Distributed Fault-tolerant Platform for Web Applications

Mikhail M. Rovnyagin[1], Dmitry M. Sinelnikov[2], Viktor V. Odintsev[3], Sergey S. Varykhanov[4]
National Research Nuclear University MEPhI (Moscow Engineering Physics Institute)
Moscow, Russia
[1]mmrovnyagin@mephi.ru, [2]dsinelnikov96@gmail.com, [3]zakhams@gmail.com, [4]masmx64@gmail.com

***Abstract*—Web applications are software applications, services or microservices that runs on a remote server. The problem of downtime for web application is important and in some cases, is critical for business. Nowadays, cluster solutions are often used to provide fault-tolerance for applications. But these solutions don't solve the problem of downtime if all instances of application are down. This paper presents a complex approach to provide fault-tolerance for web applications even if all instances of applications in the cluster are down. The approach is based on long-polling and request queueing methods. In this work Apache Kafka and Google Protocol Buffers has been used as the core for the fault-tolerant platform.**

Keywords—docker, fault-tolerance, message broker, Apache Kafka, proxy

I. INTRODUCTION

Nowadays, there are many tools for combining computers in a cluster. Clusters are used to perform big data processing tasks, deployment of fault-tolerant database and microservices architecture.

The most popular, convenient and at the same time reliable tool for deployment automation and management of applications on the server, at the moment, is Docker.

Docker allows to "pack" the application with all its environment and dependencies into a container that can be ported to any Linux system with cgroups support in the kernel, and also provides a container management environment.

In the middle of 2014, Google first announced Kubernetes [1] - an open source platform for automatic deployment, scaling and containerized application management.

In addition, the June 2016 release of Docker 1.12 was a massive event, mainly due to the support of Swarm mode. This allowed to manage Docker containers on the cluster using Docker itself.

Today, there are many tools that provide a simple and convenient solution for combining servers into clusters.

The main problem of existing tools is the lack of instruments to provide the updating or transfer of the containers without losing connections.

In addition, unfortunately, when moving containers from one cluster node to another, container data is lost, because Docker containers use stateless technology.

II. RELATED WORKS

The main goal in the developing of fault-tolerant systems is to reduce the maximum system downtime. Downtime is a period of time when the system is unavailable for use or does not respond to requests. High downtime value can lead to huge business losses.

There are two types of downtime - planned and unplanned [2]. Planned downtime is the result of maintenance that is inevitable. It includes applying patches, updating software, or even changing a database schema. Unplanned downtime, in turn, is caused by some unforeseen circumstances, for example, hardware or software failures.

System availability can be measured as a percentage of the time that the system is available [3]:

$$x = \frac{(t_m - t_{mu})}{t_m} * 100 \quad (1)$$

where t_m — number of minutes in a calendar month and t_{mu} — number of minutes, when service is unavailable in the given calendar month.

Clustering is creating multiple replicas of the application on different servers [4][5]. The cluster includes many nodes that exchange information through shared data. Thus, any node can be disconnected from the network, while the remaining nodes in the cluster will continue to function normally [6].

Load balancing [7] can effectively increase the availability of critical applications. When it is discovered that one of the copies of the application is unavailable, the traffic is automatically redistributed to other replicas [8]. For load balancing, various algorithms are used. The most common are Round Robin and its Weighted Round Robin modification [9]. The main objective of such algorithms is the uniform load distribution between the cluster nodes.

Using a backup server allows a quick switch to a backup replica of the application in the event of a failure of the main replica. There are three types of the backup servers: cold, hot and warm [10]. A cold backup server starts only after a failure of the primary server. A hot backup server starts and works

 978-1-7281-5761-0/20/$31.00 ©2020 IEEE

with the main one. In a warm backup, the server is powered on, but not performing any work, or it is turned on from time to time to get updates from the server being backed up.

Geographic replication [11] is used to protect against service failures in the time of catastrophic events, such as natural disasters, which can lead to server crashes. The main idea is to run several independent servers, one in each location. Thus, in the event of a server failure in one geographic location, the rest will continue to work properly.

Often, the approaches described above are overlapped and combined to achieve greater availability. For example, Google's systems use backups, clustering, load balancing, backup servers and geographic replication [12].

