HAL
archives-ouvertes.fr

Instability in Geo-Distributed Kubernetes Federation: Causes and Mitigation

Mulugeta Tamiru, Guillaume Pierre, Johan Tordsson, Erik Elmroth

▶ To cite this version:

Mulugeta Tamiru, Guillaume Pierre, Johan Tordsson, Erik Elmroth. Instability in Geo-Distributed Kubernetes Federation: Causes and Mitigation. MASCOTS 2020 - 27th IEEE Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems, Nov 2020, Nice, France. hal-02934475

HAL Id: hal-02934475
https://hal.inria.fr/hal-02934475

Submitted on 9 Sep 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Instability in Geo-Distributed Kubernetes Federation: Causes and Mitigation

Mulugeta Ayalew Tamiru
Elastisys AB
Univ Rennes, Inria, CNRS, IRISA
mulugeta.tamiru@elastisys.com

Guillaume Pierre
Univ Rennes, Inria, CNRS, IRISA
guillaume.pierre@irisa.fr

Johan Tordsson, Erik Elmroth
Elastisys AB
{tordsson,elmroth}@elastisys.com

***Abstract*—As resources in geo-distributed environments are typically located in remote sites characterized by high latency and intermittent network connectivity, delays and transient network failures are common between the management layer and the remote resources. In this paper, we show that delays and transient network failures coupled with static configuration, including the default configuration parameter values, can lead to instability of application deployments in Kubernetes Federation, making applications unavailable for long periods of time. Leveraging on the benefits of configuration tuning, we propose a feedback controller to dynamically adjust the concerned configuration parameter to improve the stability of application deployments without slowing down the detection of hard failures. We show the effectiveness of our approach in a geo-distributed setup across five sites of Grid'5000, bringing system stability from 83–92% with no controller to 99.5–100% using the controller.**

***Index Terms*—Self-configuration, self-adaptation, Kubernetes Federation, Fog Computing, automatic configuration tuning.**

I. Introduction

Fog computing extends cloud computing by harnessing geographically-distributed computing resources for moving computation closer to where data are generated (e.g., IoT devices). One of the main challenges in fog computing is the autonomous management of tens of thousands of remote nodes and clusters found in diverse locations. Several approaches based on modified container orchestration frameworks such as Kubernetes have been proposed [1]–[3]. More recently, Kubernetes introduced the notion of Federation (KubeFed) which provides abstractions to manage multiple geo-distributed Kubernetes clusters from a single control plane.

Since Kubernetes was designed for managing local clusters in public and private cloud settings, it assumes reliable network connectivity between the nodes with low latency, high bandwidth, and low packet loss. KubeFed makes a similar assumption: while it is designed to manage Kubernetes clusters located in different regions of the same cloud provider or multiple cloud providers, KubeFed assumes high reliability of the network connectivity between the control plane and the managed clusters. However, such assumptions are not met in many fog computing environments [4]. As a result, static configurations, including the default values of the configuration parameters for both Kubernetes and KubeFed are not necessarily well-suited to the case of geo-distributed fog computing infrastructures.

It is well known that configuration parameters may have a strong influence on the performance and availability of systems [5]. However, finding the optimal configuration settings that result in the best performance of the system is not easy because of the large parameter space and the complex interaction of multiple parameters. This is the case of Kubernetes and KubeFed which are composed of several embedded control loops [6] with numerous configuration parameters.

In this paper, we demonstrate that federated applications deployed on a geo-distributed KubeFed infrastructure with static configuration, including the default settings, may suffer from important instability where containers get repeatedly created and deleted before being able to provide a useful service. To our best knowledge, we are the first to report this undesirable behavior of KubeFed.

Our contribution is two-fold. First, we demonstrate the existence of the instability problem in a realistic geo-distributed fog computing infrastructure based on KubeFed and identify one configuration parameter (Cluster Health Check Timeout), whose value influences stability the most. We show that, for obtaining the best system behavior, the value of this parameter should be adjusted according to the characteristics of the execution environment. Second, we propose, implement, and experimentally evaluate a feedback controller that improves the stability of the system by dynamically adjusting this configuration parameter at runtime. We show that this controller is very effective for improving the system's stability across a wide range of inter-cluster network latencies and packet loss rates, without losing the ability to detect actual cluster failures. In our evaluations, the system stability improves from 83–92% with no controller to 99.5–100% using the controller. By enabling self-configuration and self-adaptation, our solution helps make KubeFed more autonomous.

This work is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 765452. The information and views set out in this publication are those of the author(s) and do not necessarily reflect the official opinion of the European Union. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

II. Background

Kubernetes is a container orchestration platform which automates the deployment, scaling, and management of containerized applications in large-scale computing infrastructures such as a cluster and a datacenter [6]. To extend it to multi-site deployments, Kubernetes Federation (KubeFed) supports resource management and application deployment on multiple Kubernetes clusters from a single control plane, thus making it suitable for managing geo-distributed resources [7].

KubeFed's implementation builds upon the concept of *custom resource definitions (CRDs)* from Kubernetes. In KubeFed terminology, a single *host cluster* runs the *federation control plane* which controls any number of *member clusters* where applications may be deployed. The host cluster is also the central point where the federation's configuration parameters are defined. The *KubeFed controller manager* is itself a Kubernetes deployment resource deployed on the host cluster. It runs several controllers to manage the member clusters, scheduling, deployments, services, and other resources.

