application Predictability through Integrated Load-Balancing and Service Time Control

Tommi Nylander, Marcus Thelander Andrén, Karl-Erik Årzén, Martina Maggio
Department of Automatic Control, Lund University

***Abstract*—Cloud computing provides the illusion of infinite capacity to application developers. However, data center provisioning is complex and it is still necessary to handle the risk of capacity shortages. To handle capacity shortages, graceful degradation techniques sacrifice user experience for predictability. In all these cases, the decision making policy that determines the degradation interferes with other decisions happening at the infrastructure level, like load-balancing choices. Here, we reconcile the two approaches, developing a load-balancing strategy that also handles capacity shortages and graceful degradation when necessary. The proposal is based on a sound control-theoretical approach. The design of the approach avoids the pitfalls of interfering control decisions. We describe the technique and provide evidence that it allows us to achieve higher performance in terms of emergency management and user experience.**

I. Introduction

Capacity provisioning is of crucial importance in modern distributed computation infrastructures. To determine the size of data centers, and properly dimension the resources to be allocated in each geographic location, most data center owners use predictions of the computational needs [29, 36]. The computational resource within a data center is then used to serve requests coming from multiple clients, providing the illusion of infinite capacity and, as a result, the possibility of bounding the latency [5, 6, 14, 24, 25, 42]. To do so the architecture uses multiple instances of the same application, here called *replicas*, and predictions and estimations of traffic and needed computational capacity.

The predictions of the incoming traffic and the corresponding estimates [16, 17] of the required computational capacity are necessarily subject to errors and uncertainty [4]. The presence of these errors naturally leads to two possible management strategies. The first strategy is over provisioning [15, 43]. Over provisioning increases the management cost for a cloud application, but guarantees user satisfaction. The second strategy is provisioning according to expectations and handling capacity shortages via user experience degradation [7, 9, 27, 34, 41], or via approximate computing [22, 37, 40]. Generally speaking, these ways of handling capacity shortages are typically clustered under the umbrella of *graceful degradation*.

Graceful degradation techniques involve taking corrective actions (that typically degrade the user experience) to ensure that the computing platform achieves predictability (for example, that any request receives a response within a given time). For example, Brownout [27] sacrifices the quality of the response given to users to ensure that a large fraction of the requests experience a predictable latency. Brownout is based on a control approach [13, 32, 33], and a controller selects – at the replica level – requests to be answered with full quality (both the mandatory and the optional part of the response are computed) and requests to be given an approximate answer (only the mandatory part is computed). The approach has proven to be successful to bound the response times of single machines. It was then combined with load-balancing strategies [11, 28], showing that the control strategy at the replica level and the load balancer could interfere with one another, potentially limiting each others benefits. For example, load-balancing strategies based on response times are to be avoided when a replica control strategy that bounds the response times is used [11]. This is not only true for brownout, but for every technique that enforces bounded response times [5, 6], like admission control policies [26, 38].

In general, the interference between two control policies is a complex problem [8, 21]. Two different decision making strategies, both working well in isolation, can interfere in unpredictable ways with one another, especially when there are delays between the two decisions. For example, the Shortest Queue First (SQF) load-balancing policy has degraded performance when a queue control strategy (like graceful degradation, or admission control) is active at the replica level, as can be seen in the example of Section II.

We propose a load-balancing and graceful degradation policy that takes into account both the decisions with the advantage of better controlling the response times and the resource utilization of the data center. This paper makes the following contributions:

- It identifies problems with the currently used load-balancing policies, due to the interplay between graceful degradation techniques at the replica level and load balancers that should distribute the load to multiple replicas.
- It proposes a new architecture, with a higher degree of controllability, that includes both load balancing and graceful degradation, solving the mentioned problems.
- It presents the control design for each of the elements in this architecture.
- It validates the proposal with an experimental campaign, comparing it to existing techniques. The proposed architecture outperforms existing ones in terms of predictability and resource usage. It is in fact able to achieve lower variance for the response times, utilizing the data center resources more efficiently.

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, by the Swedish Research Council (VR) for the projects "Feedback Computing" and "Power and temperature control for large-scale computing infrastructures", by the LCCC Linnaeus Center and, by the ELLIIT Excellence Center at Lund University.

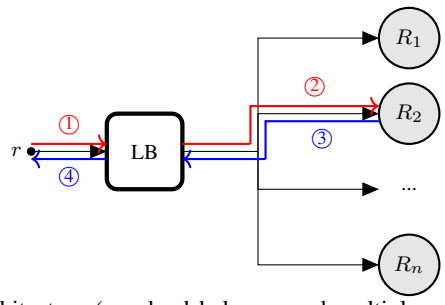

Fig. 1: Architecture (one load balancer and multiple replicas) and path of one single request ρ from the user request (step ①) to the response forwarding (step ④).

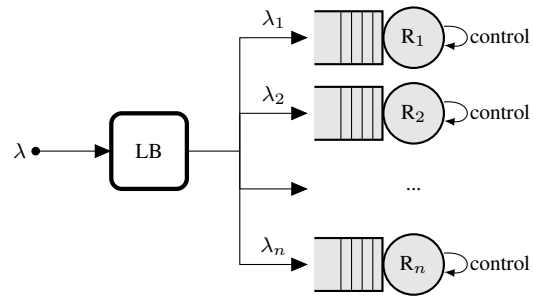

Fig. 2: The standard load-balancing architecture. The load balancer routs incoming requests directly to a replica, where the request might spend some time queuing before service. Replicas include graceful degradation controllers.

The paper is organized as follows. Section II provides a more precise statement of the problem our solution solves, and details why this is necessary for modern data centers. Section III describes our control solution, and shows block diagrams for all the elements involved. It also offers an analysis from the control perspective of the behavior of the cloud platform. Section IV provides experimental evidence for our claims and shows that the proposed approach is easy to implement and offers competitive advantages in terms of response time management. Section V casts the proposed solution in the state of the art, and Section VI concludes the paper.

II. Problem Statement

This paper deals with the problem of designing a load-balancing architecture with graceful degradation. We assume that the architecture is composed of one single load balancer (denoted with LB) and a set of n replicas (denoted with $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$. The goal of the architecture is to achieve high service predictability. We translate predictability into two related objectives, and measure it in terms of the response times for incoming requests. We want a statistic on the response times (e.g., average, 95th percentile, 99th percentile) to follow a setpoint (a predetermined value, specified for the given cloud application). Also, we want to minimize the variance in response time. A low variance of the worst-case response times, in fact, corresponds to a high degree of predictability. In the remainder of this paper, we assume a setpoint on the 95th percentile of the response times, and use the integrated absolute error (IAE) with respect to this setpoint as our predictability metric. However, similar considerations can be drawn using other statistics.

The path of one single request is shown in Figure 1. We assume that all requests enter the system through one central load balancer (step ①), which in turn routes each request to one of the n replicas (R_2 in the Figure, as shown by step ②). Finally, the replicas serve the requests. Each replica is capable of performing graceful degradation, and thus can choose to serve different amount of content, which requires more or less service time. Here we use brownout [27] for graceful degradation, but other techniques can be applied. Using brownout implies that a request can be served either with or without optional content. The service time used to compute the optional content can be spared, in case the replica detects some capacity shortage. The replica determines the response to the request and communicates it to the load balancer (step ③), which finally replies to the user (step ④). Notice that this is the standard path of a request in a multi-replica architecture, used in practical applications and also in earlier research [11, 28]. In fact, the replica cannot directly respond to the user, that has queried the server using the IP address of the load balancer. The user would not identify the replica as the server that was queried and would then terminate the connection.

