UDC 004.75 https://doi.org/10.20535.2786-8729.5.2024.315877

METHOD OF HORIZONTAL POD SCALING IN KUBERNETES TO OMIT OVERREGULATION

Oleksandr Rolik *
National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine
http://orcid.org/0000-0001-8829-4645

Volodymyr Volkov
National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine
https://orcid.org/0009-0008-5325-0368

*Corresponding author: o.rolik@kpi.ua

This paper describes the method of omitting the over-regulation effect that occurs under certain conditions by horizontal pod autoscaling microservices in the container application orchestration system Kubernetes. The effect was initially observed only for long-term HTTP WebSocket sessions, where it led to excessive use of computing resources, which reduced the efficiency of IT infrastructure management, and caused service failure. It was found that the overregulation effect is reproduced not only for connections with long-term HTTP sessions, such as HTTP WebSocket, but also for shorter-term REST HTTP sessions in case of increased delay in the metric collection cycle used for horizontal pod autoscaling. It is assumed that this effect happens due to the approach of implementing horizontal scaling controllers similar to the principles of proportional regulators in systems with negative feedback from the theory of automation and control. It is proposed to extend one of the methods used for optimizing the proportional controller to the problem consisting of reducing the time delay between scaling metrics collecting and upscale applied by the controller in Kubernetes. The applied method demonstrated its effectiveness, therefore, within the same methodology, an experiment was conducted on using the proportional-integral-differential controller for automatic horizontal scaling of pods. The results obtained showed why the proportional-integral-differential controller is not widespread among the overviewed Kubernetes solutions for horizontal automatic scaling. An assumption was made about the limitations of studying the downscaling process in Kubernetes due to the need to consider the quality of service when stopping pods and the need to collect indicator metrics using quality-of-service object management tools such as ISTIO.

Key words: Kubernetes, Microservices, Horizontal Pod Autoscaling, Proportional Regulation, Cloud Computing

1. Introduction

There are several tools and platforms for managing (orchestration) microservice applications that use cloud computing operators data center resources. Cloud Native Computing Foundation (CNCF) has already made Kubernetes a de facto standard of a platform for running and deploying microservice applications [1]. This was achieved with open architecture and out of the box valuable microservice app management features. Kubernetes fulfills the basic requirements of continuous delivery and orchestration of the microservice application in private and public clouds.

Applications should be packaged in containers and organized in pods to be executed in Kubernetes. Pod is both a collection of containers with shared network, and a Kubernetes specification on how to run them. This makes it to be used in Kubernetes as an ephemeral execution primitive [2]. Kubernetes Documentation [3] extends this thesis with the statement that the pod is the minimum entity that could be managed for scaling the service capacity.

The microservices applications runtime in the cloud environments is billed for utilized resources. In the cloud Kubernetes autoscale ss tried to be minimized during the upscale to give not more then needed to handle the load spike. And during the load drop to downscale used resources soonest.

There are three types of autoscalers provided by Kubernetes: Horizontal Pod Autoscaler (HPA), Vertical Pod Autoscaler (VPA) and Cluster Autoscaler (CA) [4].

HPA supports high availability by adjusting the number of execution and pods based on various requirements and manifests [4]. When triggered, HPA creates new pods to share the workloads without affecting the sessions already instantiated to existing ones already running inside the cluster.

Vertical Pod Autoscaler directly changes the specifications, such as requested CPU or RAM resources not affecting the actual count of managed pods [4]. Therefore, it requires restarting of these pods and thus disrupts the continuity of applications and services.

Cluster Autoscaling increases the number of nodes when it is no longer possible to schedule pods on the existing ones. CA is looking as extender of the frame of resources, but actually modifying of the capacity of the application is happening with either HPA or VPA.

HPA allows to adjust dynamically the number of pod replicas based on observed CPU usage or other resource load indicators without the need to restart already running pods, while continuing to serve existing user sessions. This makes one the most preferable in the context of keeping user quality of service level by not dropping existent sessions under growing load at least during upscale process.

However, the [5] states that when managing scaling using HPA in a Kubernetes environment, the number of replicas could begin to fluctuate in some circumstances, i.e., a beating or rattling effect occurs (referred to as thrashing, or flapping). This leads to inefficient use of resources by excessive number of pods, as well as to a violation of the quality of services provided by applications installed on these pods. This happens if the cluster is out of capacity to upscale other vital pods due to reaching the possible scale limits of by the one overscaling. Kubernetes is the most well-known platform for managing containerized workloads [1]. The number of Kubernetes implementations exceeds the number of implementations of similar orchestrators [1]. So should be existent developing methods and tools that will eliminate or at least reduce the problems of inefficient HPA scaling management in Kubernetes. In order the increasing of the efficiency of using computing resources in a Kubernetes environment is a relevant task.

2. Literature review

In [6] has been proven that the main way to maintain the quality of services at an agreed level when the quality of service deteriorates due to increased user requests. It`s to add computing resources to those applications that provide this service. Of course, the management of the amount of resources provided to applications should be automatic. Methods of the resource VPA and HPA autoscaling implemented by Kubernetes [1] could solve this task supporting both updating and applying changes in requirements for memory (RAM) and processor (CPU) resources.

In addition to CPU and RAM metrics, autoscaling can be configured based on application-specific metrics. For example, scaling can be based on the number of requests per second, queue length, or any other relevant metric, which has become a key concept of the KEDA solution [7].

Kubernetes can also scale based on external metrics, allowing you to incorporate data from external monitoring systems into the autoscaling decisions. Several publications including [8] are proposing to follow the rule: more preferable is the usage of the absolute values as minimizing the risk of the overregulation and providing more precise regulation.

