Towards Coordinated Autoscaling and Application Brownout at the Orchestrator Level

Ivan Kotegov and Antonio Filieri

Imperial College London, UK

Abstract. Modern cloud applications are expected to continuously provide adequate performance, withstanding changing workloads, heterogeneous hardware, and unpredictable infrastructure failures. Autoscaling can automatically provision resources to match performance goals but may suffer from slower reaction times and risks of over-provisioning. Brownout mechanisms, on the other hand, empower applications with the ability to quickly dim out optional features, freeing computational resources to serve core functionalities with the desired performance level. However, modifying an application to include brownout capabilities may require invasive changes to the codebase and the need to expose ad-hoc interfaces to coordinate the interaction of the brownout dimmers and autoscaling actions, avoiding interferences that may destabilize the system.

In this paper, we report on our preliminary results on the design of an application-agnostic control theoretical solution to integrate scaling and dimming capabilities at the orchestrator level. We implemented a prototype of our controller on top of Kubernetes and HAProxy to empower generic applications with coordinated autoscaling and brownout capabilities by dynamically controlling the number of active replicas and per-user access to optional API endpoints.

1 Introduction

Modern cloud applications are expected to adapt their behavior and resource allocations to continuously provide the desired quality of service in spite of unpredictable changes in their workloads and execution environments.

Most adaptation techniques rely on feedback loops to control and mitigate the effects of external phenomena on the performance of the controlled application. Feedback loops continuously measure the evolution of relevant quality figures of the running system, triggering adaptation actions when these measures deviate from acceptable ranges. Adaptation decisions, such as provisioning of additional resources or disabling optional software features, can be drawn based on a variety of methods, from casting the decision as an optimization problem to using machine learning to predict the most appropriate reactions [1]. Different methods require more or less accurate models of the relevant system behaviors (e.g., abstracted as queuing networks or difference equations), can adapt for a single or multiple goals and actuators, and provide different guarantees on the effectiveness, stability, and robustness of their decision processes [2].

Among the different adaptation paradigms considered in the field, in this paper we focus on autoscaling and brownout. Autoscaling aims at automatically allocating and deallocating computational resources as required by the application to match its performance goals. Microservices or FaaS are typical architectures that enable an application to increase its capacity by instantiating additional replicas. Brownout [3] is instead a form of graceful degradation where an application may adaptively disable optional features to reduce its computational fingerprint, releasing resources to serve core features at the required performance level. Brownout has also been used to improve the energy footprint of cloud applications [4], and resource utilization in cloud applications [5].

Differently from autoscaling, current brownout controllers require ad-hoc changes to the application codebase, increasing maintainability costs and development complexity. Furthermore, while provably effective and stable in isolation, deploying these two techniques simultaneously on the same application may induce coupling effects, where the actions taken independently by the two controllers may antagonize each other, possibly leading to oscillations.

In this paper, we present preliminary findings on our development of a control-theoretical adaptation mechanism to coordinate autoscaling and brownout decisions at the orchestrator level, with the goal of controlling the response time of an application. To coordinate the two controllers' actions to keep the response time at the desired values, we propose using a mid-range control architecture [6]. This enables the use of computationally efficient and provably stable PI controllers, coordinated to take advantage of the distinct speed, accuracy, and strengths of both autoscaling and brownout. We report on preliminary experimental results in support of our control design via a prototype implementation on top of Kubernetes and HAProxy.

2 Background

Feedback control. Figure 1 represents a basic feedback loop. The controller C computes its control signal u as a function of the difference between the desired behavior – the *setpoint* (y_{sp}) – and the measured process behavior (y). This difference is called error (e). The process behavior is affected by both the control signal and external disturbances w. Even if the external disturbances are not explicitly modeled or only coarse information about them is available, their (relevant) effects on the process propagates through the measure y via the feedback loop, allowing the controller to reject the disturbances. Controllers

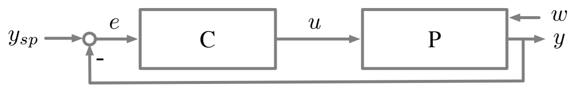

Fig. 1. Feedback control loop.

can be evaluated on several formal properties [7]. The three most relevant for this work are stability, settling time, and overshoots. A stable controller will eventually drive the process within a bounded distance from the setpoint, if feasible. Settling time is the time required to converge (up to a fixed accuracy) to the setpoint, while overshoots are related to transitory excessive reactions and should be limited.