In this architecture, the response produced in step ③ can be used to "piggy-back" information from the replica to the load balancer, without incurring an additional overhead in response time. The load balancer then tears the envelope of the response received by the replica, and only answers to the user with the actual message, in step ④.

The mentioned architecture is commonly implemented as shown in Figure 2. Each replica has an individual queue for requests, and the load balancer routes requests to the queues based on some policy e.g. Round-Robin, SQF, or a weighted probability. In turn, each individual replica has a local graceful degradation strategy — in the brownout case, a response time controller which decides if to serve optional content or not based on the last measured response time from that replica. While this architecture is conceptually simple, the predictability of the response times is highly dependent on the co-design of the load-balancing policy and the controllers in the replicas. The design will also depend on the service discipline used in the replicas (e.g. "First-In First-Out" (FIFO) or Processor Sharing (PS)). In this paper, we will assume a generalized concept of PS being used in the replicas. Specifically, the replicas will serve at a maximum M_C number of requests concurrently from the queue. FIFO and standard PS are then simply the special cases $M_C = 1$ and $M_C = \infty$ respectively. For further details, see [35].

After being routed by the load balancer, requests will spend some non-zero time queuing before service by the replica is started. The average time spent queuing will vary with e.g. workload λ, number of concurrently served requests M_C, etc., and will introduce a delay between the decisions made by the load balancer and the local replica controllers respectively. This is a problem, since delays in between the decisions introduce the risk of routing and graceful degradation counter-acting each other. Load-balancing policies which have been shown to perform well in case of static service rates can actually counter-act the work of the local controllers in the replicas,

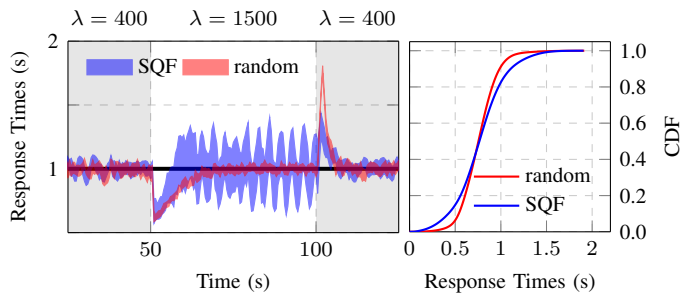

Fig. 3: Comparison between random and SQF load-balancing. The left plot shows setpoint and 95% confidence intervals for the 95th percentile of the response times of the optional-content requests served by a replica. The right plot shows the Cumulative Distribution Function (CDF) of all response times.

leading to poor predictability of response times. This can for example be the case with SQF, despite it being regarded as one of the best load-balancing alternatives [11, 28]. An example of this phenomenon is shown in Figure 3. The plots depict the results of an experiment conducted with a simulator[1] that emulates an architecture composed of a load balancer and 5 equal replicas with local graceful degradation controllers, i.e. as in Figure 2, with $n = 5$. The local controllers are using the feedback control strategy from [35], that determines the optional content computation. Each replica in the simulation takes on average 0.014 s to compute the optional content part of the response (with a variance of 0.01s^2), and 0.0002s on average for the mandatory part (with a variance of 0.001s^2). A maximum of $M_C = 15$ requests can be served concurrently in each replica. The run was repeated 20 times, in order to be able to show statistically significant behaviors (using 95% confidence intervals). The simulator uses the open-loop client model and the request arrivals are modeled using the Poisson distribution with arrival rate λ. The simulation is split into three different time intervals, in each of them the arrival rate λ is varied. In the time intervals $[0, 50)$ and $[100, 150]$, $\lambda = 400$ and in the time interval $[50, 100)$ $\lambda = 1500$.

The figure compares the SQF load-balancing strategy with a random load balancer. The leftmost plot shows confidence intervals for the 95th percentile of the response times of the requests served with optional content (the critical ones) and their setpoint of 1 s. The rightmost plot shows the Cumulative Distribution Function (CDF) for the two strategies. The use of SQF generates a higher variance in the response times, most notably during the period of heavy workload with $\lambda = 1500$ when requests will spend more time queuing at the replicas. Notably, SQF is performing worse than the simpler random choice policy. Even using specifically "brownout-aware" load-balancing policies [19, 28], maintaining predictable response times using the architecture of Figure 2 (the *de facto* standard architecture) remains a challenging task due to the interplay between the different control loops.

To avoid this problem, we instead opt for designing a new architecture where the design of the load-balancing policy and of the local controllers can be done separately, with the aim for them to integrate well from the start. The total response time of

[1]For a description of the simulator used, see Section IV-A.

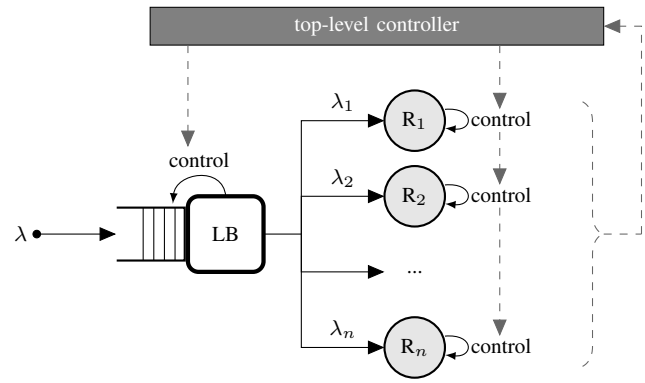

Fig. 4: The proposed load-balancing architecture.

a request is divided into two distinct parts: (i) waiting time, and (ii) service time. The load balancer controls the waiting time, and the local controller keeps the service time at a setpoint. In the following section, we describe our proposal, and detail the policies used for both load-balancing and graceful degradation, based on a control-theoretic approach.

III. Proposed Solution

Based on the idea of separating the control of the response times into two distinct parts (one for queueing time and one for pure service time), we propose the load-balancing architecture shown in Figure 4. Contrary to the architecture shown in Figure 2, our proposal contains only one central queue for incoming requests, situated at the load balancer.

The load balancer routes requests from the central queue in a "first come first served" manner. When the load balancer routes a request, a controller decides if the request should be served with optional content (normally) or not (i.e., applying graceful degradation). Based on this decision, the load balancer then attaches a flag to the request and forwards it to the replica with the highest demand for a new request.

In each replica, all the forwarded requests are assumed to be served concurrently. From the implementation perspective, each request is served in a separate thread, and all the threads are run concurrently, sharing the computational capacity. At most M_C requests may be served concurrently at each replica. Intuitively, an increase in the number of concurrent requests should result in a longer service time for each of them, and we will use this assumption here.