KubeFed introduces three concepts for each resource:

- *Template* defines the specification of a resource common across all member clusters;
- *Placement* specifies which member cluster(s) will get the resource;
- *Override* defines per-cluster variations of the template.

Using these concepts, users can define their deployments and services and decide how many application containers of a deployment should appear in which cluster. Figure 1 shows the KubeFed architecture with a host cluster and three member clusters where an "nginx" *federated service* is configured to be deployed on only two of the three member clusters.

KubeFed also offers *ReplicaSchedulingPreference (RSP)* which is an automated mechanism to distribute and rebalance federated deployments across the member clusters. This is useful when scaling an application across several clusters. As shown in Figure 2, users only need to specify the target resource to be controlled by RSP and the total number of replicas to be distributed in the federation. By default, RSP distributes the replicas evenly across all member clusters if they have sufficient resources. RSP does this in multiple iterations proportional to the number of total replicas. After calculating how many replicas should go to each member cluster, RSP modifies the Federated Deployment object to update the number of replicas on each cluster, which in turn pushes or reconciles the changes to the member clusters via the *sync controller*. The sync controller is responsible for propagating changes from the Federation Control Plane on the host cluster to the member clusters and maintaining the desired state of resources across member clusters.

There are several configuration parameters that control the behavior of the Sync Controller. The most important for our work is the *Cluster Health Check Timeout (CHCT)*, which has a default value of 3 seconds. This parameter determines the duration after which sync requests and cluster health check time out. In a geo-distributed deployment where large

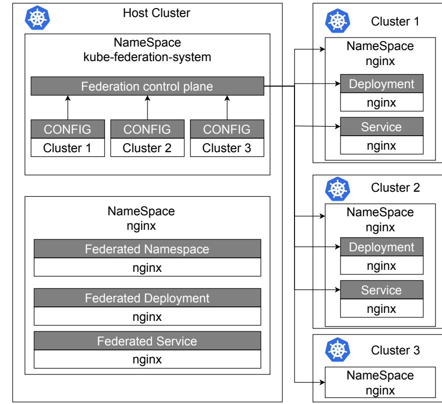


Fig. 1. A simplified view of KubeFed architecture with a host cluster and three member clusters and propagation of federated resources.

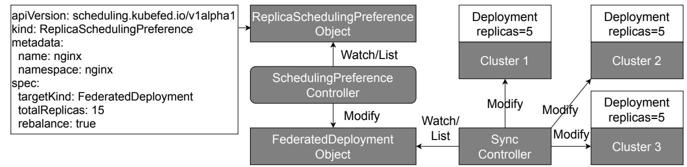


Fig. 2. Automatic distribution of deployment replicas using ReplicaSchedulingPreference on a Federation with a host cluster and three member clusters. RSP evenly distributes a total of 15 replicas across the three member clusters, 5 replicas per cluster.

network latencies and packet losses are commonplace, it is important to choose the right value for this parameter to ensure the correct functioning of the Sync Controller and RSP. A low *CHCT* value ensures fast detection of cluster failures, allowing RSP to rebalance the deployment away from failed clusters onto healthy clusters. However, fast detection increases the probability of false positives due to delays and transient network failures. Therefore, it may be necessary to increase the value of CHCT to minimize false positives. However, increasing the CHCT value may lead to delayed cluster failure detection. It is, therefore, important to consider the trade-off between fast and slow detection of cluster failures when choosing the CHCT parameter value. Manually selecting an optimal value for CHCT is challenging as the choice may depend on the computing and networking environment as well as application workload dynamics.

III. Related Work

Modern distributed systems are increasingly complex and difficult to manage manually. As a result, autonomic computing approaches have been proposed to simplify the difficult task of managing these systems by way of self-configuration, self-adaptation, self-tuning, and self-healing [8], [9].

One aspect of the complexity of modern distributed systems is the large number of configuration parameters with complex interactions that affect the performance and availability of systems. Even though good configuration settings can improve the performance of systems, finding these settings among hundreds of parameters is often far from being trivial [5].

In the past decades, several works have proposed to leverage the concepts from autonomic computing. Many of the works focus on optimizing the performance of systems by finding the best combination of configuration settings from all the possible combinations [5], [10], [11]. Some other works focus on specific type of distributed systems or frameworks such as Enterprise Java (J2EE) [12], big data management systems such as Apache Hadoop [13] and Apache Spark [14], database management systems [15], distributed message systems such as Apache Kafka [16], web servers such as Apache web server [17]–[19], Docker [20], or Kubernetes [21].

Selecting appropriate values for timeout parameters, which is the specific topic of this paper, has proven particularly challenging in a number of related works. Typically, large values result in slow failure detection whereas small values reduce the reliability of the failure detector. To address this challenge, some works have proposed delay predictors which determine at runtime values for detection timeout for fast detection while not reducing the reliability of detection [22]–[25]. In [26], the authors propose an autonomic failure detector based on feedback control theory that re-configures its timeout and monitoring period parameters at runtime in response to changes in the computing environment or application according to user-defined QoS requirements. Similarly, we show that small values for the CHCT timeout parameter of KubeFed lead to instability whereas large values lead to slow failure detection.

The focus of most of these works is to find the right trade-off between accuracy and responsiveness of failure detectors, whereas our main focus is to improve the stability of application deployments without impairing the responsiveness of failure detection. Moreover, unlike these works, our work addresses a problem in a geo-distributed fog computing environment. To the best of our knowledge, the problem of failure detection and dynamic adjustment of configuration parameters have not been widely studied in the context of geo-distributed computing environments such as fog computing.