Chosen metrics should be retrieved by collector or monitoring system, for example by Prometheus scraping approach like in [5] or push initiated from the application side commonly used in integrations with cloud monitoring like AWS CloudWatch [9]. All these systems are concentrating only on the retrieval of the metric values. The art of data transformation and processing one to get any kind of service-related or customer facing metrics, to get the value from values, is still a work of a DevOps engineering team.

In [5] are advertised results for the series of the experiments of loading some HTTP-serving pods with short-term HTTP requests modelling different conditions of load and HPA default CPU load metric as autoscaling criteria. Most of the results have depicted single wave spike of CPU load reacted to upscale without overregulation. That could be considered as absence of the overregulation effect as a drawback for proportional regulation. But different behaviour is advertised in the experiment described in [5]: the first and the second spikes over the average rate of HTTP requests in the distribution of the average of HTTP requests per second. Looking like this shape of load in the experiment was caused by the requests queueing to be called before previous resolved. This could be explained with the testing stand was stuck to send queries and the limitation is mirroring the response from the front side ingress load balancer experiencing semi-wave effect under overload.

Another proved cases of overregulation identified by this research authors that even have caused disruption described below. A simplified architecture of one with the microservice application for processing requests to the WSSL in the Fig. 1.

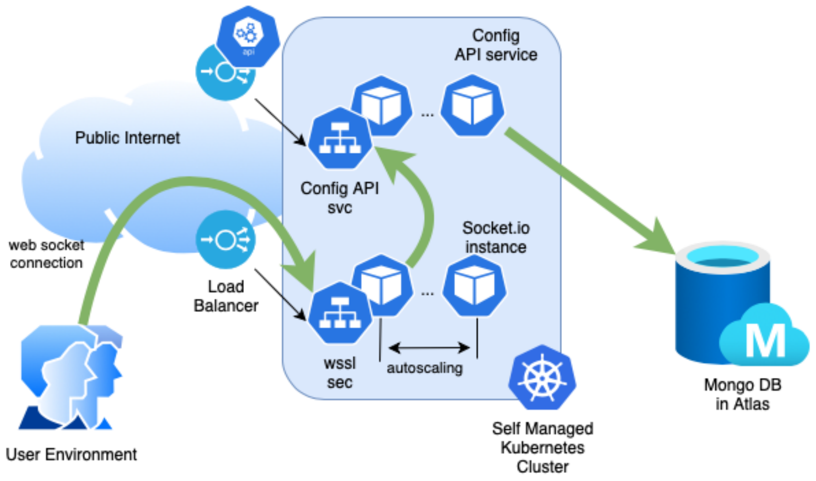


Fig. 1. Microservice Application handling WSSL (WebSocket) sessions for user chat

During the application step-like load growth of the served user requests the data storage system Mongo DB sessions has reached the inoperational state and triggered a switchover. This operation of changing the leading master node in the Mongo DB cluster is depicted in the first from the left vertical line in the Fig. 2. Following switchover hasn`t made the system to reach the stable state. This is advertised in the Fig. 2 with the next following two waves of growing queue of queries (top horizontally-oriented green line) and caused the outage of the next master and the second switchover (vertical red line).

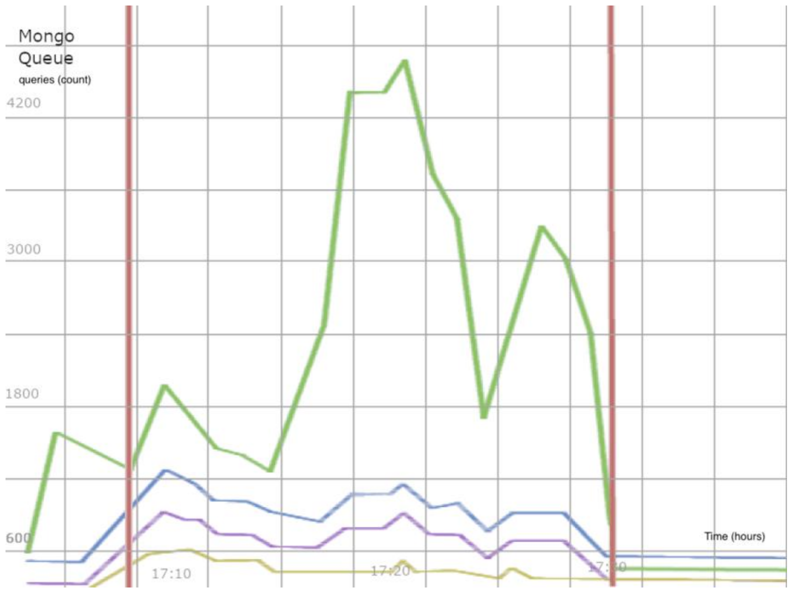
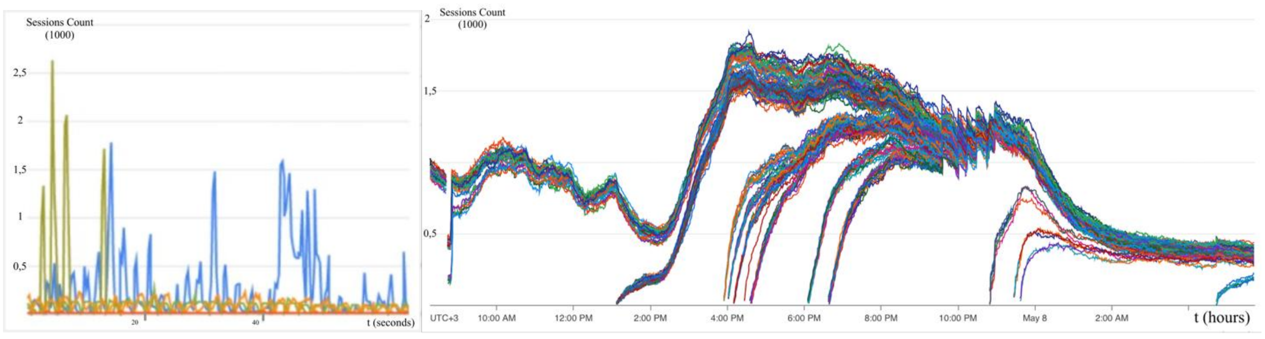


Fig. 2. Queued queries in Mongo DB per time

The root cause analysis has shown excessive upscale of the Config API pods. They were connecting directly to the MongoDB (see Fig. 1) and automatically scaled based on the CPU load. This was caused by the excessive calling of them by the Socket.io, also scaled – but on per-connections count scaling metric. That is expected behavior, but the delayed demand on the Socket.io side has caused the second load wave, that hasn't been handled by the MongoDB properly causing the second handover (the second from the left vertical line of handover in the Fig. 2).