When a response is produced, a local controller in the replica decides how many more requests it desires to handle, and attaches this integer value to the response. The response is sent back to the load balancer, triggering an event where the attached integer value is used to update a list which keeps track of the current demand of requests from each replica. The load balancer then uses this list to decide where to route the next requests, distributing the requests from to replicas according to their desires. In summary, a request entering the proposed architecture in Figure 4 goes through the following steps — $\langle$LB$\rangle$ indicates that the step is performed by the Load Balancer, $\langle$R$\rangle$ that it is performed by the Replica:

1) ⟨LB⟩ The request is put in the queue.
2) ⟨LB⟩ The request waits until it reaches head of the queue.
3) ⟨LB⟩ Routing is triggered with replica demands.
4) ⟨LB⟩ An "optional content" flag is attached to the request.
5) ⟨LB⟩ The request is forwarded to the replica.
6) ⟨R⟩ The request is served by the replica.
7) ⟨R⟩ A response is produced.
8) ⟨R⟩ A new demand value is attached to the response.
9) ⟨LB⟩ The response triggers routing modifications.
10) ⟨LB⟩ The response is sent to the user.

Assuming the time overhead due to routing is negligible, the delay between routing and graceful degradation decisions is now removed. The total response time for a request is separated into: (i) the waiting time in the central queue at the load balancer (step 2), and (ii) the service time in one of the replicas (step 7). The controller in the load balancer decides if optional content should be served or not (step 4), based on a setpoint on the waiting time in the queue (on the time needed to complete step 2). We will refer to this controller as the *waiting time controller*. By flagging a request to be served with optional content or not, the waiting time controller increases or decreases the throughput of the queue, thus affecting the waiting time of future requests.

The local controller in each replica decides how many more requests the replica should demand (step 8). This is based on a setpoint for the service time of requests (for the time needed to complete step 7). We will refer to this controller as the *service time controller*. Each service time controller affects the requests' service time by deciding the number of concurrently served requests and informing the load balancer. The service time setpoint is the same for all replicas, which ensures fairness among the requests.

Finally, we desire the overall infrastructure to follow a global setpoint that prescribes statistics on the response times (e.g., the 95th percentile of the response times of all the replicas should follow a given setpoint). A third controller is then responsible for determining the two setpoints of the other controllers – the setpoint on waiting and service time – dynamically. We refer to the third controller as the *top-level controller*.

In the following, we discuss the design of each of these three controllers in a separate section. Section III-A describes the waiting time controller, Section III-B details the service time controller, Section III-C discusses the top-level controller, and, finally, Section III-D describes additional implementational aspects, including our anti-windup strategy.

A. Waiting Time Control Design

The waiting time controller is located in the load balancer, and uses the decision of serving optional content or not as an actuator to steer the average waiting time $\bar{t}_w$ to its setpoint $r_{\bar{t}_w}$. Feedback is achieved by directly measuring the waiting time $t_w(\rho)$ of each request ρ right before it is being routed. The controller then attaches a flag, $o(\rho) \in \{0, 1\}$, to the request based on this measurement, where $o(\rho) = 1$ indicates that optional content should be computed and served. For each request ρ, the decision on the value of $o(\rho)$ is based on a threshold ψ_t on the waiting time. The threshold ψ_t is

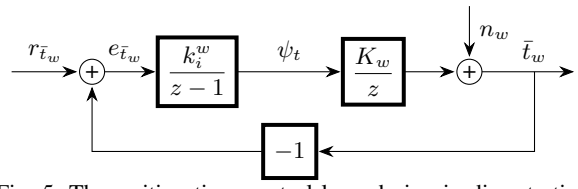

Fig. 5: The waiting time control loop design in discrete time.

updated periodically, and denoting with k the time interval $[k \cdot t, (k+1) \cdot t)$, and with $\psi_t(k)$ the value of the threshold in said time interval, the controller behaves according to Equation (1). If the measured waiting time is higher than the threshold, then no optional content is served. Otherwise, the request is served with optional content.

$$\begin{aligned} t_w(\rho) > \psi_t(k) \quad &\Longrightarrow \quad o(\rho) = 0 \\ t_w(\rho) \leq \psi_t(k) \quad &\Longrightarrow \quad o(\rho) = 1 \end{aligned} \tag{1}$$

In stationarity, the average waiting time $\bar{t}_w$ will stay in the vicinity of the threshold ψ_t. However, the exact relation will depend on the current state of the system. This motivates the need for a feedback controller, which dynamically changes the threshold ψ_t such that $\bar{t}_w$ always follows the setpoint $r_{\bar{t}_w}$. In order to design this controller, a model describing the dynamics from ψ_t to $\bar{t}_w$ is required. As a simplification, if the controller that determines the value of ψ_t is designed to be slow in comparison with the threshold algorithm specified in Equation (1), then $\bar{t}_w$ can be approximated as always staying close to the threshold ψ_t. This is a reasonable approximation, since Equation (1) is very effective at keeping the request waiting times close to the threshold ψ_t, thanks to its event-driven execution. Using this reasoning, the dynamics from ψ_t to $\bar{t}_w$ can be modeled in discrete time as:

$$\bar{t}_w(k+1) = K_w \psi_t(k) + n_w, \tag{2}$$

where K_w is a gain close to 1 and n_w is a stochastic disturbance related to the non-deterministic nature of the arrivals to the load balancer and service times in the replicas. We here use control-theoretical design principles [2] and compute the *Z-transform* of Equation (2). The pulse transfer function $H_w(z)$ from ψ_t to $\bar{t}_w$ then becomes

$$H_w(z) = \frac{K_w}{z}. \tag{3}$$

In order to achieve zero stationary error with respect to the setpoint $r_{\bar{t}_w}$, integral action is required in the controller. A pure integral controller is here used,

$$C_w(z) = \frac{k_i^w}{z-1}, \tag{4}$$

where k_i^w is the integral gain to be determined. The proposed design for the waiting time control loop is shown in the block diagram in Figure 5.

Closing the loop with the proposed controller leads to the following characteristic equation for the closed loop system:

$$z^2 - z + K_w k_i^w = 0. \tag{5}$$

We desire to place the poles of the closed-loop system system within the unit circle for stability, and on the positive real

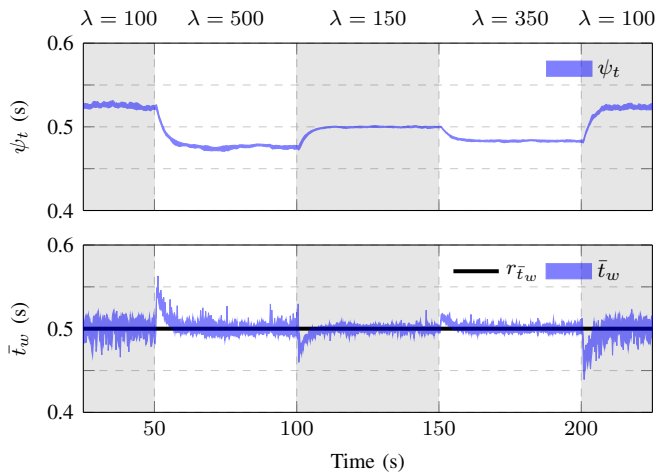

Fig. 6: 95% confidence intervals from 20 runs on thresholds ψ_t (upper) and average waiting times $\bar{t}_w$ (lower) using the proposed waiting time controller in the load balancer. The setpoint on the mean waiting time $r_{\bar{t}_w}$ is 0.5

axis for a desirable transient behavior. This corresponds to the following desired characteristic equation

$$z^2 - (a+b)z + ab = 0, \tag{6}$$

where $0 \leq a, b \leq 1$, for the desired locations of the poles. Comparing coefficients in Equations (5) and (6) results in the following system of equations:

$$\begin{aligned} a + b &= 1, \\ K_w k_i^w &= ab. \end{aligned} \tag{7}$$

Simulations suggest that the pole placement $b = 0.92$, $a = 1-b = 0.08$ gives a good transient behaviour of the closed-loop system, in terms of disturbance rejection and response speed of the controller. Using (7), this implies that we should choose $k_i^w = 0.07/K_w$. Since we expect that $K_w \approx 1$, a reasonable choice for the integrator gain is $k_i^w = 0.07$.