IV. Problem analysis

When a replicated application is deployed in a Kubernetes federation, in certain settings, the application incurs significant instability where containers are repeatedly created and deleted, which in turn causes application unavailability and unacceptably long application response times. In this section we first experimentally demonstrate the existence of this undesirable behavior, and then analyze its causes.

A. Experimental setup

To highlight the unstable deployment problem, we set up an experimental testbed as close as possible to a realistic Fog computing environment, depicted in Figure 3. We deploy six Kubernetes 1.14 clusters in five sites of the Grid'5000 experimental testbed [27]: two in Rennes, and the other four clusters in Nantes, Lille, Grenoble, and Luxembourg. Every cluster has one master node and five worker nodes. KubeFed v0.1.0-rc6 is deployed on the first cluster as the host cluster, and the remaining five member clusters are then joined to the federation. Each node in the host cluster has 4 CPU cores and 16 GB of memory allocated to it, whereas each node in the member clusters has 4 CPUs and 4 GB of RAM. This configuration emulates typical fog computing resources which are often composed of single-board machines such as Raspberry Pis [28], [29]. We control the network performance inside each cluster and between the host and member clusters using the "traffic control" (`tc`) tool available in Linux systems. The internal network of each cluster has 1 Gbps bandwidth, whereas the network characteristics between the host cluster and the member clusters are defined in Table I. These values are based on a recent study [4] which highlights the characteristics of today's networking technologies used in edge computing settings.

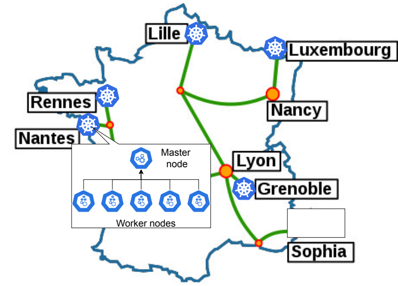


Fig. 3. Experimental setup in Grid'5000 consisting of one host cluster in Rennes, and five member clusters in Rennes, Nantes, Lille, Grenoble (France), and Luxembourg. Distances between sites range from 100 km to 850 km. Each cluster has a master node and five worker nodes. Image adapted from the Grid'5000 website.

TABLE I
Parameters of the network environment between the host cluster and the member clusters.

Network setting	Bandwidth (Mbps)	Packet loss (%)	RTT (msec) from host to member clusters				
			Rennes	Nantes	Lille	Grenoble	Luxembourg
1	15	5	100	102	123	117	127
2	10	10	200	202	223	217	227

The application used for our tests is a simple federated deployment of nginx web server that scales progressively. We scale the total number of replicas from 75, 100, 500, 1500, 2500, to 3500 to be distributed equally among the five member clusters of the federation. The task of automatically balancing the pod replicas across the member clusters is handled by KubeFed's Replica Scheduling Preference (RSP) controller.

We define three scenarios for our experiments:

- **Stationary scenario**: a federation with Network Setting 1, with no variation in the networking environment and no cluster failure;
- **Network variability scenario**: a federation where the networking environment varies between Network Setting 1 and Network Setting 2, with no cluster failure;
- **Cluster failure scenario**: a federation with Network Setting 1, with no change in the networking environment but with a failure and a recovery of one member cluster.

B. The instability problem

As the total number of replicas of the pods of the federated deployment increase, RSP calculates the number of replicas to be distributed to each member cluster. Unless other requirements such as the minimum number of replicas or weights per cluster are specified by the user, RSP opts for an even distribution across all member clusters. As per the source code of KubeFed, $clusterCount \times log_{10}(replicas)$ iterations are required to distribute all replicas among the member clusters. The time complexity of this algorithm is $O(clusterCount^2 \times log_{10}(replicas))$, where $clusterCount$ is the number of member clusters and $replicas$ the requested number of containers. After determining the number of replicas per cluster, RSP updates the Federated Deployment Object's *override* field. The changes are then automatically pushed to each cluster by the Sync Controller as depicted in Figure 2.

In a geo-distributed federation setup, there are two ways in which instability may arise due to network delays or transient network failures:

1) **Reconciliation failure**: Push reconciliation requests to one or more member clusters may time out prematurely at the time of scheduling by RSP, in which case the sync controller tries to re-sync the resources until the desired state is achieved. If transient network failures continue to happen, it may take a long time for the reconciliation to terminate.
2) **Health check failure**: One or more member clusters may be declared *unhealthy* by the *kube-controller-manager* if health check requests time out, in which case RSP recalculates the distribution of replicas and rebalances them by moving replicas away from the now-unhealthy clusters to healthy ones. RSP re-syncs the resources to the unhealthy clusters, if they become healthy again. These actions may repeat over and over in network environments with a large number of transient failures.

The unstable behavior is manifested by the number of replicas on the affected member clusters being significantly fewer than the desired numbers, sometimes even reaching zero. Figure 4 shows the number of deployment replicas, which is the number of replicas pushed by the *kubefed-controller-manager*, and the actual number of running pods in one of the member clusters in our setup during a period of instability. As shown in the figure, the number of replicas that the *kubefed-controller-manager* pushes to the *member cluster* fluctuates widely over time, in turn affecting the number of pods actually running on the cluster.