The issue was considered in the application software design gap of missed outgoing shaper on Config API side. And was temporarely resolved with setting the pods count upscale limit. But from the theory of regulation this case is looking pretty common to autoscaling overregulation [10]. The effect that possibly could be predicted, but wasn't widely known in Kubernetes autoscaling before.

As was reviewed before, the negative feedback regulations is a common approach for the most Kubernetes autoscaler contolers like default one [4] and KEDA [7], but the flapping effect is not so common in the authors practice. Suggesting that is was caused by the short-time of live for typical HTTP and HTTP for REST API calls usually set to seconds apart to days set for WebSocket HTTP sessions [11]. This is why the problem as advertised in the Fig. 3 is typical for more live long sessions. For understanding the difference, take a look in the Fig. 3,*a* advertising REST API HTTP Config API sessions. Time-connection diagrams to REST API service from the mentioned above application in the Fig. 3,*a* (different colour-different instance) and WebSocket HTTP sessions example in the Fig. 3,*b* in the same time range.



a *b*

Fig. 3. The shape of sessions for the described in the Fig. 2 environment reproducing the overregulation outage case: *a* – HTTP REST session; *b* – WebSocket session

Each line in the Fig. 3 is the sum of simuleteneous sessions on some pod of the mentioned application component. Sharp spike on the left REST diagrams – it's a short life HTTP REST sessions. Long living curves on the left advertising long-term WSSL sessions continuing to utilise pod resources for hours.

Both Kubernetes native [5] and CNCF graduated KEDA [7] are limited to solutions operating like single criteria controllers: implementing proportionally scaling targeted microservice based on some referenced metric converted to or retrieved as integer value. Kubernetes native HPA controller [5] is advertising the mechanism of limiting the speed of upscale and downscale. The practice of applying one is kept on the Kuberenetes site realibility engineers or DevOps engineers. And it`s the only available for now mechanism described as metod of overregulation omitting [5]. Also absence of the point of view to the autoscalers as proportional regulator almost missed in Kubernetes native HPA controller managing recomendations [5] and in the official documentation. The second mechanism of proportional regulator from autoscaling tuning: time parameters tuning.

This raises the problem of the absence of the described methods for optimizing HPA and omitting it`s overregulation – also the typical issue of the proportional regulator [10]. And makes a statement for the need of the research of the method of HPA tuning as a proportional regulator, to prove the overregulation effect is reproducable also for short-term sessions like HTTP REST and not only WebSocket one as mentioned for the case described in the Fig. 1 and Fig. 2.

3. The aim and objectives of the study

The purpose of the study is to provide the method of setting HPA in Kuberentes to omit overregulation effect and to define the proportional regulator nature of the HPA and mathematical approach of identifying key parameters for optimising HPA.

Infrastructure should be configured to achieve maximum efficiency in the use of computing resources when utilised for commercial microservice applications. From the user's point of view, this means that the information will be received with the stable quality, regardless of the significant dynamics of user requests, and it will be payed the minimum cost for the computing resources used. To achieve this, the IT infrastructure management system should automatically increase the number of resources when the quality of the service deteriorates. HPA is increasing the number of pods but keeping the proportional approach to metrics defined for autoscale. This could cause overregulation in the extremums of the load or even to disruptions in the regular functioning of applications. Microservices WSSL overregulation is already identified for the case described in the Fig.1 and Fig. 2. To be able to proceed with meaningful experiments also required to reproduce the conditions of overregulation for the short-term REST HTTP sessions.

So the object of the study is a horizontal autoscaling of the pods serving REST HTTP service with overregulation and their behaviour under applied autoscaling optimization method.

Describing the math base of the effect to understand is it possible to reproduce one for short-term REST HTTP sessions. The rule for calculating the desired replicas count of deployment by an explicit horizontal autoscaler in Kubernetes [5] can be formulated as follows:

$$N_d = \left\lceil N_c \cdot \frac{M_c}{M_n} \right\rceil, \tag{1}$$

where $\lceil x \rceil$ is a ceiling function, the smallest integer that is not less than x, N_c is the number of replicas current, N_d is a number of replicas desired, M_c is metric previous value, M_n is metric new value.

In case the autoscale metric is resource-based (such as CPU, RAM, etc.), it becomes possible to consider management based on the principle of resource usage functions. Then the following equation becomes:

$$N_c(t) = \frac{W_c(t)}{R_c(t)}, \tag{2}$$

where C is a one-dimensional resource quantity (usually CPU or RAM), $W_c(t)$ is a function of resource usage C (workload), $R_c(t)$ is a function of request for a resource (request) for one instance, $N_c(t)$ is a function of the total number of replicas (requested number of replicas),

Taking into account (2) equation (1) the definition of the current number of replicas changes to:

$$N_d(t) = \left\lceil \frac{W_c(t)}{R_c(t)} \cdot \frac{M_n(t)}{M_p} \right\rceil, \tag{3}$$

considering that metric collection is not an instantaneous operation, and by default in Kubernetes it is collected by the autoscale controller with a certain delay t_r (time delay for retrieval), which is an option for launching kube-controller-manager [3], equation (3) for metrics takes the form:

$$N_d(t) = \left\lceil \frac{W_c(t)}{R_c(t)} \cdot \frac{M_n(t-t_r)}{M_p} \right\rceil, \tag{4}$$

or

$$N_d(t) = \left\lceil \frac{W_c(t)}{R_c(t)} \cdot \frac{M_n - \Delta M(t_r)}{M_p} \right\rceil, \tag{5}$$

where $\Delta M(t_r)$ represents the expected change in the metric over the delay. The more accurate the prediction $\Delta M(t_r)$, the more accurate the scaling decision will be.