The robustness of this design choice can be tested by using Equation (5) to examine for what values of the process gain K_w the closed-loop system remains asymptotically stable (i.e. when the poles are within the unit circle). Inserting $k_i^w = 0.07$ in (5), the closed loop system remains stable for $K_w \leq 14.3$. Since K_w is expected to have a value close to one, this implies a very robust control design.

An example showing the control action of the waiting time controller when using the proposed architecture during different workloads is presented in Figure 6. The setup is the same as for the comparison made in Figure 3, and the 95% confidence intervals are based on 20 runs. Here we see how the waiting time controller dynamically adjusts the threshold ψ_t with the changing workload such that the mean waiting time $\bar{t}_w$ follows the setpoint r_{t_w}, which has a static value of 0.5 in this example.

B. Service Time Control Design

Each replica has a service time controller, responsible for keeping the average service times (for requests serving optional content) $\bar{t}_s$ at the setpoint $r_{\bar{t}_s}$. The value used for feedback is thus the average value of the service times of all completed requests during each time interval k. The service time controller

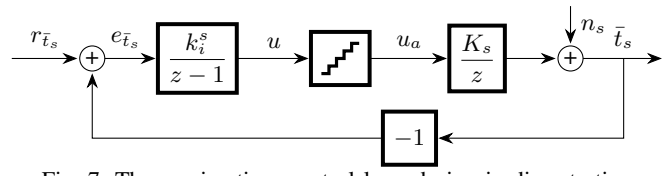

Fig. 7: The service time control loop design in discrete time.

can affect the service times by changing the integer number of simultaneous requests $u_a \in \mathbb{Z}^+$ to run. However, the control signal $u \in \mathbb{R}^+$ computed by the controller is a non-negative real-valued number, which thus has to be quantized as $u_a = \lceil u \rceil$ before it can be actuated (the ceiling function is used here for the quantization).

To be able to assess the behaviour of the control strategy and theoretically analyze the system, we need a model relating u to $\bar{t}_s$. In the modeling process, the quantization effects are neglected, i.e. we assume $u_a = u$. Assuming that all forwarded requests to the replica will be served concurrently, and assuming that a change in u is reflected very fast in $\bar{t}_s$, we can use the following simple discrete-time model:

$$\bar{t}_s(k+1) = K_s u(k) + n_s, \tag{8}$$

where K_s is a gain relating the number of simultaneous requests to the average service times and n_s is a stochastic disturbance describing the variance in the service times. Note that this model (8) has the same structure as the waiting time model (2). As a result, a majority of the analysis in Section III-A can be re-used. However, in this case, the gain K_s can not be assumed to always stay close to 1. In fact, K_s is directly related to the speed of the replica, which can vary greatly with time and also be different among the different replicas. This gain thus has to be estimated by the replica controller. The estimation $\hat{K}_s$ is performed, in each replica, using an exponentially weighted moving average filter:

$$\hat{K}_s(k+1) = (1-\alpha)\hat{K}_s(k) + \alpha \frac{\bar{t}_s(k)}{u_a(k)}, \tag{9}$$

where α is a design parameter, here set to $\alpha = 0.5$ (based on preliminary experiments). Using this estimated gain and the controller design in Section III-A, and in particular the results from Equation (7), the following adaptive integral controller is proposed for the service time:

$$k_i^s = \frac{c(1-c)}{\hat{K}_s}. \tag{10}$$

The location of the slowest closed-loop pole $0 \leq c \leq 1$ is a trade-off between rejection of control errors caused by changes in server speed, robustness to estimation errors in $\hat{K}_s$, quantization errors, and rejection of the stochastic noise n_s. Taking these elements into consideration, we place the pole in $c = 0.8$, which results in a stable closed-loop system as long as $\hat{K}_s \geq K_s/6.25$. We consider this a robust enough design. The adaptive integral control design is thus:

$$k_i^s = \frac{0.16}{\hat{K}_s}. \tag{11}$$

The block diagram of the complete service time model and control design is shown in Figure 7.

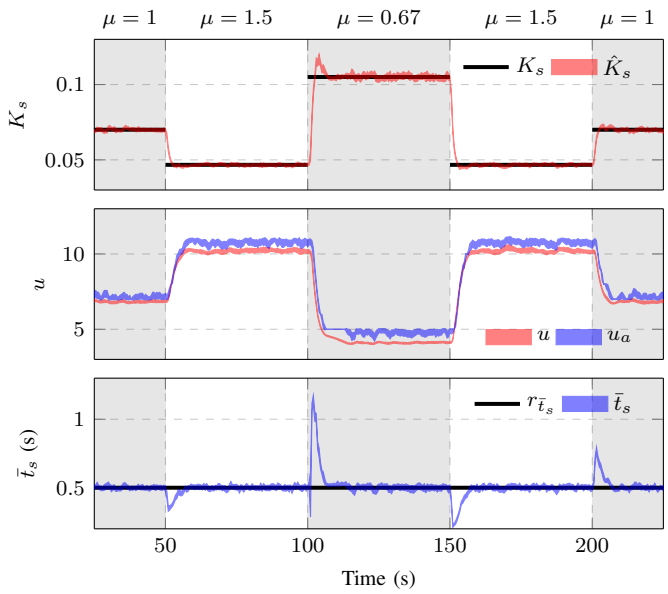

Fig. 8: 95% confidence intervals from 20 runs on estimated gain $\hat{K}_s$ (upper), service time control signals u and u_a (middle) and average service times $\bar{t}_s$ (lower) using the proposed service time controller in one replica. The true gain values K_s (upper) and the setpoint on average service time $r_{\bar{t}_s} = 0.5$ (lower) are plotted for reference.

The actuation of the quantized control signal u_a, representing the number of simultaneous requests to run in a replica, is as previously mentioned performed using piggy-backing. In more detail, the following steps are involved:

1) At startup, both u and u_a are initialized to zero.
2) The control signal u is updated every time interval k according to the scheme in Figure 7.
3) At every request completion, a new value of u_a is computed: $u_a^{new} = \lceil u \rceil$. The difference $u_a^{'} = u_a^{new} - u_a$ is determined and the old value of u_a is updated to u_a^{new}.
4) The response of the completed request is sent back to the load balancer, using piggy-back to send also $1 + u_a^{'}$, the number of new requests that the replica wants to serve.

The steps above constitute the actuation of u_a, completing the control design. The mentioned design ensures stability, tackles robustness issues, and guarantees a fast convergence, as shown in the experimental validation presented in Section IV.

