Algorithm for Containers' Persistent Volumes Auto-scaling in Kubernetes

Igor Konev, Igor Nikiforov, Sergey Ustinov
Peter the Great St. Petersburg Polytechnic University
St. Petersburg, Russia
hokletka@gmail.com, igor.nikiforovv@gmail.com, usm50@yandex.ru

***Abstract*—The article describes a proposal of an algorithm for Persistent Volumes auto-scaling in a container orchestration system – Kubernetes. Auto-scaling stands for decreasing/increasing target resources depending on some load. In case of horizontal Volumes scaling, we vary the number of Volumes, while in the vertical case we vary the size of each Volume. Also we introduce mixed scaling that includes both: horizontal and vertical as sub steps to reach a desired state. It is proposed to choose from the methods above based on free storage capacity of Kubernetes cluster and its nodes. Approaches described in the article allow to use persistent storage resources in an efficient way without manual interaction with Kubernetes cluster.**

I. Introduction

Many modern companies use multi-machine associations as infrastructure and computing resources, for example: Amazon, Facebook, Google and others. Currently, containerization is gaining popularity [1], while classical virtualization is used less and less [2]. A Container is a software product wrapped in an environment (for example, an image of Linux distribution) that includes all of its dependencies. This approach is very convenient for transporting and deploying applications. Kubernetes is open source software for automating the deployment, scaling and management of containerized applications [3]. To work with Kubernetes, it is required to set up an environment – Kubernetes cluster. The cluster is a set of computers connected by Kubernetes-related software to make a united computational resource.

To dive into Kubernetes and make further reading understandable, there are some Kubernetes abstractions:

- **Pod** is a kind of logical unit that includes one or more containers [4]. Containers in Pod share a common space: storage volumes and networking stack. Containers are combined into Pod to provide specific functionality. Typically, there is only one main-process container in Pod. Other containers are usually called sidecars and placed near the main container to do ancillary work. For example, one Pod can contain a container-server and a container that collects logs from this server.
- **Persistent Volume (PV)** is a volume abstraction representing a portion of the disk space available to a node. PV can be mounted to a Pod container for persistent storage of information [4].
- **Persistent Volume Claim (PVC)** is a request for dynamic allocation of Persistent Volume of a certain size in a running Kubernetes cluster [4].
- **Stateful Set** is an abstraction over Pod that introduces special resources for easy management of Pods that require Persistent Volumes. In Stateful Set specification users can describe desired state of stateful application: needed containers, volumes and number of replicas. Then Kubernetes uses the provided specification and creates a bunch of Pods and Persistent Volume Claims for these Pods [4].
- **Container Storage Interface (CSI)** is a specification that provides interfaces for Storage management in Kubernetes [5]. Basically, it is responsible for creating, deleting, mounting, extending of Persistent Volumes based on Persistent Volume Claims. Implementation of CSI always relates to a storage provider. It can be local physical drives installed on a node or a storage system that is placed outside the cluster. So there are a lot of CSI implementations made by different storage providers.

Term auto-scaling usually refers to scaling of applications or their number [6]. Kubernetes already has in-built methods for application auto-scaling. But there are also some problems related to scaling of volumes. When users plan storage capacity for their applications it is often difficult to predict the load on the system and how much storage space is required, so space is allocated with a margin. This leads to inefficient use of resources, and the reserved storage space is idle. On the other hand, we have the opposite problem. Nowadays applications continuously generate data for storing, for example in the IoT sphere. This puts a huge load on the underlying databases [7]. When the persistent free space allocated to the application is running out, this can lead to the failure of the entire application, so the cluster administrator is forced to monitor such situations and manually add storage space as needed.

In order to overcome these difficulties, we introduce an algorithm of volumes auto-scaling. Proposed algorithm contributes to the efficient use of storage resources and automatically expands volumes as needed. We have also developed a prototype implementation of volumes auto-scaling that covers expansion cases. The results of the work reflect how much human resources such an approach can save.

The article is structured as follows. The second section contains the overview of existing auto-scaling solutions developed for Kubernetes. Based on the related works and community needs we introduce our solution in the third section. Then there are several paragraphs that describe one part of our algorithm in detail, including basic implementation and evaluation of the obtained results.