To quantify the unstable behavior we introduce the *stability* metric υ as follows:

$$\upsilon[\%] := \frac{1}{n} \cdot \sum_{i=1}^{n} \left(100 - \frac{100}{T} \cdot \sum_{t=1}^{T} \frac{d_i - p_{i,t}}{d_i} \right) \qquad (1)$$

where n is the total number of *member clusters* and $i \in [1, n]$; T is the full experiment duration and time $t \in [0, T]$; d_i is the desired number of pods in cluster i; $p_{i,t}$ is the number of running pods in cluster i at time t. Stability is a measure of how much and for how long the number of replicas in the member clusters is close to the desired number of replicas: a system which fails to deploy any replica during the entire experiment will have $\upsilon = 0\%$ whereas a perfectly working and stable system will have $\upsilon = 100\%$.

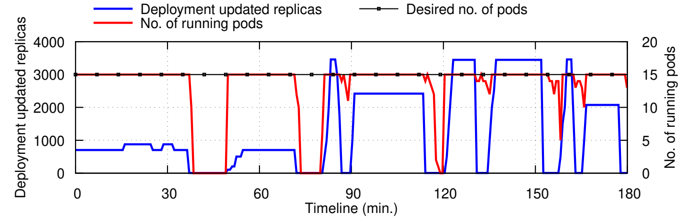


Fig. 4. The number of updated replicas of the deployment and the number of actual running pods on one of the member clusters of the federation.

C. The influence of configuration parameters

To identify which of KubeFed's configuration parameters has the greatest influence on the stability of the deployments in the member clusters, we conduct principal component analysis on the data obtained from the measurement of stability by varying the values of several parameters. We identified eight configuration parameters which might influence the behavior and stability of the system: timeout durations, health check periods, numbers of retries, etc. We then measured the stability derived from 705 randomly-chosen sets of parameter values.

Our results show that the first source of stability variations between different configurations can be attributed to a single parameter. Specifically, we observe instability mainly when the Cluster Health Check Timeout (CHCT) parameter has too low values. Even the default value of 3 s for this parameter leads to significant instability. We also notice that increasing the value of the CHCT parameter significantly improves the stability of the system.

D. Trade-off between instability and failure detection delay

Although the stability of the system improves when the value of the CHCT parameter is increased, setting very large values to the CHCT parameter leads to slower failure detection as the system needs to wait until the CHCT timeout expiration before it updates the status of the failed cluster as "Offline." As shown in Figure 5, increasing CHCT leads to greater system stability; however, it also increases the failure detection delay. The goal of our controller is to identify a sweet spot which implements the necessary trade-off between these two effects.

E. The influence of the networking environment

The last important factor which influences the occurrence of instability is the network performance between the clusters. Figure 6 depicts the stability of the system for the three scenarios. We see that the system is very unstable for the *Stationary scenario*. Moreover, stability gets worse as the network latency and packet loss is increased or one of the clusters fails in the *Network variability* and *Cluster failure*

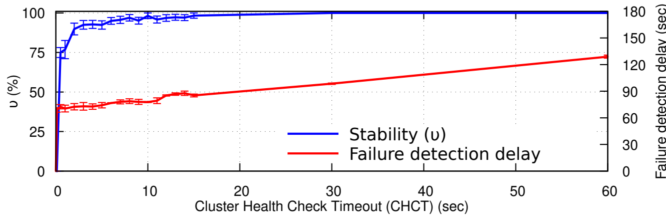


Fig. 5. Stability and failure detection delay as CHCT varies.

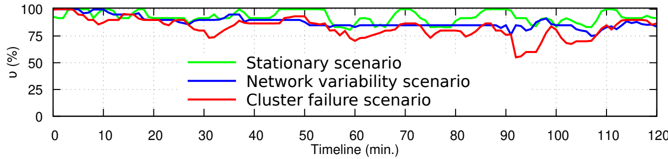


Fig. 6. Instability of the uncontrolled system under the three scenarios.

scenarios, respectively. Table II shows the system stability measures of the uncontrolled system under the three evaluation scenarios.

TABLE II
AVERAGE NO. OF TIMEOUT ERRORS PER MINUTE (N) AND STABILITY (υ) OF THE UNCONTROLLED SYSTEM FOR THE THREE EVALUATION SCENARIOS.

Experiment Scenario	Avg. N	Avg. υ (%)
Stationary	4	92
Network variability	3	87
Cluster failure	3	83

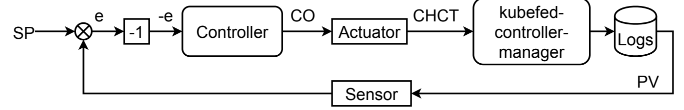


Fig. 7. System design of our controller.

V. A CONTROL-BASED APPROACH FOR TUNING CHCT

Since the CHCT parameter value is the cause of most instability problems, a natural solution would consist of finding a better value for this parameter. Generally speaking, the CHCT value should be set as low as possible (to reduce the delay in detecting actual cluster failures), but not too low either (because this would generate instability). However, no single "best" value can be found, as the choice of a good value largely depends on the operational conditions such as the inter-cluster network characteristics and the application workload.

Instead, we propose to dynamically adjust the CHCT value at runtime using a feedback controller which reacts to changes in operational conditions. Unlike other configuration tuning methods, this adaptive approach does not require prior knowledge of the infrastructure and it is simple to implement. In this section, we present the details of our solution including our design decisions, controller design, and tuning of the controller parameters.