An example showing the control action and gain estimation of the service time controller when using the proposed architecture is presented in Figure 8. The setup is the same as for the comparison made in Figure 3, but here we instead vary the service times for both optional and mandatory content by scaling them by a factor $1/\mu$ during different time intervals of 50 s. The service time controller is able to efficiently estimate the gain K_s and dynamically adjust the number of concurrently served requests u_a such that the mean service time $\bar{t}_s$ follows the setpoint r_{t_s}, which has a constant value of 0.5 in this example.

C. Top-Level Control Design

To ensure that the global setpoint on respose times is followed, we employ a top-level controller. This controller decides the setpoints of the other two controllers, i.e., the

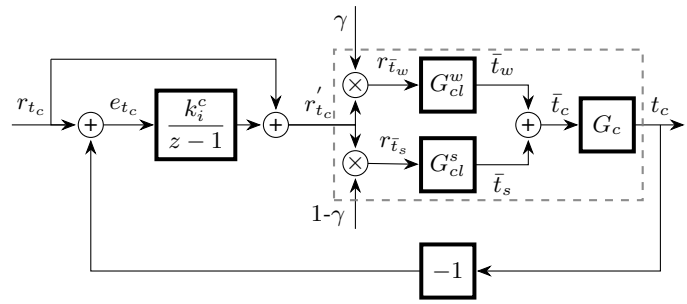

Fig. 9: The complete control loop design in discrete time. G_C represents the relation between average response times $\bar{t}_c$ and chosen statistical measure for feedback t_c.

setpoint on the waiting and on the service time, respectively $r_{\bar{t}_w}$ and $r_{\bar{t}_s}$. The setpoint r_{t_c} prescribes a statistical measure obtained from the vector of response times, e.g. the 95th percentile. The top-level controller receives the measured value of the same statistic of the response times t_c as a feedback signal. The controller then dynamically adjusts the setpoints $r_{\bar{t}_w}$ and $r_{\bar{t}_s}$. While the top-level controller should react to persistent errors in the response times, we also wish to avoid being too sensitive to outliers and transient errors in the inner control loops. This motivates the choice of a top-level controller which is slow with respect to the dynamics of the waiting- and service-time control loops. We can then re-use again the analysis from Section III-A, and propose the following simple integral controller:

$$C_c(z) = \frac{k_i^c}{z-1}. \tag{12}$$

The integral gain k_i^c is chosen as a sufficiently small value. Studying the behavior of the system, we selected $k_i^c = 0.01$.

Using this controller's output signal, we change both the other setpoints simultaneously. We specify a fixed ratio $\gamma \in [0, 1]$, which divides the total response time into a fraction γ (due to the waiting time) and $1-\gamma$ (due to service time). A block diagram of our proposed design for the top-level controller is shown in Figure 9. The dashed area in the figure represents the plant to control, while the rest is the top-level controller. In the plant, the inner control loops (Sections III-A and III-B) are represented by the blocks G_{cl}^w and G_{cl}^s for the waiting time- and service time control loop respectively. These control loops are given in detail in Figures 5 and 7. The block G_c represents the conversion block that translates average response times into the statistic that is used as a feedback signal.

In the real system, the top-level controller will be located in the load balancer. From there, updated setpoints on service time can be propagated to the replicas using the requests.

The design parameter γ decides which part of the system the requests will spend most time in, and can be tuned to handle uncertainties in the system. With γ close to one the requests will spend most time waiting in the queue, while the replicas will serve fewer requests concurrently. This is beneficial for the overall predictability of the response times in the case when most uncertainty lies in the service times. The opposite case with γ close to zero is beneficial when most uncertainty lies in the arrival rate of incoming requests.

D. Implementation Aspects

The solution proposed in this paper is capable of handling graceful degradation for a wide range of arrival rates. However, it clearly cannot cover all the possible arrival rates, as there are limitations (on the amount of simultaneous requests that can be served in general terms), imposed by the capacity of the replicas. Computing these limitations is fairly straightforward.

If n replicas serve only mandatory content, with a service time of t_m per request, we can compute the upper bound on the overall rate of requests that can be served by the system (with full degradation) as $\mu_{\max} = n/t_m$. In turn, this means that arrival rates $\lambda > \mu_{\max}$ will lead to over-utilization and instability. In this case, it is possible to detect that additional replicas should be started and an auto-scaler can efficiently take care of ensuring a viable operation region. The design of such auto-scaling policy is beyond the scope of this paper. Alternatively, over-utilization can be handled using admission control in the central queue.

During periods of abnormally small workloads, the response times will stay below the setpoint, even though optional content is served to all requests. This poses no issue to the user, but the controllers in the system will see a persistent error in response time, and would ideally like to throttle the throughput further by serving more optional content and more requests concurrently in the replicas. However, since it is not possible to serve more than 100% optional content and route more requests if the central queue is empty, the control signals will be saturated and unable to eliminate the error in response time. Controllers with integral action which experience persistent control errors under saturation are prone to *integrator wind-up*, a well known phenomenon in control theory [2]. The effect of integrator wind-up is that the controller will be unresponsive for a period of time when returning to normal workloads, which of course is unacceptable. Being a well-studied problem however, there exists several efficient algorithms in the control literature for removing wind-up from controllers, and the implementation done in our simulator features anti-windup.

Another aspect to consider when implementing strategies for load-balancing and graceful degradation is how the computational time needed to compute the control decisions scale with growing arrival rates and number of replicas. The controllers presented in Section III update their decisions based on a fixed sampling period. This means that their computational time is unchanged with respect to the arrival rate. Despite this, some logic has to be executed on a per-request basis (e.g. the decision on optional content, a single comparison of two floating point numbers). The computation that is done per request is in all cases simple, and has negligible execution times. The most expensive computation done on a per-request basis is sorting of the list with number of desired requests for each replica. The time it takes from when a request sends its desired new incoming request value to the time it actually gets forwarded new requests is negligible, and when request are routed to the replica, the corresponding element is removed from the vector that should be sorted. Given the speed of other components in the system, it is unlikely that the list contains demands from more than one replica at any given time, which makes the sorting operation negligible in terms of time complexity.

IV. Experimental Validation

This section presents our results. We validate our control strategy using the open source Python-based brownout simulator[2], built to mimic the behavior of cloud applications [27] and described in Section IV-A. We present the results obtained with the new architecture proposal in Section IV-B.

A. The simulator

The simulator defines the concepts of *Client*, *Request*, *Replica*, *Replica Controller*, and *Load Balancer*. Clients issue requests to be served by a replica. Clients can behave according to any inter-arrival time distributions and both according to the open-loop or to the closed-loop client model [1, 39]. In the closed-loop model, clients wait for a response and issue a new request only after some think time. In the open loop model, clients do not wait and instead issue new requests with a specific request rate. Being better at modelling a large number of independent users, we performed the evaluation with open-loop clients.

For each request, the simulator computes the service time. The time it takes to serve requests with only the mandatory or with the optional content in addition to the mandatory one are computed as random variables, with normal distributions, whose mean and variance are based on profiling data from the execution of experiments on a real machine [27].

Replicas implement a replica controller, that takes care of selecting – for each request – when to serve optional content. In the simulator, we used the replica controller described in [35] and used the suggested tuning parameters. For the control strategy presented in Section III-B, we use a sampling period of 0.25 s. The controller code developed in the simulator can be directly plugged into brownout-aware applications like RUBiS[3] and RUBBoS[4].