If to take into account that launching new replicas may not be an instantaneous operation and require time for initiation and readiness, we introduce variables that determine the delay time for adding a new instance t_{up} (time delay upscale is time spent for pod initiating) and the delay time for collapsing the replica during downscaling t_{down} (time delay downscale is time spent for demolishing). And since control is a system of equations for both deployment under load and collapsing (reducing the number of replicas) when the load drops, we obtain the formula:

$$N_d(t) = \begin{cases} \left\lceil \frac{W_c(t)}{R_c(t)} \cdot \frac{M_n - \Delta M(t_r)}{M_p} - \Delta N_{t_{up}} \right\rceil, \text{ for upscaling } M_n - \Delta M(t_r) > M_p, \\ \left\lceil \frac{W_c(t)}{R_c(t)} \cdot \frac{M_n - \Delta M(t_r)}{M_p} - \Delta N_{t_{down}} \right\rceil, \text{ for downscaling } M_n - \Delta M(t_r) < M_p. \end{cases} \tag{6}$$

In case of upscale $\Delta N_{t_{up}}$ or downscale $\Delta N_{t_{down}}$ does not depend or only slightly depend on external factors (availability of connections to the database, waiting in line for service registration, etc.) ΔN_t can be replaced by a shift in the resource function $W_c(t)$ and simplify (6) to:

$$N_d(t) = \begin{cases} \left\lceil \frac{W_c(t - t_{up})}{R_c(t)} \cdot \frac{M_n - \Delta M(t_r)}{M_p} \right\rceil, \text{ for upscaling } M_n - \Delta M(t_r) > M_p, \\ \left\lceil \frac{W_c(t - t_{down})}{R_c(t)} \cdot \frac{M_n - \Delta M(t_r)}{M_p} \right\rceil, \text{ for downscaling } M_n - \Delta M(t_r) < M_p. \end{cases} \tag{7}$$

Below is applying the extreme case to the (7) for checking for the correctness of the obtained equation. Fot the case suggesting that the change in metrics is immediate to the load $\Delta M(t_r)$ is just the differential of the metric change from M_n to M_p with delay t_r:

$$\Delta M(t_r) = M'(t - t_r)dt, \tag{8}$$

That is, for the case of linear load growth:

$$\Delta M(t) = 0. \tag{9}$$

So on the range of t_r the (7) for $Nd(t)$ is simplifying to initial:

$$N_d(t) = \left\lceil \frac{Wc(t - t_{\text{up/downscale}})}{R_c(t)} \cdot \frac{M_n}{M_p} \right\rceil. \tag{10}$$

where $t_{\text{up/downscale}}$ is the time lag for upscale-downscale decision taking. Obtained equation (10) common to the (3) is proving the made transformation hasn't added mistake.

From the formula (10) could be concluded that the time tuning is the key criteria that could be used to proof getting of the flapping effect and resolving one (same as for traditional proportional controller [10]). To prove possibility of stabilizing flapping effect will take upscale version of the formula (7).

4. The study materials and methods

The default Kubernetes autoscaler was choosen as the autoscaler controller for the cases of the reproducing the overregulation affect. For the review and comparison of the methods of resolving of the issue to exclude one preferred direct call to Kubernetes API to set the desired count of the replicas.

The cases of the overregulation on HPA were observed before in the Fig. 2 only for the cases of the usage of the long-term WSSL sessions with timeout counted to days and big enough Google Cloud environment that will be hard to budget for the research. As the preparation stage was tried to identify the test load profile making it possible to use small environment and HTTP REST calls for reproducing the overregulation case. The traffic load profile was reproducing the most common REST GET and POST calls to simulate traditional Web application backend load. Simulation of the internal load made by processing the calls inside stub application in the pods reproducing typical parsing and processing of the REST calls. Stub application developed to be single threaded to make more predictable load behavior as the aim of the experiment to understand and compare actual load with required to handle the ingress requests and as load itself is put out of the scope of the experiment.

Experimental stand used for executing the set of computational experiments and to perform analysis over collected statistics. The results used to prove the presence of the common effects of the overregulation considering Kubernetes HPA setup as a negative feedback regulator (see Fig. 4).