To estimate fault tolerance of the system, special software is often used, that physically interferes with the operation of the system under test [13]. Such software reboots the cluster nodes, overloads the network and puts the system under high loads, creating critical situations.

III. Design modelling as a queue system

In this article, we propose a way to increase the fault tolerance of connections by using a connection queue: the proxy sends requests to the desired queue, which forwards this request to the service, if it is available. In case of service failure, the queue simply accumulates messages in itself, after restoration service all accumulated messages from the queue fall into this service and the system continue to work properly.

In terms of queuing theory [14], a multichannel queuing system (QS) with balking and reneging is considered. There are n channels (services) which are receiving the flow of requests with the intensity λ (arrival rate). The service flow has an intensity of μ (service throughput) [15].

As known, the probability of a QS rejection is the probability that all n channels of the system will be occupied. From the Erlang formula [16], the probability of QS rejection is:

$$P_f = \frac{\rho^n}{n!} P_0 \tag{2}$$

There is a formula to determine the probability that there are 0 customers in the system for queues with dependence on state of system:

$$P_0 = (1 + \rho + \frac{\rho^2}{2!} + \cdots + \frac{\rho^k}{n!} + \cdots + \frac{\rho^n}{n!})^{-1} \tag{3}$$

$$P_0 = \frac{1}{\sum_{i=0}^{n} \frac{\rho^i}{i!}} \tag{4}$$

Where the channel utilization is expressed as follows:

$$\rho = \frac{\lambda}{\mu} \tag{5}$$

As you know, factorial grows faster than an exponential function. Thus, in case of failure of one of the channels, the probability of rejection of the entire system increases:

$$P_f = \frac{\rho^n}{n!} \frac{1}{\sum_{i=0}^{n} \frac{\rho^i}{i!}} \tag{6}$$

In this case, the relative throughput of the system decreases:

$$Q = 1 - P_f \tag{7}$$

To solve this problem, in this paper we propose to apply a buffer queue for each of the services. Thus, in terms of queuing theory, queues become channels, and when using a fault-tolerant queue, the number of channels n remains unchanged when a service or even a cluster node unavailable.

The queue requirements are as follows:

- Data storage on a disk, but not in random access memory (RAM). Since, the RAM is cleared when the node is restarted. Also with a large number of service requests, memory leaks may occur.
- Fault tolerance.

The Apache Kafka [17] platform meets all the requirements and therefore has been chosen as a queue for the fault tolerant platform.

IV. Architecture Design

Based on these studies, this article proposes a platform architecture to solve the problem of service failures. The proposed solution is a proxy service with queues.

The general system architecture is shown in the figure 1.

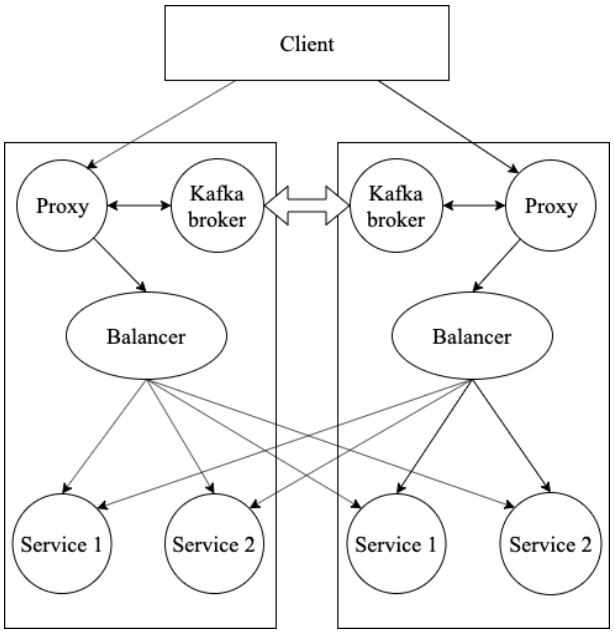

Fig. 1. The scheme of common architecture

Requests from the client come to the proxy, which is deployed on all nodes to provide fault tolerance. The proxy sends requests to the queue, then fetches them from the queue

and sends them to the balancer. Thus, all requests are stored in the queue.

The balancer distributes the load between services deployed on different nodes. In case of failure of one of the service instances, the balancer starts sending requests to the second instance. If all service instances are down, requests will accumulate in the queue and, after the service is restored, will be sent to it.