II. Related works

At the time of this writing, no work has been found related to autoscaling volumes. However, it is worth considering existing approaches to autoscaling other Kubernetes cluster resources.

A. Horizontal Pod Autoscaler (HPA)

Horizontal Pod Autoscaler offers mechanisms to scale the number of Pods based on various metrics [8]. Users provide desired values for metrics they want to observe in HPA. After that HPA Controller watches for metrics source, compares actual values to the desired ones and performs scaling. To simplify, the desired number of Pod replicas is counted according to the following formula:

$$dr = cr * \frac{cm}{dm}$$

where *dr* is the desired number of Pod replicas, *cr* – current number of replicas, *cm* – current value of metric, *dm* – desired value of metric.

There are several types of metric sources. HPA can use built-in metrics – usage of CPU and RAM, provided by metric-server. Also users can create custom metrics related to Kubernetes objects or refer to external metrics that do not relate to Kubernetes at all.

B. Vertical Pod Autoscaler (VPA)

The other autoscaling controller created by the Kubernetes community is Vertical Pod Autoscaler. This controller adjusts Pod resource limits according to incoming load [9]. It manipulates CPU and memory requests of the watched application. Vertical Pod Autoscaler is easy to configure since its configuration requires only operating mode and application to watch. Default operating mode can set proper resource limits to Pods on startup and also regulate limits on the running Pods. But such changes require restart of all Pod's containers. Thus the main disadvantages of the solution are possible downtime and not considering cluster available resources, so Pods can be stuck in pending state after VPA recommendation. Also, VPA is not compatible with Horizontal Pod Autoscaler, and they are not able to work together.

C. Cluster Autoscaler (CA)

Cluster Autoscaler performs autoscaling of a completely different kind. It works like an automatic horizontal scaling algorithm for cluster nodes. Cloud provider user setups Cluster Autoscaler with node pool – the source of computational resources for the cluster. The further process is automatic: dependent on cluster status and resource requests of running Pods, CA adds or removes nodes from the Kubernetes cluster [10]. The implementation of CA is highly infrastructure driven, so Kubernetes provides only interfaces and implementations of basic algorithms for CA. The rest of the implementation relies on cloud providers.

Common autoscalers, used in Kubernetes clusters, were reviewed. But none of them specializes in working with stateful applications, especially in scaling of applications' storage capacity in dependence of consumption.

III. General algorithm of volumes auto-scaling

The main idea behind the volumes auto-scaling algorithm is to calculate the current storage consumption of an application, compare it to the *threshold* set by the user and perform scaling if necessary. We split scaling into two opposite steps: expansion and shrinkage. The criterion for when to start expanding is simple: if the percentage of storage consumption is greater than a predetermined threshold, then the expansion must be performed. The starting point for shrinkage is slightly more complicated. We have another threshold for users to set called the *minimal threshold*. But it's too rushly to perform scaling down as soon as the volumes consumption is less than the minimum threshold. This can lead to problems, for example, at the start of the application, when it does not yet store any persistent data. Or to unexpected shrinkage during abrupt deletion of data, for example, during dropping database tables. In order to solve mentioned issues, we have introduced one more configurable variable named the *shrink check period*. This variable means how long must elapse to make the following shrinkage procedure possible. Thus, by setting the period value according to the data writing load profile of a particular application, the problem of when to start volumes shrinking can be effectively resolved. The summary of the above is represented on the algorithm (Fig. 1).

Storage consumption check for a particular application should occur on a Kubernetes cluster every N seconds or be triggered by pressure events of persistent memory. We call our proposed solution as Volume Autoscaler in the manner of auto-scaling algorithms, developed for the other resources by the Kubernetes community.

We assume the following limitations in our work. Firstly, we compute the general consumption percent for all volumes of an application. Because we consider that a volume auto-scaling algorithm is required for persistent applications that are able to switch to another volume, if the current one is full, or distribute the load on the volumes evenly. Anyway, there is a possibility to implement Volume Autoscaler in the way when it watches not all application volumes, but their subset. The second restriction is that we assume all application volumes to be the same size. The Volume Autoscaler algorithm is designed to work with all volumes in the same manner. As it's been said our target audience is the developers of persistent applications, such as Cloud Storages [11], they usually set up a batch of equal persistent volumes to their application replicas for storing data. Also such restriction allows to simplify the algorithms for calculating the possibility of scaling.