A. Feedback controller design

Feedback controllers are widely used in mechanical and electrical systems, and they have also gained widespread use in computer systems [30]. A controller implements a feedback loop which monitors the system to be controlled, and implements automatic changes and then manipulates the input as needed to drive the system's variable toward the desired setpoint.

Figure 7 shows the design of our proposed solution. The controller continuously monitors the measured output (called Process Variable, PV, in control theory terminology) of the kubefed-controller-manager to detect indications that the CHCT value is either too high or too low. It then produces a signal called control output (CO) that reduces the error (e) that indicates the deviation of the measured output from the reference value. Finally, the controller decides whether CHCT must be adjusted, and the actuator implements the change.

a) Choosing the Process Variable: A naïve approach would consist of periodically evaluating the KubeFed stability metric, and of incrementing the CHCT value whenever the measured stability differs from the desired setpoint of 100% ($e > 0$). However, this would mean that the system must enter a period of instability before the CHCT value is updated. This reaction would be too late for our purpose which is precisely to prevent instabilities from occurring.

It is, therefore, necessary to base the controller reactions on other metrics which show early indications that instability is about to occur. For this we use the timeout errors written by the *kubefed-controller-manager* in its logfile whenever it fails to reach one of its member clusters. KubeFed starts deleting containers in the "failed" cluster and restarting them in other clusters when these timeout errors accumulate, thereby potentially triggering instability. We can thus use the occurrence of the first timeout errors in the log file as early indications that instability may soon take place.

Another motivation for selecting the number of timeout errors as the process variable PV is because this metric is readily available in the host cluster where we deploy our controller, unlike stability which needs to be computed after collecting metrics from each individual member cluster. Frequently collecting metrics from the member clusters may be very difficult, especially in periods of bad network performance when the CHCT value needs to be quickly adjusted.

b) Controlling the CHCT value: If no timeout errors are reported, then we know that the system should achieve 100% stability. We, therefore, define the setpoint SP to 0 timeout error. As a result, the controller increments the CHCT parameter value until the number of timeout errors found in the log files during the control interval reaches zero.

However, setting SP to zero creates a new problem. In the standard feedback control theory, one should allow both positive and negative errors so that the controller can automatically increase or decrease CO proportional to the error. In our case, since we define SP as zero, it is impossible to observe a number of timeout errors lower than the setpoint, and the controller cannot decrease the CHCT value as a result of such

Algorithm 1: Feedback controller algorithm.

Data: Positive Gain K_p, Negative Gain K_n
Result: CO
initialization;
SP := 0;
decrement_period := 3;
t := decrement_period;
while *true* **do**
    PV = number of timeout errors;
    e = -(SP - PV);
    CHCT = current value of the CHCT parameter;
    if $e > 0$ **then**
        CO = CHCT + K_p * e;
        t = decrement_period;
    else
        Do nothing;
    t = t - 1;
    if $t == 0$ **then**
        CO = (1 - K_n) * CHCT;
        t = decrement_period;

negative errors. As a result, even though we can increase CO proportionally to the error, for the decreasing part we need to deviate from the standard approach of feedback control design and come up with a different approach. For simplicity, we decided to decrease CHCT periodically if no timeout error has been identified, independently from any indication that the CHCT value may be too high. The controller ensures the trade-off between improving stability and fast detection of failures by preventing CHCT from reaching very large values that could lead to increase in the failure detection delay.

We choose a sampling interval of 1 minute for practical reasons. To change the CHCT value of a running KubeFed, it is necessary to stop and restart the containers which execute the *kubefed-controller-manager*. This operation takes a few dozen seconds. A sampling interval of 1 minute, therefore, gives enough time for the system to change the CHCT value before starting the next iteration of the control algorithm.

c) Control algorithm: Our control algorithm is presented in Algorithm 1. The controller periodically measures the number of timeout errors which occurred in the previous period, and compares it to the setpoint $SP = 0$. If timeout errors have occurred, then the controller increments CHCT by a value proportional to the number of timeout errors and to the positive gain K_p, which is in line with the standard design of a proportional feedback controller. On the other hand, if no timeout errors have been found during three consecutive periods, the controller decreases CHCT proportionally to the negative gain K_n.

The two gain parameters K_p and K_n respectively define how aggressive the controller should be in increasing and in decreasing the CHCT value.

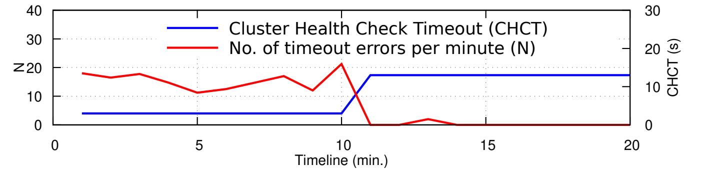


Fig. 8. Step response of KubeFed controller manager as the CHCT parameter is increased from the default 3 s to 13 s.

B. Tuning the controller parameters

Defining the controller parameters requires one to find a trade-off between a system which would react too slow to environment changes to provide an appropriate reaction and one which may potentially over-react to any such changes.

To get an initial estimate for K_p we use the Ziegler-Nichols rules, which are a set of simple heuristics that perform well a wide variety of situations [31]. The Ziegler-Nichols tuning method does not require detailed knowledge of the controlled system, and the rules can be expressed entirely in terms of the system's step-input response (i.e., the system's reaction characteristics upon a change of its parameter value) [30].