Thus, the platform provides stable operation when a cluster node is down or one of the services is down.

To provide parallel processing of connections, it is necessary to run several handlers and balance the connections between them.

The connection balancing scheme for the handlers is shown in the figure 2.

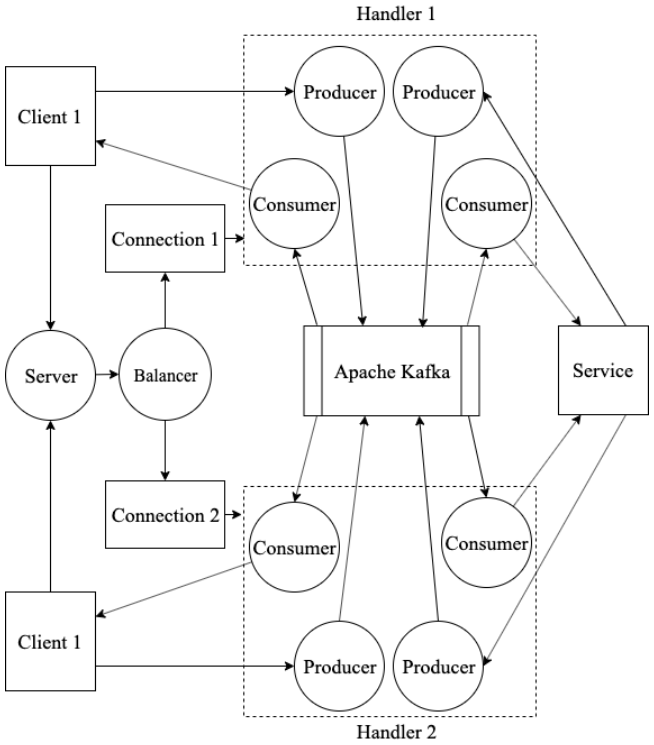

Fig. 2. The scheme of connection balancing between handlers

The process of creating a connection consists of the following steps:

1) The client connects to the server.
2) The server transfers this connection to the balancer.
3) The balancer builds a connection by connecting to the client and service.
4) The balancer transfers the connection to a free handler or waits until one of them will be released.
5) The handler begins to receive data from the client and transfers it to the service through a message broker.

The general architecture of the developed platform is shown in the figure 3.

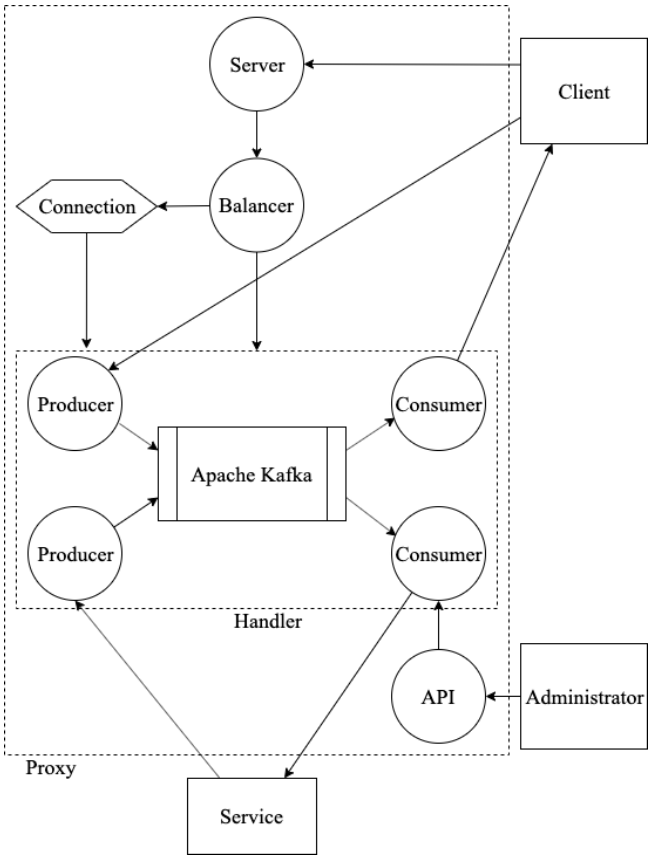

Fig. 3. The scheme of platform architecture

The server receives requests from the client and sends them to the balancer. When the balancer receives a request from the server, it creates a new connection and passes it to a free handler. The handler provides data transfer between the client and the service through a message broker.