IV. Volumes expansion algorithm in detail

Let us review the volumes expansion algorithm in detail. We do not dive into volumes shrinkage since it can be done in

a similar way, but in the opposite direction. Also we consider the

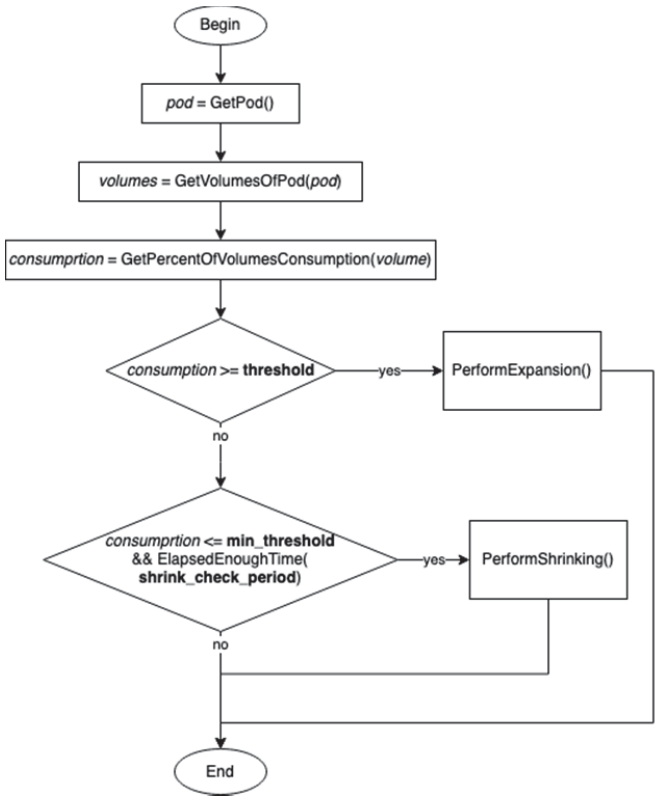

Fig. 1. One iteration of the volume auto-scaling main algorithm.

automatic volumes expansion problem as more important, because lack of free persistent memory can lead to application crashes. In accordance with the problem being solved, expansion should be done when the application does not have enough space to store data. In that situation we have several ways to perform expansion. Basically, it can be done vertically or horizontally.

In terms of volumes, vertical expansion means increasing volumes' sizes without changing their count (Fig. 2). To change volume size in Kubernetes, it's needed to change a size of the corresponding Persistent Volume Claim. But this feature is available only if CSI installed to a cluster implements it, since the feature is not mandatory for matching CSI specification. The only open question is how much to increase the size of the volumes in case of a lack of free space. To do this, we introduce a special variable *expansion coefficient*, the value of which must be set by the user when configuring the Volume Autoscaler. This is due to the fact that the expansion coefficient parameter strongly depends on the specifics of the application, on how quickly it runs out of storage space. So, the current volume size is multiplied by the expansion coefficient, and we get the new desired volume size.

Horizontal expansion is about increasing volumes count without touching their sizes. To create volumes for containers in Kubernetes dynamically, it is needed to create new Persistent Volume Claims and attach them to the running Pod. It requires Pod restart and re-creation, since we change its specification and add new volumes to Pod's containers. In order to count the new number of volumes, we use the same expansion coefficient, we discussed above. From a user perspective the described approach looks like the following (Fig. 3).