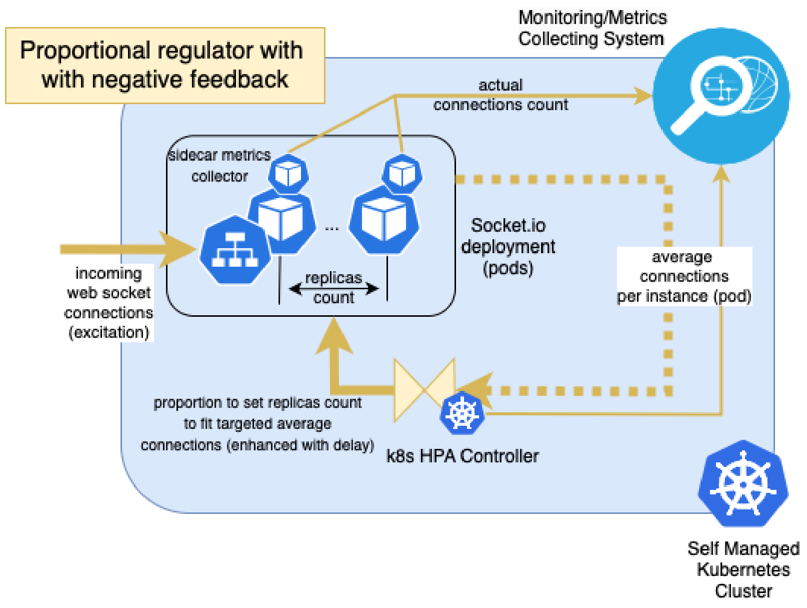


Fig. 4. Kubernetes HPA as a proportional controller with negative fedback

These simulations allow the exploration of how HPA adjusts the number of pod replicas based on observed metrics such as CPU utilization or custom performance indicators. Through iterative experimentation within the simulated environment, could be analyzed the effectiveness of HPA in maintaining desired performance levels while minimizing resource consumption. Additionally, computational tests facilitate the investigation of the impact of different parameters, such as scaling thresholds or cooldown periods, on the regulation behavior of HPA. This approach provides valuable insights into the underlying mechanisms driving HPA's proportional regulation with negative feedback, aiding in the optimization of Kubernetes deployments for enhanced efficiency, resilience, and scalability.

For metrics collecting used hybrid sampling defined by start of the experiment load as an event-driven and tracked as time-driven to track the setup reaction and load and post-load adoptions in HPA.

For collecting pod autoscaling time, key metric CPU and replicas count metrics is used a Prometheus deployed in the same cluster and integrated to Kubernetes via metrics server and per-node exporters. To identify client side metrics are used locally written by load-testing tool logs, synchronized with Prometheus by the timestamps.

Measures are collected as timestamp-value records for the few parameters. Test application actual pods count. CPU load, where 100% are considered as one virtual CPU utilization. HPA controller state per the same timestamp as other values.

According to [10] the traditional way of resolving proportional regulator limitations is substituting one with proportional–integral–derivative (PID) controller. For this default Kubernetes autoscaler was substituted with straight-decision schema of upscaling the application advertised in the Fig. 5.

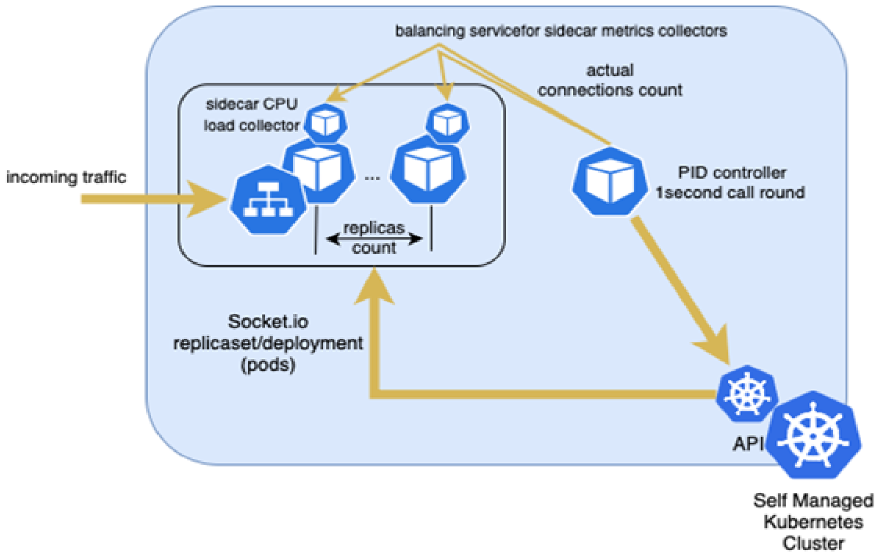

balancing servicefor sidecar metrics collectors
actual connections count
sidecar CPU load collector
incoming traffic
...
replicas count
PID controller 1second call round
Socket.io replicaset/deployment (pods)
API
Self Managed Kubernetes Cluster

Fig. 5. Application Shcema for the direct HPA with a PID controller with negative feedback

The main hypothesis of the research is that tuning of the HPA will be almost the same as tuning of the proportional regulator making it possible to resolve overregulation effect on the HTTP REST calls by applying method of the time delay tuning.

The method of horizontal pod scaling in kubernetes to omit overregulation could be simplified to the next set of steps that should be performed for each application as part of it's autoscaling tuning.

Step 1. Setting up the monitoring system like in the Fig 4 and Fig 5. for collecting same upscale and customer service facing metrics from the application and cluster.

Step 2. Perform load tesing using artificial or real customer data to put the maximum expected load or over the expected one.

Step 3. Based on collected metrics to identify actual system overscaleduring upscale. To identify tdoes it fit the critical path for possibilities to upscale or by the costs limitations.

Step 4. To tune the time parameters and return to the Step 2 to get the most applicable state of the system upscale.

The primary objective was to get more optimized by the count of instantiated pods with predefined profiles with fixed requests for computing resources of CPU under HPA upscaling for the conditions reproducing the cases of the overregulation.

Simulation with fixed value of the load and predictable count of the pods that could handle the load was aimed to examine the system performance under the peak load and compare it to ideal case make it possible to follow on deviation of the actual number of pods from ideal case by the tuning of the parameters supposed as affecting the autoscaling process has proved the concept of the nature of autoscaling and propose the next methods of resolving the overregulation and flapping effects in the Kubernetes.

As the key parameters affecting upscaling with overregulation was proposed and proved the parameter of the time gap between metrics affecting upscaling decision and application replicas scaling decision. To be able to prove the method used for analysis of the problem and it's nature common to the issues of the proportional regulator, it was used another method of the optimization of the upscale process using proportional-integral-derivative controller incorporated into the application to be able to affect own replica count through Kubernetes API.