B. Experimental Results

To evaluate the predictability of our solution and compare it to the state of the art, we run simulations of 100 randomized scenarios in sequence, each lasting 50s. We then aggregate the results on response times for all the requests in all the scenarios. For the request generation, we use the open-loop client model and the same random seed generator, ensuring that the same number of requests are generated in all the scenarios and that the throughput of the cloud application is the same across the experiments, irrespective of the strategy used.

We use a fixed setpoint $r_{t_c} = 1$s on the 95th percentile of the response times throughout all scenarios. For each scenario, we randomize the number of replicas n, the average service times t_o (optional content) and t_m (mandatory content) for each individual replica (with the variance fixed to 0.01 s^2 and 0.001s^2 respectively), the number of concurrently running requests M_C (i.e., roughly the number of threads that replicas use to serve requests) and the expected optional content ratio θ.

[2]https://github.com/cloud-control/brownout-lb-simulator
[3]https://github.com/cloud-control/brownout-rubis
[4]https://github.com/cloud-control/brownout-rubbos

TABLE I: Bounds on randomized scenario parameters.

Parameter	Min	Max
n	3	10
t_o $[10^{-2}s]$	1	4
t_m $[10^{-4}s]$	2	8
θ	0.1	0.9
M_C	5	30

The values are sampled from uniform probability distributions, with bounds in Table I.

The arrival rate for each scenario is set to

$$\lambda = n\left(\theta \cdot \frac{1}{\bar{t}_o} + (1-\theta) \cdot \frac{1}{\bar{t}_m}\right), \tag{13}$$

where $\bar{t}_o$ and $\bar{t}_m$ are the average service times for optional and mandatory content respectively over the replicas (replicas can be different in their speed). We specify the arrival rate to avoid degenerate scenarios where the system either becomes unstable or where the workload becomes too low – assuming that an auto-scaler is in charge of selecting a correct number of replicas to run in the system.

We compare our proposed solution to three alternative strategies for the same 100 scenarios. For our solution, we use the ratio parameter $\gamma = 0.9$ (i.e. that each request is supposed to spend 90% of its time in the waiting process and 10% of its time being served) as well as $\gamma = 0.7$, as we expect great variations in service times between each scenario. The other evaluated strategies are state of the art solutions from the literature [11, 28, 35], using the architecture in Figure 2. The evaluated strategies are:

ILAC-γ: The integrated load-balancing and service time control (ILAC) architecture of this paper, with the marked γ parameter. We use both $\gamma = 0.9$ and $\gamma = 0.7$.

Brownout^{CC} + EPBH: A solution that employs cascaded control, Brownout^{CC} [35], paired with a brownout-aware weighted probability algorithm for load balancing (EPBH) [11].

Brownout^{CC} + SQF: The Brownout^{CC} controller, with the SQF algorithm for load balancing.

Brownout + EPBH: The original brownout controller [27], using the EPBH weighted probability algorithm for load balancing.

To evaluate the predictability of each strategy, we measure the Integrated Absolute Error (IAE) of deviations from the setpoint on the 95th percentile of response times. Given a sampling interval of length h, we compute the IAE as:

$$\text{IAE} := h \sum_k |r_{t_c}(k) - t_c(k)|, \tag{14}$$

where the summation is done over all sampling intervals k of the experiment. To complement this metric, we also record the standard deviation of the overall response times and the maximum recorded response time.

The results of the experiment for each strategy are summarized in Table II, along with Cumulative Distribution Functions of the optional content response times in Figure 10. Comparing the results of ILAC-γ for $\gamma = 0.9$ with $\gamma = 0.7$ indicates that a large value of γ indeed was favourable in the experiment.

TABLE II: Results from the experiment.

Strategy	IAE [s]	Standard Deviation [s]	Max Response Time [s]
ILAC-0.9	134.4	0.0953	1.41
ILAC-0.7	254.9	0.1412	2.36
Brownout^{CC} + EPBH	423.3	0.2640	2.58
Brownout^{CC} + SQF	823.1	0.2961	3.25
Brownout + EPBH	10980	1.3577	7.27

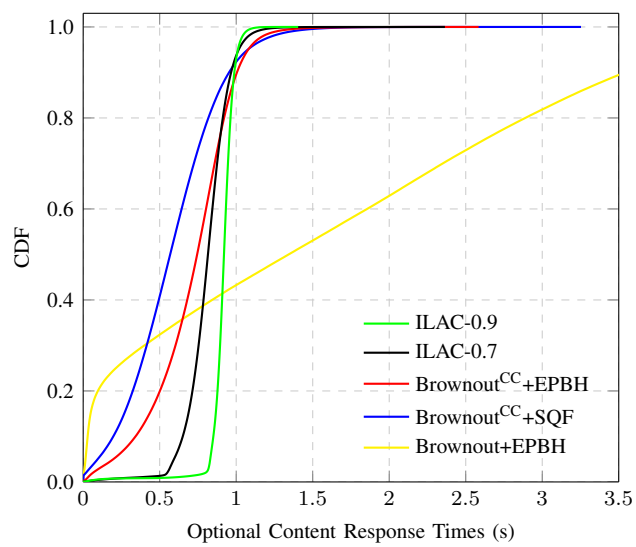

Fig. 10: Cumulative Distribution Function (CDF) of response times for all requests with optional content.

Still, the ILAC-γ significantly outperforms the other considered strategies in both cases. The closest competitor, Brownout^{CC} combined with EPBH, has a roughly 3 times larger IAE value that ILAC-0.9. The corresponding factor to the Brownout^{CC} + SQF strategy is roughly 6, and over 80 for the Brownout + EPBH strategy. The superior predictability of the proposed ILAC-γ strategy is also reflected in the overall standard deviations and maximum recorded response times, with the maximum response time for ILAC-0.9 being 1.41 seconds. The results show the effectiveness of our proposal and highlight the problem of co-design with the standard architecture in Figure 2, where the efficiency of the EPBH load-balancing alternative varies greatly with the choice of the controller used for graceful degradation.

The performance of the evaluated strategies are also exemplified in Figure 11, which shows averaged values of the 95th percentile of response times from 20 runs of 5 of the 100 scenarios. The parameter set for each scenario is given in Table III. We see in in the figure that the performance of the Brownout^{CC} + SQF and Brownout^{CC} + EPBH strategies are heavily dependent on the given scenario, whereas the proposed strategy keeps a high predictability regardless of the parameters used for the simulations. This robustness clearly highlights the benefits of the architecture proposed in Figure 4 combined with a control-theoretical design approach for the decision-making.