Figure 8 shows the step response of the system as the CHCT parameter is suddenly increased from the default value of $\textit{CHCT} = 3\,sec$ to $\textit{CHCT} = 13\,sec$ at time $t = 10\,min$. From the system's step response, we estimate three parameters that are used in the Ziegler Nichols tuning rules:

- The process gain K is the ratio of the change in process output ΔPV that results from a change of input ΔCO:

$$K = \frac{\Delta PV}{\Delta CO}$$

- The time constant T is the time it takes for the process to settle to a new steady-state after experiencing a sudden change in input, i.e., the time it takes the process to reach about two-thirds of its final value.
- The dead time τ is the delay until an input change begins to affect the output.

From Figure 8, we find $K = 1.5$, $T = 60\,s$, and $\tau = 60\,s$. Based on these step-input response values we can define K_p:

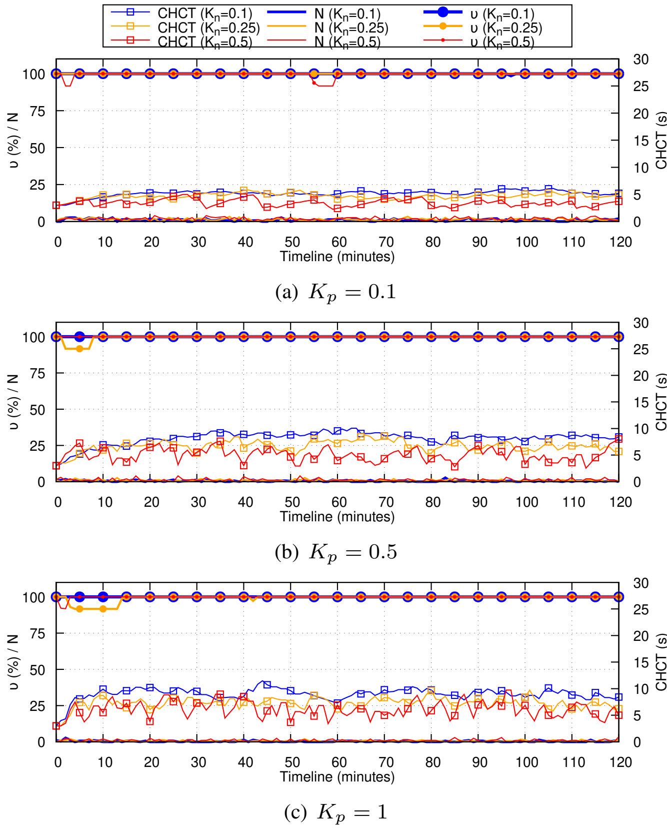
$$K_p = \alpha \times \frac{T}{K \times \tau}$$

where α is a coefficient which typically falls in the range $[0.3, 1.2]$ [30]. We can therefore estimate that K_p should fall in the range $[0.2, 0.8]$. Based on these estimations, in the next section we experiment the controller with K_p values of 0.1, 0.5, and 1. Similarly, we use K_n values of 0.1, 0.25, and 0.5.

VI. Evaluation

A. Experimental Setup

We evaluate our controller using the same experimental setup as described in Section IV-A with the same application and workload for the three scenarios. However, now we also deploy our controller on the master node of the host cluster. We run a total of nine experiments per scenario, one for each combination of the K_p and K_n parameters of the controller.

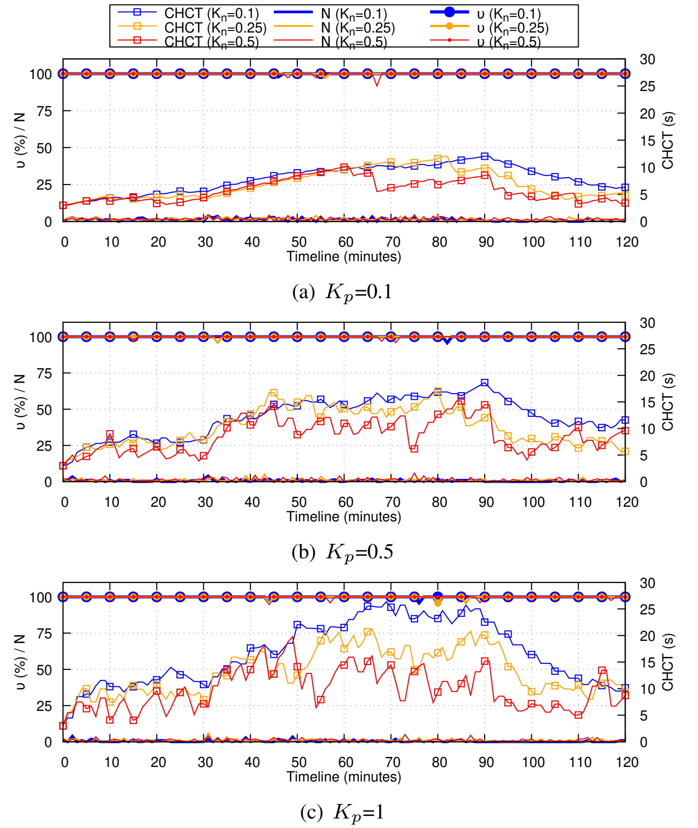


(a) $K_p = 0.1$

(b) $K_p = 0.5$

(c) $K_p = 1$

Fig. 9. Stationary scenario with different values of K_p.