5. Results of the investigating

Taking into account formula (10) between tuned parameters was choosen t_r (time for metrics retirieval defined with controller option “horizontal pod Autoscaler sync period”) as the easiest one to affect in the lab. With such assumption we are getting:

$$N_d(t) = \left\lceil \frac{W_c(t-t_{up})}{R_c(t)} \cdot \frac{M_n - \Delta M(t_r)}{M_p} \right\rceil, \text{ for upscaling } M_n - \Delta M(t_r) > M_p. \qquad (11)$$

For Kuberenets the change of this parameter requires the cluster restart. But this is much simplifying the experiment itself without need to tune the application and doesn’t require adding artificial delays for the case reproducing in clusters with small actual load (affordable for student experimental stands).

Increasing of the time of the metrics retrieval t_r is equal to changing the reaction time, which also means switching the traditional proportional regulator from balanced to the point where over-regulation may be visible.

In the next experiments was used an application simulating processing of the mix of the REST HTTP GET POST calls (simulating user ingesting queries). The set of quieries is made almost the same. The load is made not equal for the all time of the experiment to achieve the affect of the growing spike of delayed service like in the Fig. 2.

Application is made single threaded with architectural execution limitation in 1 CPU (or virtual CPU utilized in the Kuernetes cluster. All pods were scheduled to be executed on the same node as ingesting traffic was terminated to to exclude different pathes traffic routing lag. The amount of free virtual CPUs (vCPU) was much more than actually used by the application to make it possible.

Application was deployed in two replicas of pods making possible to check balancing on the initial state and but making useless for analysis first load part of the diagrams (fisrt around 10seconds of the experiments). In all cases were used the next criteria for application scaling: an average surge of up to 80% vCPU usage for the all running instances.

As serving HTTP calls containers are made as simple as possible and their scheduling enforced to the particular node it was possible to achieve semi 1s bootstrap of the containers to minimize affect on the rescaling results.

In the Fig. 6 is depicting 2 minutes for the case when horizontal pod autoscaler is set to non-default 20 seconds.

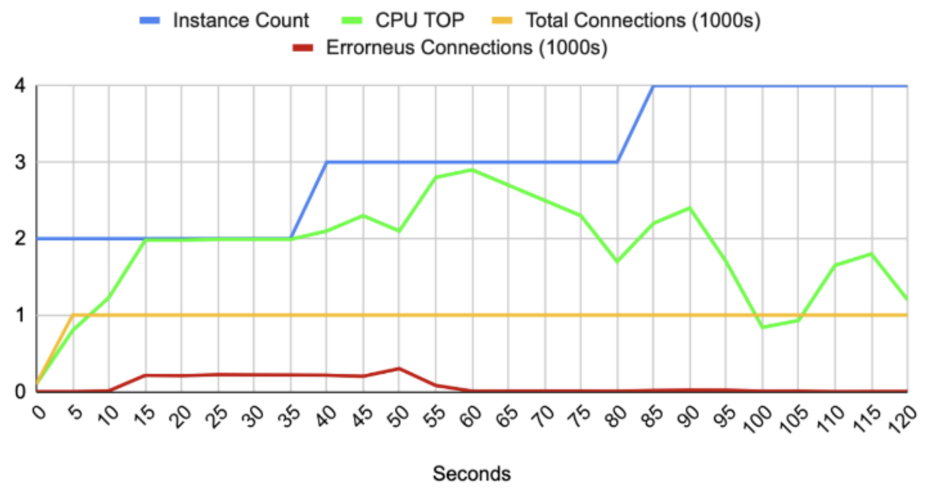


Fig. 6. Pods counts by time of the load experiment for the 15 seconds metric retrival

The blue line of the actual pods count should show the ceil function with round in 80% of the average CPU top per pod level and till the second 60 the graph behavior is semi expected. The CPU load spike on the second 60 is not causing upscale and to set 15 second horizontal-pod-autoscaler-sync-period time (reaction time t_r). Plus time required to apply the changes, in this case as we see upscal in the 85th second is around 5–10 seonds in addition is causing overreaction on the 85+ second. Following absece downscaling caused by the default Kubernetes autoscaler cooldown timer.

For the next case advertised in the Fig. 7 the time of the reaction is downsized to 15 seconds making it possible to reach actual load drop on the 75th second and not to cause overreaction causing overregulation.

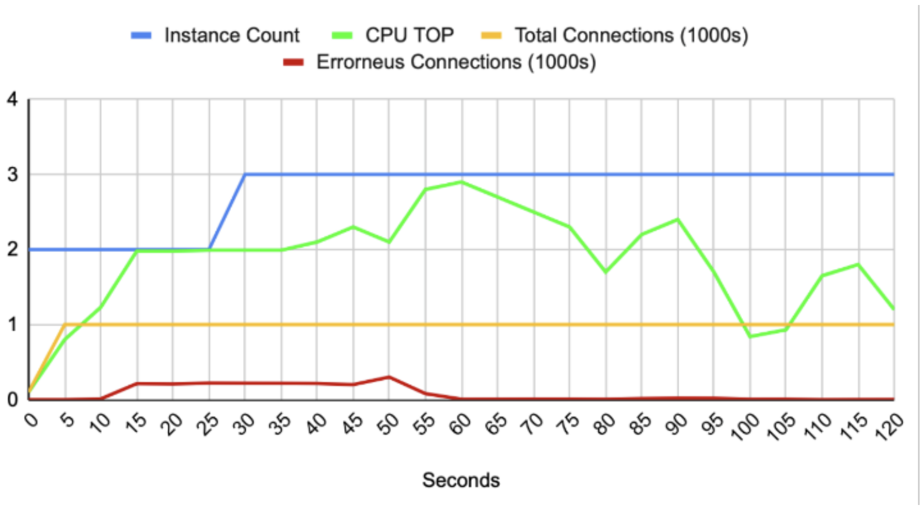


Fig. 7. Pods counts by time of the load experiment for the 15 seconds metric retrival

In proportional control systems such as Kubernetes HPA, time lag plays a critical role in the quality of control [12]. A significant time lag between the detection of a deviation from the setpoint and the initiation of corrective action can compromise the system's ability to maintain stability and desired performance levels. A long time lag can lead to undercontrol where corrective action is delayed, resulting in performance degradation or resource depletion. Conversely, as the time lag becomes shorter, the system becomes more sensitive, but this can also lead to overcontrol.