PID controllers. Equation (1) formalizes the control law of a PID controller.

$$u(t) = \underbrace{K \cdot e(t)}_{\text{proportional}} + \underbrace{\frac{K}{T_i} \cdot \int e(\tau) d\tau}_{\text{integral}} + \underbrace{K \cdot T_d \cdot \frac{de(t)}{dt}}_{\text{derivative}} \tag{1}$$

where $e(t) = y_{sp}(t) - y(t)$ is the error, which the controller aims to minimize. A PID is tuned through the three parameters K, T_i, and T_d. K is the proportional gain. Higher values of K lead to "stronger" reactions, which may reduce the settling time of the closed loop, but could destabilize it. The integral term mitigates steady-state deviations form the setpoint that the proportional component cannot handle, it can, however, lead to overshoots. The integral action also helps to smooth the reaction to fast changes in the error. The derivative term can reduce the convergence time and increase stability, but it may induce the controlled system to follow error variations due to external disturbances. For our application, we will not use the derivative component ($T_d = 0$), preferring disturbance rejection over settling time. This setting is called the PI controller. PID controllers do not require an explicit analytical model of the process but can be tuned using human expertise or established heuristics [6]. While usually sub-optimally, PIDs can control several classes of nonlinear systems [6].

3 Coordinating Scaling and Dimming

Our goal is to consistently control the number of replicas allocated to an application (scaling) and the brownout of optional features (dimming). We will refer to the corresponding controllers as *scaler* and *dimmer*, respectively. Control should be placed at the orchestrator/load-balancer level, instead of modifying the application's logic, for a better separation of concerns. The control goal is to keep the average response time close to a prescribed setpoint.

Individually, both a scaler and a dimmer can be implemented using PI. The former actuated on the number of replicas, the latter regulating the rate at which access to optional features endpoints is allowed. However, if the two controllers act independently, they can interfere with one another, possibly leading to oscillations and destabilizing the system. For example, consider the measured response time exceeds the setpoint, the scaler can allocate an additional replica, while at the same time the dimmer may disable optional features, reducing computational demand. The reduced demand would trigger the scaler to deallocate excessive replicas, while the dimmer would at the same time restore optional features. In control terms, there are two actuators coupled on a single measure.

Mid-range control. PI(D)s are single-input single-output (SISO) controllers, i.e., manipulating one actuator to control one measurement, and cannot be easily extended to control multiple inputs or multiple outputs. We can observe that the dimmer and the scaler have different dynamics. Dimmer decisions can be enforced quickly (access control) and have high resolution, deciding with high precision the rate at which optional features should be served. However, dimming can compensate only moderate load variations, since core application features are always served. Scaling, on the other hand, can compensate for larger variations by provisioning additional resources, but its actions have lower resolution and take longer to actuate.

This situation lends itself to the design of a *mid-range control* architecture [6]. This architecture is described in Figure 2. The dimmer takes the measured response time and controls the rate at which optional features are served to reach the setpoint. The scaler, on the other hand, takes the control signal emitted by the dimmer and controls the number of provisioned replicas to keep the dimmer around the middle of its operational range (d_{sp}). When the error e exceeds the

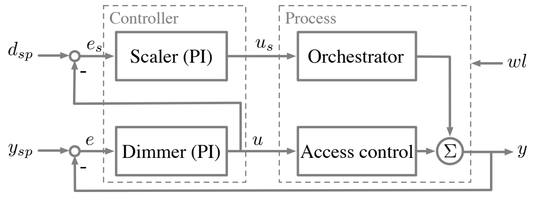

Fig. 2. Controller architecture.

capabilities of the dimmer, its control signal u is likely to saturate to the top or the bottom of its range (i.e., either serving optional features to every user or to none). This deviation of u from its mid-range d_{sp} will then trigger the reaction of the scaler, which will allocate or deallocate replicas to bring the dimmer back to d_{sp} where it can limit serving optional features as needed for the new number of replicas. Both the dimmer and the scaler are PI controllers.

4 Preliminary Results

Implementation. We implemented the control architecture of Section 3 by extending HAProxy-Ingress, an open-source Kubernetes Ingress controller. Dimming is implemented by generating and dynamically updating the HAProxy's configuration defining access control lists (ACLs) for optional application features. The dimmer control signal represents the maximum number of requests a user (uniquely identified by session id) can make over a 30 secs window before their requests to optional features get disabled. The range of the dimmer control signal is 1-1000. The scaler controls the number of pods in the cluster (1-6). Average response time is measured by HAProxy over the last 1024 requests.