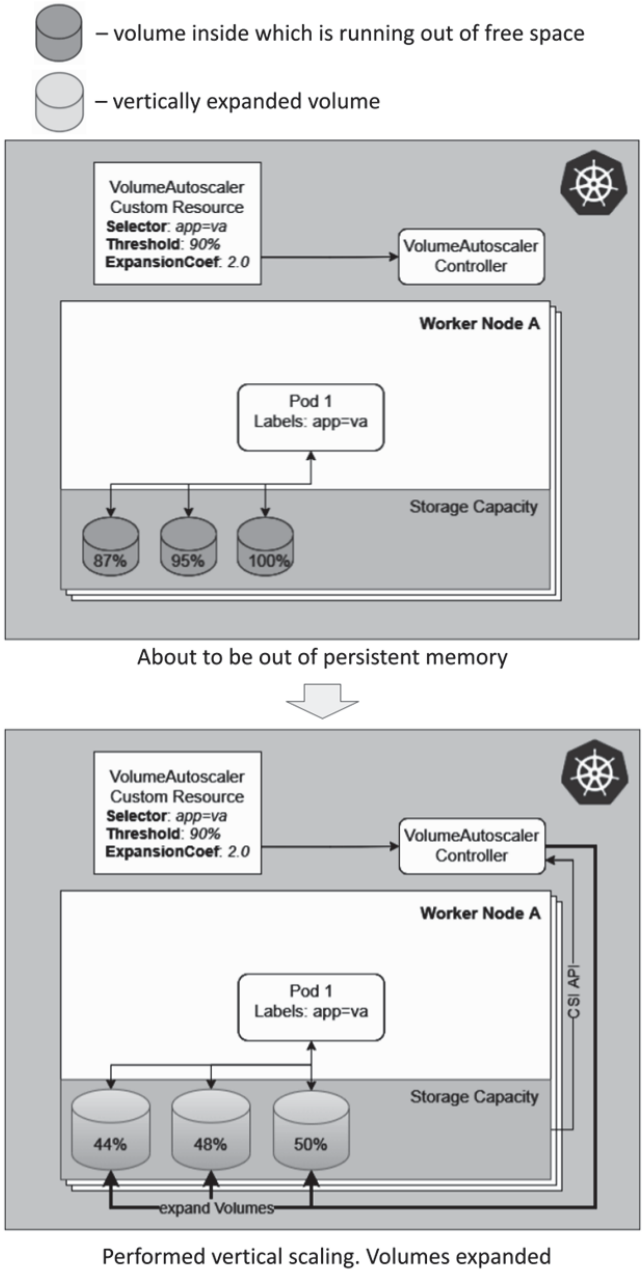

Fig. 2. The result of vertical expansion from a user perspective

We prefer vertical expansion over horizontal because it may reduce downtime of an application. The fact is that horizontal expansion always requires application restart. At the same time, the vertical one may not require the restart at all, and can be performed on a running Pod. It happens when CSI installed to a cluster supports a so-called feature – online expansion [5]. Let us give an example of how it works in the case of using Logical Volume Manager in CSI. LVM is a Linux subsystem designed to create volumes abstractions over physical hard drive/drives. When we change volume size in PVC, CSI detects it and

performs an ex of corresponding Logical Volume placed on the Linux node via LVM utility *lvextend* [12].