The API module provides HTTP methods for the proxy management for the administrator. One of these methods is to suspend receiving of new messages from the message broker.

Thus, the platform provides the possibility to transfer or update the service without losing connections.

V. Experiments

To test the workflow of the implemented platform, a test case was developed. The test case consists of the following steps:

1) Kubernetes Cluster deployment.
2) Deployment of a service's container with volume in Kubernetes cluster (NoSQL Redis database [18] was used for testing).
3) The Apache Kafka cluster deployment.
4) Configuration of proxy in Zookeeper.
5) Deployment of the developed proxy.
6) Checking service availability through proxy (the service should be available).
7) Changing the maintenance flag from false to true in Zookeeper.

8) Checking service availability through proxy (service requests should be sent, but the response should not be received).
9) Turning off the service container.
10) Migrating container's volume to the second node of the cluster.
11) Launching the service on the second node of the cluster.
12) Changing the maintenance flag from true to false in Zookeeper.
13) Checking service availability through proxy (the service should be available).
14) Checking the service data was saved after the transfer.

The developed system has successfully passed all the tests.

As part of the work, load testing experiment of the developed platform was also performed. Experiment was performed on a cluster consisting of two nodes using the Apache AB tool [19].

During the experiment, the HTTP request time and the maximum number of requests per second (RPS), depending on the server load, were measured.

Comparison was made with the HAProxy proxy server [20]. Httpbin has been chosen as a service, responding to requests [21].

The results of measuring the HTTP request are shown in the table I.

TABLE I. HTTP REQUEST TRANSMISSION TIME TEST RESULTS

Server	HTTP request time, ms
Proxy	13
HAProxy	9

The results of measuring requests per second, depending on the server load, are shown in the figure 4.

As can be seen from the test results, there is a slight decrease in performance compared to HAProxy. The main reason for this are delays that occur on connecting and transferring data through the Apache Kafka.

VI. CONCLUSION

In this work, an analysis was carried out and a distributed fault-tolerant platform was developed. According to the queueing theory and experiments this platform provides fault-tolerance for web services due to usage of Apache Kafka platform as a channel for HTTP requests. Although, this solution has an affect on the performance of HTTP requests, it provides fault-tolerance, even if some web services were down for a while.

REFERENCES

[1] N. Poulton, *The Kubernetes book*. United Kingdom: Nigel Poulton, 2018, ISBN: 978-1521823637.

[2] E. Marcus, *Blueprints for high availability*. New York Chichester: Wiley, 2003, ISBN: 978-0471430261.

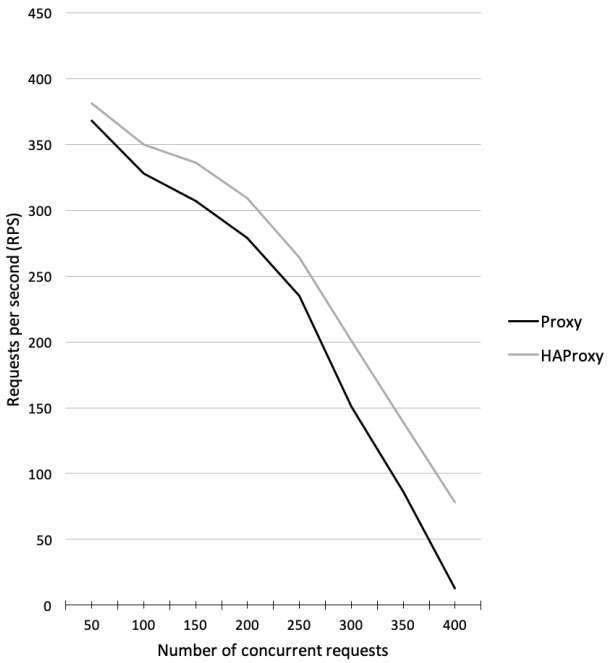

Fig. 4. Dependence of Requests Per Second on number of concurrent requests

[3] F. Piedad and M. Hawkins, *High Availability: Design, Techniques, and Processes*. Prentice Hall, 2001.