Application is using the same serving containers, but extended with sidecar containers collecting actual CPU load. The default metric exporting in the Kubernetes also advertsing around 15 seconds lag in metrics collecting. So Kuberenetes API called by the additional pod containing autoscaling logic (same as was set to Kubernetes HPA autoscaler) directly setting desired count of replicas.

Also taken a note for errors growing on the shapes of swing of load from the 80 second to the 85 second in the Fig. 8.

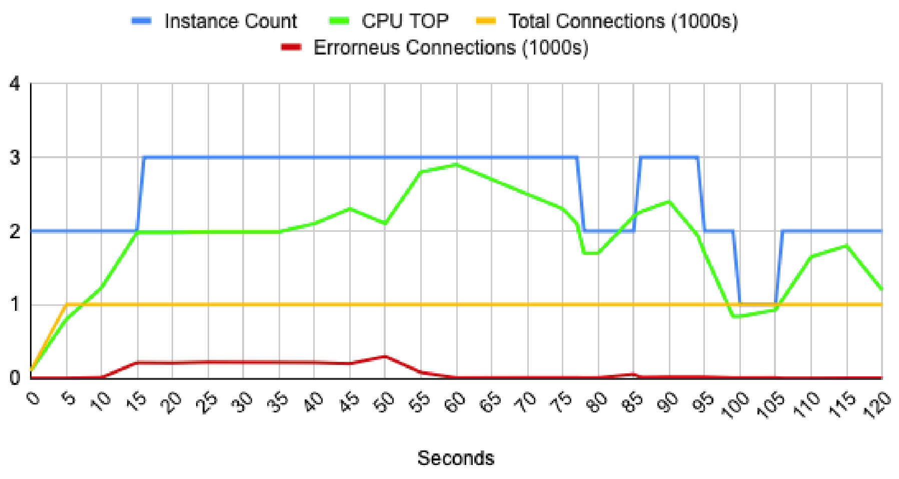


Fig. 8. Pods counts by time of the load experiment for the 1 seconds metric retrival under PID

6. Discussion

In the realm of scientific inquiry into optimizing containerized environments, suggesting Kubernetes Horizontal Autoscaling (HPA) based on collected metrics can be likened to employing a proportional regulator. Just as a proportional regulator dynamically adjusts its output in response to changes in the error signal, HPA dynamically scales the number of pod replicas based on observed metrics such as CPU utilization or custom performance indicators. The principal digram is advertised in the Fig. 6.

This parallels the principles of feedback control systems in engineering, where proportional control mechanisms aim to maintain a desired setpoint by continuously monitoring and adjusting system parameters. In the context of Kubernetes, HPA acts as an automated feedback loop, ensuring that the application's resource allocation matches current demand levels. By correlating collected metrics with scaling decisions, HPA optimizes resource utilization, enhances system responsiveness, and fosters efficient operation of containerized workloads. Thus it's exemplifying the application of control theory principles in modern cloud-native computing environments.

Very close results of the experiments for the PID-optimized regulator in the Fig 6 and time-optimized one in the Fig. 8. They are showing time gap between metrics collection and apply minimizing could be enough for reaching semi-optimal level of autoscaling regulations for some conditions. This also could be an explanation why most common non-default autoscaling solutions like KEDA and default Kubernetes autoscaler are utilizing only simple proportional regulator logic.

Overregulation occurs when the system reacts too swiftly to minor fluctuations, resulting in frequent and unnecessary scaling actions. This effect is already confirmed by [5] and named as trashing. This over-adjustment can lead to increased resource churn, reduced efficiency, and potentially oscillatory behavior, undermining the overall stability and performance of the application. Therefore, balancing the time lag in proportional regulation is crucial to achieving optimal regulation without succumbing to the pitfalls of under or overregulation.

The experiment is covering only the upscale process as experiencing effect of overregulation, but the second side of the optimization during scale is a downscale process advertised only on the last PID-regulator experiment. This process is more complicated in the aspect of providing proper level of service and affect to end-user sessions as requiring drop of one during downscale. So for this collecting of the service level indicators (SLI) and setting the service layer object (SLO) becomes vital to be able to estimate how smooth is the process of the downscale. The aspect of customer sessions and user experience affect with the delay added by the collecting of the metrics and the possibilities on unequal loading based API calls by types rebalancing could be achieved with ingress products like ISTIO [12].

To fulfill the question of optimal scale is needed also to review the downscale process optimization. One is requiring the proper pod terminating (and closing or dropping customer sessions). To cover the case the research should be continued, and the experiment is repeated including the aspect of service level indicators values and their collecting and management affect to the autoscaling process.

Conclusions

Provided research has shown that the default Kubernetes autoscaler could reproduce the flapping effect even for short-term HTTP REST API calls and not only long term WebSocket. Identified, that the HPA logic is highly dependent from the time gap between metrics collecting and scale decision applying pretty same as proportional regulator. Proved the method of HPA time delay tuning by minimization of the metrics retrieval time. The default Kubernetes autoscaler utilizing proportional controller with negative feedback logic could be both non-reproducing flapping and reach the accuracy of incorporated PID controller by applying this change of one. This mean that time tuning could be enough for reaching stable upscale process and explains why nowadays autoscaling controllers like KEDA are still utilizing only simple proportional regulator logic.