(a) K_p=0.1

(b) K_p=0.5

(c) K_p=1

Fig. 10. Network variability scenario: network latency increases at $t = 30\,min$ and decreases back at $t = 90\,min$.

Each experiment is run for two hours. We repeat each of the 27 experiments three times and report the mean value.

B. Experimental Results

We present the results of the stationary, network variability and cluster failure scenarios in Figures 9, 10 and 11 respectively. In all scenarios, the controller adjusts CHCT according to the conditions, and significantly improves the federation stability compared to the no-controller scenario from Figure 6.

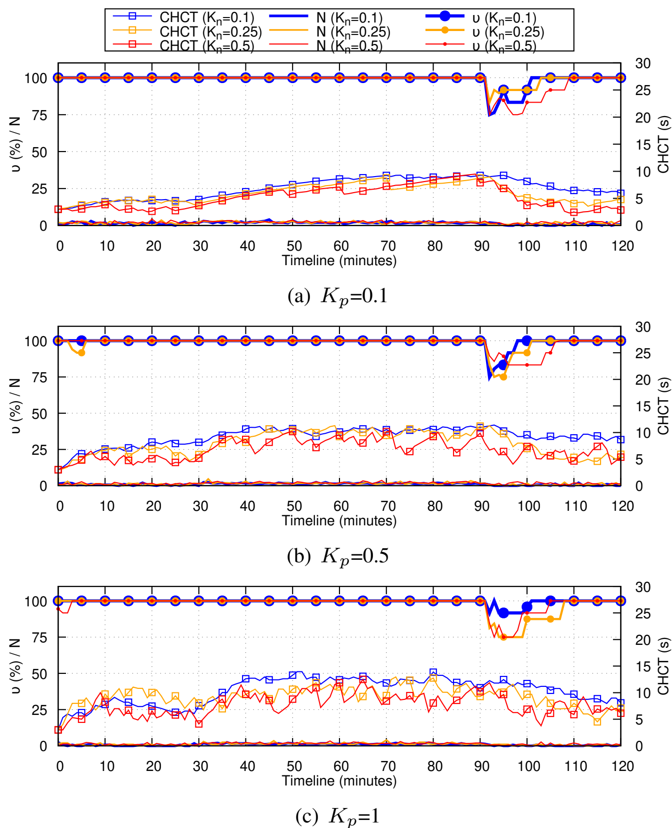
In the *Network variability scenario*, CHCT increases from $t = 30\,min$ as a reaction to the degraded networking performance, and decreases back at $t = 90\,min$ when network performance returns to normal. Similarly, in the *Cluster failure scenario*, CHCT increases after $t = 30\,min$ as a reaction to the detected cluster failure, and decreases from $t = 90\,min$ after cluster recovery. The stability drop at $t = 90\,min$ is a direct consequence of cluster recovery, as several pods get stopped in other clusters and restarted in the recovered one.

In all scenarios, we see a faster increase of the CHCT parameter as K_p increases, and faster decrease as K_n increases. In some cases, the larger values of K_n lead to a brief instability as the CHCT parameter is aggressively decreased to very low values, leading to timeouts.

To determine values of K_p and K_n which work in all three scenarios, we compare all 27 cases for accuracy. Table III shows the accuracy of the controller in decreasing the number of timeout errors N, and in improving the stability υ. For each scenario, we show the best values for N and υ in **bold** and the worst values in *italic*. We see that the controller with K_p value of 1 and K_n value of 0.1 has the best values for N in all three scenarios, and the best value of υ in two out of three scenarios. Thus, we conclude that the controller works best in all three scenarios for this combination of values of the parameters K_p and K_n. This configuration improves stability from 83–92% with no controller (see Table II) in stationary situations to 99.5–100% using the controller, even in challenging scenarios with network variability or cluster failures.

TABLE III
ACCURACY OF THE PROPOSED CONTROLLER AS THE VALUES OF K_p AND K_n VARY IN THE THREE SCENARIOS. THE BEST VALUES ARE **BOLD**, WHEREAS THE WORST VALUES ARE *Italic*.

K_p	K_n	Stationary scenario		Network variability scenario		Cluster failure scenario	
		N	υ (%)	N	υ (%)	N	υ (%)
0.1	0.10	0.98	99.99	1.18	**99.98**	1.55	98.77
	0.25	*1.41*	**100.00**	1.40	99.98	1.85	99.16
	0.50	0.87	99.91	*1.64*	*99.89*	*2.11*	97.71
0.5	0.10	0.48	**100.00**	0.64	99.97	1.05	99.22
	0.25	0.78	99.59	0.90	99.97	1.28	98.66
	0.50	1.12	**100.00**	1.19	99.92	1.63	98.39
1	0.10	**0.31**	**100.00**	**0.46**	99.93	**0.82**	**99.48**
	0.25	0.55	*99.22*	0.75	99.90	1.10	*97.65*
	0.50	0.86	99.87	1.00	99.91	1.36	98.14



(a) K_p=0.1

(b) K_p=0.5

(c) K_p=1

Fig. 11. Cluster failure scenario: one cluster fails at $t = 30\,min$ and recovers at $t = 90\,min$.

VII. Conclusion

Geo-distributed systems such as fog computing platforms need to operate in difficult and uncertain networking conditions. In particular, it is notoriously difficult to distinguish actual node failures from delays caused by the networking or local node condition. We demonstrated that these effects can create significant instability in Kubernetes Federations. We identified the main configuration parameter which influences this behavior, and proposed a feedback controller which dynamically adapts its value to the operational conditions, and improves the system stability from 83–92% with no controller to 99.5–100% using the controller.