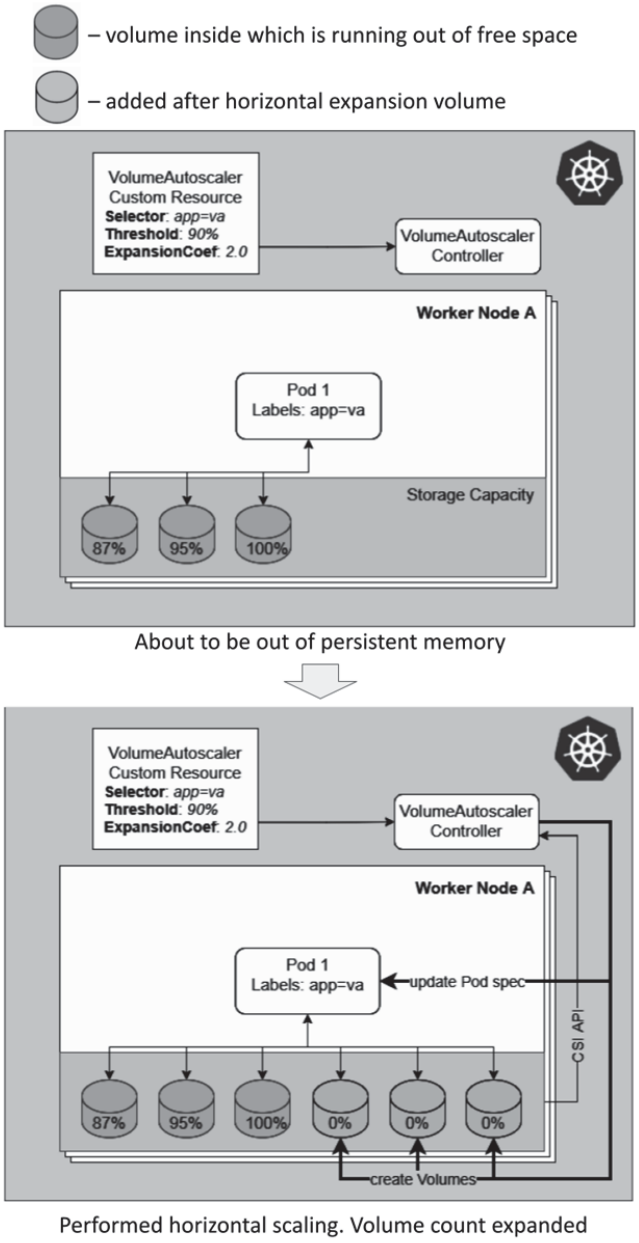

Fig. 3. The result of horizontal expansion from a user perspective

After that CSI triggers extension of the file system installed on the Logical Volume, that is specific for the used file system. Thus the volume size can be increased without Pod re-creation.

Possibility to execute vertical or horizontal volumes expansion depends on available storage capacity in a Kubernetes cluster. For example, in one physical drive layout we are able to perform only vertical expansion because a system has enough space on existing drives but it doesn't have excess drives. We assume that we place volume to a drive one to one to avoid the single point of failure problem [13]. On the other hand, there could be a situation when the system has several free drives but there is no available capacity on drives occupied by volumes. Then only horizontal expansion is possible.

To sum up, it's important to know available persistent capacities on a cluster and layout of physical devices which are underlying the capacities. But by default, Kubernetes doesn't provide an API to check available capacity. Some CSI vendors afford their own API for these purposes [14]. In this paper we do not bind to a particular CSI vendor and introduce a volumes auto-scaling approach without implementation details. So, in the algorithm of choosing the expansion mode we use an abstract function *GetFreeCapacities* that returns a platform agnostic list of free capacities (Fig. 4). After that, we count possible expansion coefficients for every type of expansion in order of priority. If we find an expansion type which satisfies the given expansion coefficient, then we execute it. If a cluster has no capacity for any expansion, then we raise the "Not Enough Capacity" notification to a user.

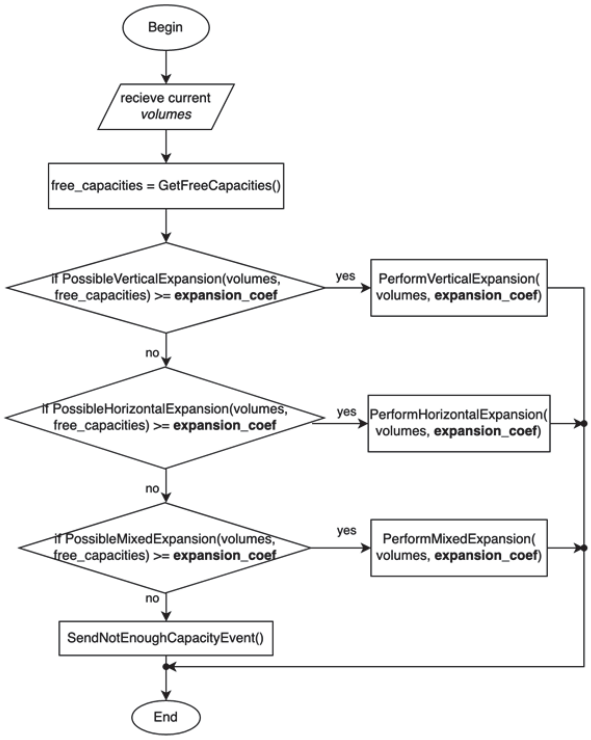

Fig. 4. Algorithm of executing certain expansion type, chosen based on available storage capacity on a cluster