References

[1] E. Truyen, D.V. Landuyt, D. Preuveneers, B. Lagaisse W. Joosen "A Comprehensive Feature Comparison Study of Open-Source Container Orchestration Frameworks", Appl. Sci. 2019, 9(5), pp. 35–43; https://doi.org/10.3390/app9050931.

[2] B. Burns, J. Beda, K. Hightower and L. Evenson "Kubernetes: Up and Running", O'Reilly Media inc, 2022 pp. 56–112 ; ISBN 978-1-098-11020-8.

[3] D. C. Marinescu. "Chapter 8 – Cloud Hardware and Software", Cloud Computing (Second Edition), Morgan Kaufmann, pp. 281–319, https://doi.org/10.1016/b978-0-12-812810-7.00011-x, ISBN 978-0-12-812810-7.

[4] D.V. Martins "Scaling of Applications in Containers". Master's thesis, 2023. pp. 13–15.

[5] T. -T., Nguyen, Y. -J. Yeom, T. Kim, D. -H. Park, and S. Kim, "Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration". Sensors, 20(16), 4621. 2020 pp. 13-15; https://doi.org/10.3390/s20164621.

[6] O.I. Rolik, S.F. Telenik, M.V. Yasochka, Upravlinnya korporativnoyu infrastrukturoyu. Kyiv, Naukova Dumka, 2018, 576 p., ISBN 098-966-00-1665-1.

[7] D.K. Alqahtani, A.N. Toosi, "Container Orchestration in Heterogeneous Edge Computing Environments. InResource Management in Distributed Systems", 2024 pp. 151–168. https://doi.org/10.1007/978-981-97-2644-8_8, ISBN: 978-981-97-2643-1.

[8] E. Casalicchio and V. Perciballi, "Auto-Scaling of Containers: The Impact of Relative and Absolute Metrics," *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, Tucson, AZ, USA, 2017, pp. 207–214, https://doi.org/10.1109/FAS-W.2017.149.

[9] G. Quattrocchi, E. Incerto, R. Pinciroli, C. Trubiani and L. Baresi, "Autoscaling Solutions for Cloud Applications Under Dynamic Workloads", in IEEE Transactions on Services Computing, vol. 17, no. 3, pp. 804–820, May-June 2024, https://doi.org/10.1109/TSC.2024.3354062.

[10] K. Johan, Å. Richard, M. Murray. "Feedback Control Systems", 1963, p. 295 ISBN-13: 978-0-691-13576-2.

[11] D. Skvorc, M. Horvat and S. Srbljic, "Performance evaluation of WebSocket protocol for implementation of full-duplex web streams", 2014, 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, Croatia, 2014, pp. 1003–1008, https://doi.org/10.1109/MIPRO.2014.6859715.

[12] S. Singh, C. H. Muntean and S. Gupta, "Boosting Microservice Resilience: An Evaluation of Istio's Impact on Kubernetes Clusters Under Chaos", 2024, 9th International Conference on Fog and Mobile Edge Computing (FMEC), Malmö, Sweden, 2024, pp. 245–252, https://doi.org/10.1109/FMEC62297.2024.10710237.

УДК 004.75

МЕТОД ГОРИЗОНТАЛЬНОГО МАСШТАБУВАННЯ ПОДІВ У КУБЕРНЕТЕС ДЛЯ ЗАПОБІГАННЯ ПЕРЕГУЛЮВАННЯ

Олександр Ролік *
National Technical University of Ukraine
“Igor Sikorsky Kyiv Polytechnic Institute”, Kyiv, Ukraine
http://orcid.org/0000-0001-8829-4645

Володимир Волков
National Technical University of Ukraine
“Igor Sikorsky Kyiv Polytechnic Institute”, Kyiv, Ukraine
https://orcid.org/0009-0008-5325-0368

В статті представлено метод запобігання ефекту надмірного регулювання, що за певних умов виникає при автоматичному горизонтальному масштабуванні мікросервісів в системі оркестрації контейнерних застосунків *Kubernetes*. Ефект виявлено, спочатку, лише на тривалих *HTTP* сесіях *WebSocket*, на яких він призвів і як до надмірного використання обчислювальних ресурсів, що знизило ефективність управління *IT*-інфраструктурою, так і спричинило відмову сервісу. Виявлено, що ефект надмірного регулювання відтворюється не тільки для з’єднань із довгостроковими *HTTP*-сесіями як-то *HTTP WebSocket*, так і для більш короткотривалих *REST HTTP* сесій за умови збільшення затримки в циклі збору метрик, що використовувались для горизонтального автомасштабування. Зроблено припущення, що це пов’язано з підходом реалізації контролерів горизонтального масштабування схожим до принципів роботи пропорційного регулятора у системах з негативним зворотнім зв’язком з теорії автоматики і управління. Запропоновано вирішити проблему одним з методів оптимізації пропорційного регулятору, а саме зменшення часової затримки управління автоматичного масштабування в *Kubernetes*. Застосований метод продемонстрував ефективність, тому у межах тієї ж методології був проведений експеримент з застосування пропорційно-інтегрально-диференціального регулятору для автоматичного горизонтального масштабування подів. Отримані результати показали, чому пропорційно-інтегрально-диференціальний регулятор не поширений серед розглянутих рішень *Kubernetes* для горизонтального автоматичного масштабування, і зроблено припущення про обмеження дослідження зворотного до масштабування процесу згортання кількості реплік подів в *Kubernetes* через необхідність врахування якості обслуговування при зупинці подів і необхідності збору метрик індикаторів засобами управління об’єктами якості сервісу як то *ISTIO*.

Ключові слова: *Kubernetes*, мікросервіси, горизонтальне автомасштабування, пропорційне регулювання, *WebSocket*, хмарні обчислення, *SLI*, *ISTIO*, інформаційні системи, системи управління.