References

[1] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards network-aware resource provisioning in Kubernetes for fog computing applications," in *Proc. IEEE NetSoft*, 2019.

[2] F. Rossi, V. Cardellini, F. Lo Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with Kubernetes," *Computer Communications*, vol. 159, 2020.

[3] C. Wöbker, A. Seitz, H. Mueller, and B. Bruegge, "Fogernetes: Deployment and management of fog computing applications," in *Proc. IEEE/IFIP NOMS*, 2018.

[4] S. Noghabi, L. Cox, S. Agarwal, and G. Ananthanarayanan, "The emerging landscape of edge computing," *GetMobile: Mobile Computing and Communications*, vol. 23, no. 4, 2020.

[5] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang, "BestConfig: Tapping the performance potential of systems via automatic configuration tuning," in *Proc. ACM SoCC*, 2017.

[6] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *ACM Queue*, vol. 14, no. 1, 2016.

[7] Kubernetes SIG Multicluster, "Kubernetes cluster federation," https://github.com/kubernetes-sigs/kubefed, 2020.

[8] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, 2003.

[9] A. G. Ganek and T. A. Corbi, "The dawning of the autonomic computing era," *IBM systems Journal*, vol. 42, no. 1, 2003.

[10] H. Chen, G. Jiang, H. Zhang, and K. Yoshihira, "Boosting the performance of computing systems through adaptive configuration tuning," in *Proc. ACM SAC*, 2009.

[11] W. Zheng, R. Bianchini, and T. D. Nguyen, "MassConf: Automatic configuration tuning by leveraging user community information," in *Proc. ACM/SPEC ICPE*, 2011.

[12] S. Bouchenak, N. De Palma, D. Hagimont, and C. Taton, "Autonomic management of clustered applications," in *Proc. IEEE Cluster*, 2006.

[13] M. Li, L. Zeng, S. Meng, J. Tan, L. Zhang, A. R. Butt, and N. Fuller, "MRONLINE: MapReduce online performance tuning," in *Proc. ACM HPDC*, 2014.

[14] G. Wang, J. Xu, and B. He, "A novel method for tuning configuration parameters of Spark based on machine learning," in *Proc. IEEE HPCC/SmartCity/DSS*, 2016.

[15] D. van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," in *Proc. ACM SIGMOD*, 2017.

[16] L. Bao, X. Liu, Z. Xu, and B. Fang, "AutoConfig: Automatic configuration tuning for distributed message systems," in *Proc. ASE*, 2018.

[17] D. Menascé, D. Barbará, and R. Dodge, "Preserving QoS of e-commerce sites through self-tuning: A performance model approach," in *Proc. ACM EC*, 2001.

[18] N. Gandhi, D. M. Tilbury, Y. Diao, J. Hellerstein, and S. Parekh, "MIMO control of an Apache web server: Modeling and controller design," in *Proc. IEEE ACC*, 2002.

[19] Y. Diao, N. Gandhi, J. L. Hellerstein, S. Parekh, and D. M. Tilbury, "Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache web server," in *Proc. IEEE/IFIP NOMS*, 2002.

[20] K. Ye and Y. Ji, "Performance tuning and modeling for big data applications in Docker containers," in *Proc. NAS*, 2017.

[21] T. Chiba, R. Nakazawa, H. Horii, S. Suneja, and S. Seelam, "ConfAdvisor: A performance-centric configuration tuning framework for containers on Kubernetes," in *Proc. IEEE IC2E*, 2019.

[22] W. Chen, S. Toueg, and M. K. Aguilera, "On the quality of service of failure detectors," *IEEE Transactions on computers*, vol. 51, no. 5, 2002.

[23] F. Lima and R. Macêdo, "Adapting failure detectors to communication network load fluctuations using SNMP and artificial neural nets," in *Proc. LADC*, 2005.

[24] R. C. Nunes and I. Jansch-Porto, "QoS of timeout-based self-tuned failure detectors: The effects of the communication delay predictor and the safety margin," in *Proc. DSN*, 2004.

[25] L. Falai and A. Bondavalli, "Experimental evaluation of the QoS of failure detectors on wide area network," in *Proc. DSN*, 2005.

[26] A. S. de Sá and R. J. A. Macêdo, "QoS self-configuring failure detectors for distributed systems," in *Proc. IFIP DAIS*, 2010.

[27] D. Balouek, A. Carpen-Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lebre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*, vol. 367, Springer, 2013.

[28] F. P. Tso, D. R. White, S. Jouet, J. Singer, and D. P. Pezaros, "The Glasgow Raspberry Pi cloud: A scale model for cloud computing infrastructures," in *Proc. ICDCS Workshops*, 2013.
[29] A. van Kempen, T. Crivat, B. Trubert, D. Roy, and G. Pierre, "MEC-ConPaaS: An experimental single-board based mobile edge cloud," in *Proc. IEEE Mobile Cloud Conference*, 2017.
[30] P. K. Janert, *Feedback control for computer systems: introducing control theory to enterprise programmers*. O'Reilly Media, Inc., 2013.
[31] J. G. Ziegler and N. B. Nichols, "Optimum settings for automatic controllers," *Trans. of the ASME*, vol. 64, 1942.