Expansion algorithm (Fig. 4) also includes so-called *mixed expansion*. It is one more expansion type, we introduce in this paper. In some cases, a Kubernetes cluster does not have enough available storage capacity to satisfy the given expansion coefficient. For example, if the system has four storage devices each of 1.5Gb. Observed application has three volumes on them. Each volume is placed onto a separate device and has a size of 1Gb. Expansion coefficient set by a user is 2.0. When free persistent memory runs out in volumes, Volume Autoscaler tries to perform expansion. But neither horizontal nor vertical expansion cannot provide the desired expansion coefficient. In that situation vertical expansion is able to multiply volumes size by 1.5, while horizontal guarantees only coefficient of 1.33. That is where we offer a mixed expansion

that consists of two steps: vertical and horizontal. If we combine both types of expansion in the described example, it's possible to reach the desired expansion coefficient 2.0. First, vertical scaling is carried out, in which all three volumes of the application occupy 1.5 Gb each. After that Volume Autoscaler adds one more volume of 1.5 Gb size to the last free storage device. Thus, Volume Autoscaler increased volume size from 3Gb to 6Gb and got expansion coefficient 2.0 (Fig. 5).

V. StatefulSet based implementation of volumes expansion

There are several ways to implement the Volume Autoscaler algorithm. The first possibility is to create a new workload resource such as Deployment or StatefulSet from scratch and control Pods directly. But in our prototype we use Kubernetes StatefulSet as a secondary underlying resource to perform volumes expansion. With that way our controller shouldn't create Pods and Persistent Volume Claims directly, all needed actions are performed on Pod template and PVC templates of StatefulSet [4]. It's a straightforward solution to demonstrate the working of the proposed algorithm. But such a solution is not production-ready because usually in practice there is a need to scale volumes for a particular Pod. And using StatefulSet with several Pod replicas it's impossible to control volumes of individual Pod. So, we use StatefulSet with one Pod replica for our proof of concept.

The Volume Autoscaler accepts configuration from users. Configuration includes observable application, threshold and expansion coefficient. As far as our controller is Kubernetes-native, we introduce Volume Autoscaler Custom Resource Definition. This is a special way to register a new type of custom resource in Kubernetes [15]. After registration users are able to create Volume Autoscaler custom resources to give instruction to our algorithm on how to work with their applications. The example of Volume Autoscaler custom resource is presented on Listing 1.

Let us slightly overview StatefulSet based implementation of volume expansion. The first thing our controller must do is to identify free space left in a volume. It is possible to use Kubernetes kubelet metrics exposed to Prometheus. The second possible, but security risky option is to connect to the needed Pod's container and use linux utilities to count free storage space. After counting the consumption, we compare it to the given threshold and start expansion if needed. The vertical expansion with StatefulSet basis is presented on Algorithm 1.

Listing 1 VolumeAutoscaler specification

```
apiVersion: volume.scaling.com/v1alpha1
kind: VolumeAutoscaler
metadata:
  name: volume-autoscaler-example
  namespace: vatest
spec:
  expansionCoefficient: 2.0
  targetStatefulSet: autoscaler-test
  threshold: 90
```

Algorithm 1 StatefulSet based vertical expansion

0: Collect all PVCs related to StatefulSet;
1: Update size of PVCs with the desired value;
2: Make updated specification of StatefulSet with new size in PVCs template;
3: It's impossible to update StatefulSet spec because the spec is immutable. So delete StatefulSet in orphan mode to leave its Pod running;
4: Re-create StatefulSet with updated specification.

For the horizontal expansion the algorithm is presented on Algorithm 2.

Algorithm 2 StatefulSet based horizontal expansion

0: Make updated specification of StatefulSet with new number of PVCs and mounts for them in Pod template;
1: It's impossible to update StatefulSet spec because it's immutable. So delete StatefulSet in orphan mode to leave its Pod running;
2: Re-create StatefulSet with updated specification;
3: Delete running Pod to restart it with new number of volumes;
4: StatefulSet controller re-creates Pod with attaching old and new volumes automatically.