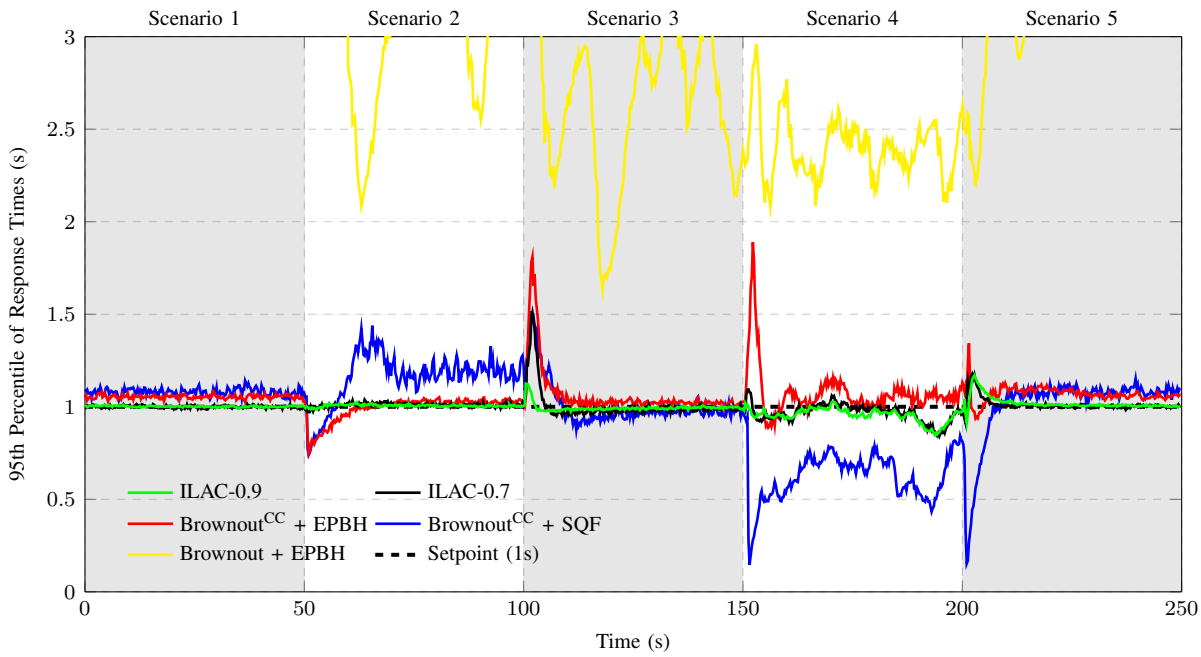

Fig. 11: Averaged values of the 95th percentile of response times from 20 runs of 5 selected scenarios. The parameter sets of each scenario is given in Table III.

TABLE III: Parameters of 5 selected scenarios (out of the 100 tested).

Scenario	#1	#2	#3	#4	#5
n	9	6	4	6	9
λ [s^{-1}]	570	890	330	310	570
$\bar{t}_o$ [$10^{-2}s$]	2.5	2.2	2.7	2.3	2.5
$\bar{t}_m$ [$10^{-4}s$]	5.4	4.3	6.3	4.6	5.4
θ	0.62	0.29	0.43	0.84	0.62
M_C	11	13	15	29	11

V. Related work

Building distributed systems that offer guarantees on their timely execution while the system is subject to *uncertainty* and *changes* is a challenging task. Bounding latencies is of utmost importance, but this is quite difficult in the presence of changes [5, 6, 14, 24, 25, 42]. Changes are unpredictable, they can be dramatic, and they can include malfunctioning [23], slow down [12], failures [18], and much more. Graceful degradation [31] is then introduced into the runtime system, to handle these changes and guarantee performance in the presence of uncertainty. This paper shows that graceful degradation and load-balancing can interfere with one another. We focus on a unified solution, to avoid this interference.

In replicated cloud services, load balancers have a crucial role for ensuring resilience and performance [3, 20]. Load-balancing algorithms can either be global (inter-data center) or local (intra-data center or cluster-level). Global load-balancing decides what data center to direct a user to, depending on geographic proximity [30] or price of energy [10]. Once a data center is selected, a local algorithm directs the request to a machine in the data center. Our contribution is of the local type.

Various local load-balancing algorithms have been proposed. For non-adapting replicas, SQF has been considered very close to optimal, despite it using little information about the state of the replicas [19]. Previous results show that for self-adaptive, brownout replicas, SQF performs quite well [28], but can be outperformed by weight-based, brownout-aware solutions [11]. In this article, we improve on brownout-aware load balancing, by combining the load-balancing strategy with the graceful degradation decision, obtaining better performance in terms of variance of response times, and show improved performance, compared to previously developed algorithms.

VI. Conclusion

This paper proposes a new load-balancing architecture that combines the action of the load balancer with graceful degradation techniques like brownout or admission control. We have designed the system and synthesized the load balancing strategies. The advantage of the proposed solution lies in the interplay between the two control solutions. While in previous solutions the two different components – load-balancer and graceful degradation controller – could compete and generate oscillations in response times, our proposal does not suffer from this issue.

Our proposed architecture has an important tuning parameter: the percentage of time that should be spent waiting and in service for each request. Our experimental campaign showed that – irregardless of the selected percentage time – the response times using the proposed load-balancing strategy are much more predictable than with any other previously explored strategy. Their variance is in fact much smaller than with other strategies, and their maximum is much closer to the desired setpoint than if other strategies are used.

In the future, we plan to combine the proposed architecture with auto-scaling features, that trigger new replicas to be started or old replicas to be removed. We also envision using the architecture for fault detection and countermeasures.

References

[1] F. Alomari and D. A. Menasce. "Efficient Response Time Approximations for Multiclass Fork and Join Queues in Open and Closed Queuing Networks". In: *IEEE Trans. Parallel Distrib. Syst.* 25.6 (June 2014).

[2] K.-J. Åström and B. Wittenmark. *Computer-Controlled Systems*. 3rd ed. Mineola, NY: Dover Publications Inc., 2011.

[3] L. A. Barroso and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool, 2009.

[4] N. Bencomo and A. Belaggoun. "A world full of surprises: bayesian theory of surprise to quantify degrees of uncertainty". In: *36th International Conference on Software Engineering, ICSE14, Companion Proceedings*. 2014, pp. 460–463.

[5] M. Björkqvist, N. Gautam, R. Birke, L. Chen, and W. Binder. "Optimizing for Tail Sojourn Times of Cloud Clusters". In: *IEEE Transactions on Cloud Computing* 6.1 (2018), pp. 156–167.

[6] M. Björkqvist, R. Birke, and W. Binder. "Resource management of replicated service systems provisioned in the cloud". In: *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*. 2016, pp. 961–966.

[7] D. Breitgand and A. Epstein. "Improving consolidation of virtual machines with risk-aware bandwidth oversubscription in compute clouds". In: *Proceedings of the IEEE INFOCOM 2012, Orlando, FL, USA, March 25-30, 2012*. 2012, pp. 2861–2865.

[8] A. Diaconescu, K. L. Bellman, L. Esterle, H. Giese, S. Götz, P. R. Lewis, and A. Zisman. "Architectures for Collective Self-aware Computing Systems". In: *Self-Aware Computing Systems.* 2017, pp. 191–235.

[9] S. Ding, S. Gollapudi, S. Ieong, K. Kenthapadi, and A. Ntoulas. "Indexing strategies for graceful degradation of search quality". In: *ACM SIGIR conference on Research and development in Information Retrieval*. ACM. 2011, pp. 575–584.

[10] J. Doyle, R. Shorten, and D. O'Mahony. "Stratus: Load Balancing the Cloud for Carbon Emissions Control". In: *TCC* 1.1 (2013).

[11] J. Dürango, M. Dellkrantz, M. Maggio, C. Klein, A. V. Papadopoulos, F. Hernández-Rodriguez, E. Elmroth, and K.-E. Årzén. "Control-theoretical load-balancing for cloud applications with brownout". In: *53rd IEEE Conference on Decision and Control, CDC 2014, Los Angeles, CA, USA, December 15-17, 2014*. CDC14. 2014, pp. 5320–5327.