Experiment. We adapted the JPetStore benchmark adding artificial optional features, similarly to [3]. The optional features simulate a random delay averaging 1 second. Kubernetes is deployed on a cluster of 6 i7-4790 workstations and uses Weave Net CNI. We used JMeter distributed testing with 6 instances to simulate users. Each simulated user generated a sequence of up to 1000 requests separated by a random delay between 1 and 100 milliseconds. Each user is randomly assigned a ratio between 0 and 50% denoting the proportion of optional features requested. The controllers have been tuned manually, with $K = 0.05$ and $T_i = 3$ for the dimmer and $K = 0.0025$ and $T_i = 1000$ for the scaler. The scaler uses a hysteresis of 2 minutes to allow for resource allocation and measurements update. The target mid-range for the dimmer is set to 600 ± 200.

Results. An example run is shown in Figure 3. Time is reported in steps of 15 seconds. The top subfigures shows the number of active user sessions over time, and the measured response time and setpoint, respectively. As observable in the figure, small variations in the workload are handled by the dimmer, without

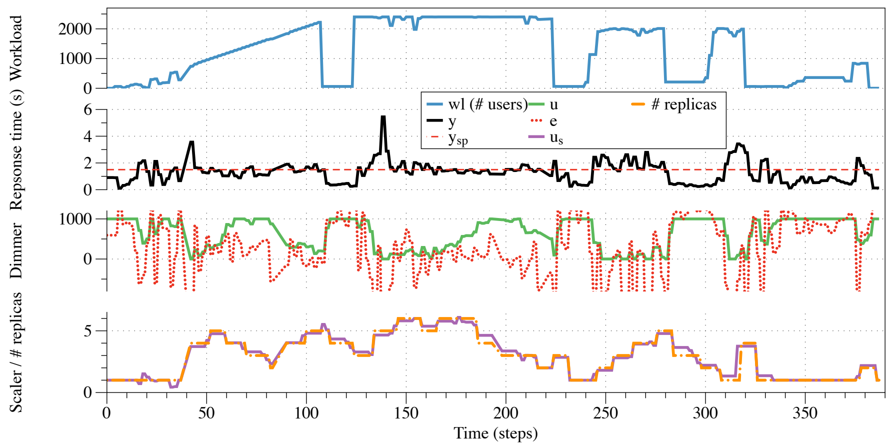

Fig. 3. Experiment results. From the top: workload wl in number of active user sessions; Response time (y) and its setpoint (y_{sp}); Dimmer control signal (u_d) and error (e_d); Scaler control signal (u_s) and corresponding number of replicas.

the intervention of the scaler (whose control signal is nonetheless tracking the deviation of u from its midrange). Larger variations and higher spikes (e.g., flash crowds) require the provisioning of more replicas, which is triggered when the dimmer signal u saturates towards 1; similarly, replicas are removed when u saturates towards 1000 (i.e., in our setup, serving optional features to all users). The two controllers coordinate their operations avoiding oscillations and bringing the average response time around or below the setpoint.

5 Conclusions

We presented preliminary results on the use of a mid-range control architecture to coordinate autoscaling and brownout at the orchestrator level. The use of PI controllers requires a few arithmetic operations to determine the control actions, avoiding the overhead of more complex strategies. However, in these experiments, the controllers have been tuned manually, which may require some expertise. Autotuning and adaptive PIs may provide better performance, in particular allowing to retuning the controllers to better fit the current workload trends, reducing settling time and overshooting phenomena [8].

References

[1] C. Qu et al. “Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey”. In: *ACM Comput. Surv.* 51.4 (July 2018).
[2] T. Chen et al. “A Survey and Taxonomy of Self-Aware and Self-Adaptive Cloud Autoscaling Systems”. In: *ACM Comput. Surv.* 51.3 (June 2018).
[3] C. Klein et al. “Brownout: Building More Robust Cloud Applications”. en. In: *ICSE*. Hyderabad, India: ACM Press, 2014, pp. 700–711.
[4] M. Xu et al. “Energy Efficient Scheduling of Cloud Application Components with Brownout”. In: *IEEE Trans. Sustain. Comput.* 1.2 (July 2016), pp. 40–53. arXiv: 1608.02707.
[5] C. Wang et al. “Effective Capacity Modulation as an Explicit Control Knob for Public Cloud Profitability”. In: *ACM Trans. Auton. Adapt. Syst.* 13.1 (May 2018), 2:1–2:25.
[6] K. J. Åström et al. *Advanced PID control*. Vol. 461. ISA, 2006.
[7] A. Filieri et al. “Control Strategies for Self-Adaptive Software Systems”. In: *ACM Trans. Auton. Adapt. Syst.* 11.4 (Feb. 2017), 24:1–24:31.

[8] M. Maggio et al. “Control Strategies for Predictable Brownouts in Cloud Computing”. en. In: *IFAC Proceedings Volumes*. 19th IFAC World Congress 47.3 (Jan. 2014), pp. 689–694.