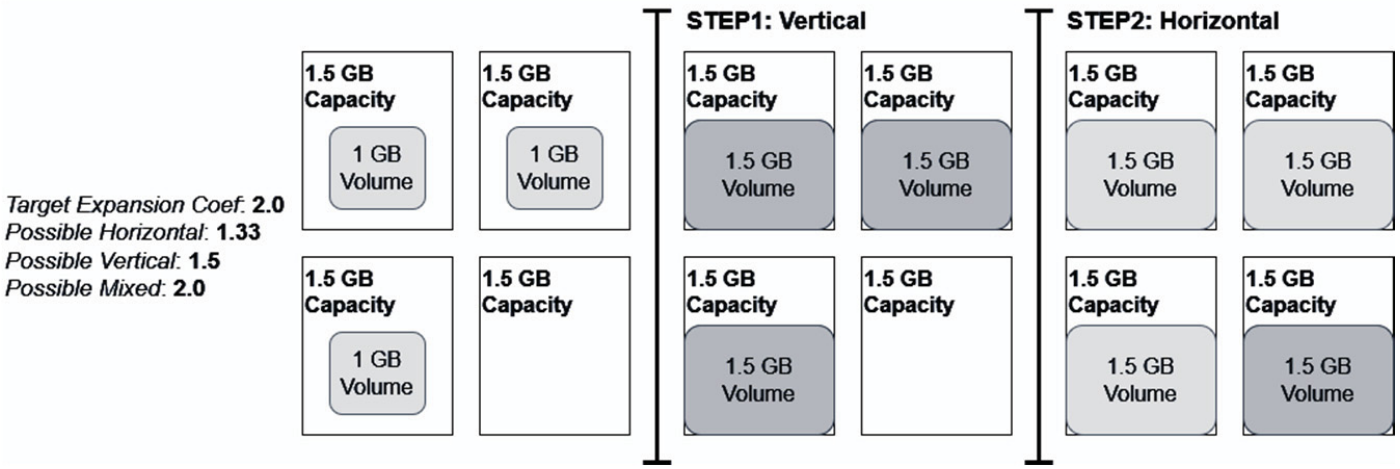

Fig. 5. Mixed expansion in action

Undoubtedly, this prototype is not free from shortcomings. In addition to those already mentioned, there is a danger in orphan deletion of a StatefulSet, since this operation is not atomic and contradicts the Kubernetes concept that the StatefulSet specification is immutable. But such the implementation of the algorithm proposed in the paper is enough to demonstrate its functional capabilities and gather results.

VI. Results of Volume Autoscaler in use

For our tests we prepared StatefulSet with one replica of the busybox application. This application has three Persistent Volume Claims, i.e., three volumes. Each volume has a size of 1Gb. Let us consider the time that is required for manual horizontal and vertical expansion with coefficient 2.0. And also we measure the time spent by Volume Autoscaler for the same operations. We keep track of time from the moment when it's become required to perform expansion, to the moment when all expansion actions are done, and the application is running with scaled up volumes. To measure the elapsed time for manual tests, we just use a timer. Naturally, such measurements have a large error, since the result depends on the specific person performing the task. Therefore, we took the average of several calculations, and our executor was an experienced Kubernetes administrator. In addition, in the current experiment, the error is not very important. The results of experiments are presented in Table I.

TABLE I. Time spent for one expansion iteration

	Time of horizontal expansion (from 3 volumes to 6 volumes), s	**Time of vertical expansion (from 1Gb volumes to 2Gb volumes), s**
Manual	122	80
Volume Autoscaler	52	34

The main note here is that the Volume Autoscaler algorithm is designed to automatically track, when an application runs out of storage space, and to expand volumes at that point. So, the time spent by Volume Autoscaler to the expansion itself does not matter so much. On the other hand, for now in Kubernetes the administrator needs to monitor that situation manually or create an alert system to receive notifications when persistent memory is running out. Assume an abstract situation when the application requires more persistent memory each hour. In this case, the administrator will spend 122 seconds per hour for horizontal expansion and 80 seconds for vertical one. At the same time, let the administrator spend 1 hour per month on maintenance, updating and configuring Volume Autoscaler. Let us extrapolate this situation to the annual time period and build a graph of time that's needed to administrator to operate different solutions (Fig. 6).

According to the graph, using the Volume Autoscaler can save a huge amount of time and human resources. And this is despite the fact that the experiment does not take into account that with each subsequent manual expansion, the administrator must spend more time, because he has to deal with more and more volumes.