[12] N. Fallahi, B. Bonakdarpour, and S. Tixeuil. "Rigorous Performance Evaluation of Self-Stabilization Using Probabilistic Model Checking". In: *SRDS*. 2013.

[13] A. Filieri et al. "Control Strategies for Self-Adaptive Software Systems". In: *TAAS* 11.4 (2017), 24:1–24:31.

[14] S. Ghahremani, H. Giese, and T. Vogel. "Efficient Utility-Driven Self-Healing Employing Adaptation Rules for Large Dynamic Architectures". In: *2017 IEEE International Conference on Autonomic Computing (ICAC)*. 2017, pp. 590–68.

[15] A. Greenberg, J. Hamilton, D. Maltz, and P. Patel. "The cost of a cloud: research problems in data center networks". In: *ACM SIGCOMM computer communication review* 39.1 (2008), pp. 68–73.

[16] D. Grimes, D. Mehta, B. O'Sullivan, R. Birke, L. Chen, T. Scherer, and I. Castineiras. "Robust Server Consolidation: Coping with Peak Demand Underestimation". In: *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 2016, pp. 271–276.

[17] J. Grohmann, N. Herbst, S. Spinner, and S. Kounev. "Self-Tuning Resource Demand Estimation". In: *2017 IEEE International Conference on Autonomic Computing*. 2017, pp. 21–26.

[18] Z. Guo et al. "Failure recovery: when the cure is worse than the disease". In: *HotOS*. 2013, pp. 8–14.

[19] V. Gupta, M. Harchol Balter, K. Sigman, and W. Whitt. "Analysis of Join-the-shortest-queue Routing for Web Server Farms". In: *Perform. Eval.* 64.9-12 (2007), pp. 1062–1081.

[20] J. Hamilton. "On designing and deploying internet-scale services". In: *LISA*. USENIX, 2007, 18:1–18:12.

[21] J. Heo and T. Abdelzaher. "AdaptGuard: Guarding Adaptive Systems from Instability". In: *Proceedings of the 6th International Conference on Autonomic Computing*. 2009, pp. 77–86.

[22] M. Hoger and O. Kao. "Record Skipping in Parallel Data Processing Systems". In: *2016 International Conference on Cloud and Autonomic Computing*. 2016, pp. 107–110.

[23] A. Iosup, N. Yigitbasi, and D. Epema. "On the Performance Variability of Production Cloud Services". In: *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*. 2011, pp. 104–113.

[24] S. A. Javadi and A. Gandhi. "DIAL: Reducing Tail Latencies for Cloud Applications via Dynamic Interference-aware Load Balancing". In: *2017 IEEE International Conference on Autonomic Computing*. 2017.

[25] T. Kaler, Y. He, and S. Elnikety. "Optimal Reissue Policies for Reducing Tail Latency". In: *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. 2017, pp. 195–206.

[26] M. Kihl, A. Robertsson, and B. Wittenmark. "Control Theoretic Modelling and Design of Admission Control Mechanisms for Server Systems". In: *Networking 2004*. Ed. by N. Mitrou, K. Kontovasilis, G. N. Rouskas, I. Iliadis, and L. Merakos. 2004.

[27] C. Klein, M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez. "Brownout: Building More Robust Cloud Applications". In: *36th International Conference on Software Engineering*. ICSE14. Hyderabad, India: ACM, 2014, pp. 700–711.

[28] C. Klein, A. V. Papadopoulos, M. Dellkrantz, J. Dürango, M. Maggio, K.-E. Årzén, F. Hernández-Rodriguez, and E. Elmroth. "Improving Cloud Service Resilience Using Brownout-Aware Load-Balancing". In: *IEEE 33rd International Symposium on Reliable Distributed Systems*. SRDS14. IEEE Computer Society, 2014, pp. 31–40.

[29] A.-D. Lin, C.-S. Li, W. Liao, and H. Franke. "Capacity Optimization for Resource Pooling in Virtualized Data Centers with Composable Systems". In: *IEEE Transactions on Parallel and Distributed Systems* 29.2 (2018), pp. 324–337.

[30] M. Lin, Z. Liu, A. Wierman, and L. L. H. Andrew. "Online algorithms for geographical load balancing". In: *IGCC*. IEEE, 2012.

[31] Y. Lin and S. S. Kulkarni. "Automated Multi-graceful Degradation: A Case Study". In: *SRDS*. 2013.

[32] M. Litoiu, M. Shaw, G. Tamura, N. M. Villegas, H. A. Müller, H. Giese, R. Rouvoy, and É. Rutten. "What Can Control Theory Teach Us About Assurances in Self-Adaptive Software Systems?" In: *Software Engineering for Self-Adaptive Systems III. Assurances*. 2013, pp. 90–134.

[33] M. Maggio, T. F. Abdelzaher, L. Esterle, H. Giese, J. O. Kephart, O. J. Mengshoel, A. V. Papadopoulos, A. Robertsson, and K. Wolter. "Self-adaptation for Individual Self-aware Computing Systems". In: *Self-Aware Computing Systems.* 2017, pp. 375–399.

[34] T. Neumann. "Query simplification: graceful degradation for join-order optimization". In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM. 2009, pp. 403–414.

[35] T. Nylander, C. Klein, K.-E. Årzén, and M. Maggio. "Brownout^{CC}: Cascaded Control for Bounding the Response Times of Cloud Applications". In: *2018 American Control Conference*. 2018.

[36] P. Östberg et al. "Reliable capacity provisioning for distributed cloud/edge/fog computing applications". In: *2017 European Conference on Networks and Communications (EuCNC)*. 2017, pp. 1–6.

[37] J. Perez, R. Birke, and L. Chen. "On the latency-accuracy tradeoff in approximate MapReduce jobs". In: *IEEE Conference on Computer Communications*. 2017, pp. 1–9.

[38] A. Robertson, B. Wittenmark, and M. Kihl. "Analysis and design of admission control in Web-server systems". In: *Proceedings of the 2003 American Control Conference, 2003.* Vol. 1. 2003, 254–259 vol.1.

[39] B. Schroeder, A. Wierman, and M. Harchol-Balter. "Open Versus Closed: A Cautionary Tale". In: *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*. NSDI'06. San Jose, CA: USENIX Association, 2006.

[40] H. Sun, R. Birke, W. Binder, M. Björkqvist, and L. Chen. "Acc-Stream: Accuracy-Aware Overload Management for Stream Processing Systems". In: *2017 IEEE International Conference on Autonomic Computing (ICAC)*. 2017, pp. 39–48.

[41] L. Tomás and J. Tordsson. "Cloud Service Differentiation in Overbooked Data Centers". In: *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. 2014, pp. 541–546.

[42] C. Wang, B. Urgaonkar, A. Gupta, L. Chen, R. Birke, and G. Kesidis. "Effective Capacity Modulation as an Explicit Control Knob for Public Cloud Profitability". In: *2016 IEEE International Conference on Autonomic Computing (ICAC)*. 2016, pp. 95–104.

[43] J. Xue, R. Birke, L. Chen, and E. Smirni. "Managing Data Center Tickets: Prediction and Active Sizing". In: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2016, pp. 335–346.