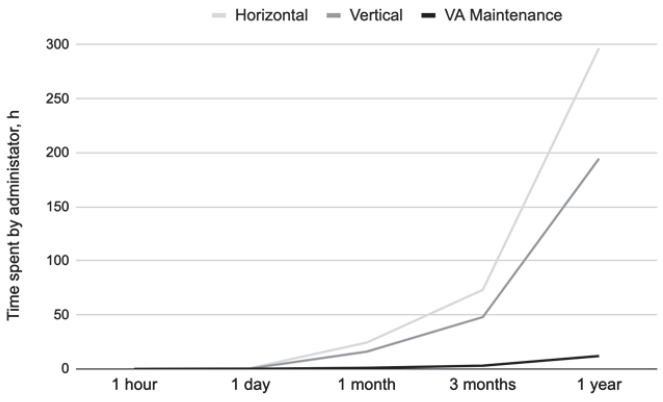

Fig. 6. Time taken by the administrator to maintain the expansion of volumes for various time intervals

VII. Conclusion

In this article we have introduced a new algorithm of automatic volume scaling in Kubernetes. We made an overview of the proposed algorithm and also analyzed in detail the part related to the expansion of volumes. To demonstrate the Volume Autoscaler potential we have implemented a lightweighted StatefulSet based version of expansion scenario. Indeed, in the experimental part, we were able to demonstrate how much resources the proposed automation can save.

Persistent applications can either reduce or increase the load on volumes. To use storage resources efficiently, cluster administrators need to constantly respond to this. Volume Autoscaler just helps to cope with this problem, while minimizing labor costs. With the right algorithm setup, the application will not crash when there is not enough permanent memory, and physical devices will not be occupied by the application aimlessly.

Our future plans include the development of the entire algorithm, including volume shrinking. In addition, the implementation based on StatefulSet has a number of described disadvantages, so it is necessary to work out approaches that solve these problems. Probably, Volume Autoscaler should work with Pods directly, bypassing standard Kubernetes controllers.

References

[1] H. V. Netto, A. F. Luiz, M. Correia, L. de Oliveira Rech and C. P. Oliveira, "Koordinator: A Service Approach for Replicating Docker Containers in Kubernetes," *2018 IEEE Symposium on Computers and Communications (ISCC)*, Natal, 2018, pp. 58-63.

[2] S. Shirinbab, L. Lundberg, E. Casalicchio, "Performance evaluation of containers and virtual machines when running Cassandra workload concurrently," *Concurrency and Computation: Practice and Experience,* 32, 2020.

[3] Burns, Brendan, B. Grant, D. Oppenheimer, E. Brewer, J. Wilkes, "Borg, Omega, and Kubernetes," *Communications of the ACM 59*, no. 5, 2016, pp. 50-57.

[4] Kubernetes official documentation, Web: https://kubernetes.io/.

[5] Container Storage Interface official documentation, Web: https://kubernetes-csi.github.io/.

[6] S. Taherizadeh, M. Grobelnik, "Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications," *Advances in Engineering Software*, Volume 140, 2020, 102734.

[7] P. Kochovski, R. Sakellariou, M. Bajec, P. Drobintsev and V. Stankovski, "An Architecture and Stochastic Method for Database Container Placement in the Edge-Fog-Cloud Continuum," *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 396-405.

[8] T.T. Nguyen, Y.J. Yeom, T. Kim, D.H. Park, S. Kim, "Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration," *Sensors (Basel)*. 2020, 20(16), 4621.

[9] G. Rattihalli, M. Govindaraju, H. Lu and D. Tiwari, "Exploring Potential for Non-Disruptive Vertical Auto Scaling and Resource Estimation in Kubernetes," *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, 2019, pp. 33-40.

[10] M. Wang, D. Zhang and B. Wu, "A Cluster Autoscaler Based on Multiple Node Types in Kubernetes," *2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, 2020, pp. 575-579.

[11] J. D. K. Waguia and A. Menshchikov, "Threats and Security Issues in Cloud Storage and Content Delivery Networks: Analysis," *2021 28th Conference of Open Innovations Association (FRUCT)*, 2021, pp. 194-199.

[12] System Manager's Manual of lvextend, Web: https://man7.org/linux/man-pages/man8/lvextend.8.html.

[13] J. Yang and J. Harting, "Improving Storage Subsystem Reliability - A Case Study," *2007 Annual Reliability and Maintainability Symposium*, 2007, pp. 434-439.

[14] Storage capacity tracking proposal of Kubernetes community, Web: https://github.com/kubernetes/enhancements/tree/master/keps/sig-storage/1472-storage-capacity-tracking.

[15] J. Dobies, J. Wood, *Kubernetes Operators*. O'Reilly Media, Inc., 2020.