[4] A. A. Kozlov, A. A. Aleshina, I. S. Kamenskikh, M. M. Rovnyagin, D. M. Sinelnikov and D. A. Shulga, "Increasing the functionality of the modern NoSQL-systems with GPGPU-technology," *2016 IEEE NW Russia Young Researchers in Electrical and Electronic Engineering Conference (EIConRusNW)*, St. Petersburg, 2016, pp. 242-246. DOI: 10.1109/EIConRusNW.2016.7448164.

[5] M. M. Rovnyagin, V. V. Odintsev, D. Y. Fedin and A. V. Kuzmin, "Cloud computing architecture for high-volume monitoring processing," *2018 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, Moscow, 2018, pp. 361-365. DOI: 10.1109/EIConRus.2018.8317107.

[6] A. B. M. Moniruzzaman and S. A. Hossain, "A low cost two-tier architecture model for high availability clusters application load balancing" 2014. arXiv: 1406.5761 [cs.NI].

[7] I. S. Kamenskikh, D. M. Sinelnikov, D. S. Kalintsev, A. A. Kozlov, M. M. Rovnyagin and D. A. Shulga, "Software development framework for a distributed storage and GPGPU data processing infrastructure" *2016 IEEE NW Russia Young Researchers in Electrical and Electronic Engineering Conference (EIConRusNW)*, St. Petersburg, 2016, pp. 216-219. DOI: 10.1109/EIConRusNW.2016.7448158.

[8] M. M. Rovnyagin, S. S. Varykhanov, Y. V. Maslov, I. S. Riakhovskaia and O. V. Myltsyn, "NFV chaining technology in hybrid computing clouds," *2018 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, Moscow, 2018, pp. 109-113. DOI: 10.1109/EIConRus.2018.8317042.

[9] W. Wang and G. Casale, "Evaluating weighted round robin load balancing for cloud web services" in *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, Sep. 2014, pp. 393–400. DOI: 10.1109/SYNASC.2014.59.

[10] A. Dudin, V. Klimenok, and V. Vishnevsky, "Analysis of unreliable single server queueing system with hot back-up server" in *Optimization in the Natural Sciences*, A. Plakhov, T. Tchemisova, and A. Freitas, Eds., Cham: Springer International Publishing, 2015, pp. 149–161, ISBN: 978-3-319-20352-2.

[11] T. Amjad, M. Sher, and A. Daud, "A survey of dynamic replication strategies for improving data availability in data grids" *Future Generation Computer Systems*, vol. 28, no. 2, pp. 337–349, 2012, ISSN: 0167-739X.

[12] A. Gupta and J. Shute, “High-availability at massive scale: Building Google’s data infrastructure for Ads” in *Workshop on Business Intelligence for the Real Time Enterprise (BIRTE)*, 2015.

[13] D. Powell, E. Martins, J. Arlat, and Y. Crouzet, “Estimators for fault tolerance coverage evaluation” in *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*, Jun. 1993, pp. 228–237. DOI: 10.1109/FTCS.1993.627326.

[14] V. Sundarapandian, *Probability, statistics and queuing theory*. New Delhi: PHI Learning, 2009, ISBN: 978-8120338449.

[15] S. H. Kamali, M. Hedayati, A. S. Izadi, and H. R. Hoseiny, “The monitoring of the network traffic based on queuing theory and simulation in heterogeneous network environment” in *2009 International Conference on Computer Technology and Development*, vol. 1, Nov. 2009, pp. 322–326. DOI: 10.1109/ICCTD.2009.242.

[16] U. Bhat, *An introduction to queueing theory: modeling and analysis in applications*. Boston, MA: Birkhäuser, 2015, ISBN: 978-0817684204.

[17] T. Shapira, *Kafka: the Definitive Guide*. O’Reilly Media, Inc, 2017, ISBN: 9781491936153.

[18] J. Carlson, *Redis in action*. Shelter Island, NY: Manning, 2013, ISBN: 978-1617290855.

[19] ab - Apache HTTP server benchmarking tool, [Online]. Available: https://httpd.apache.org/docs/current/programs/ab.html (visited on 09/21/2019).

[20] N. Ramirez, *Load balancing with HAProxy : open-source technology for better scalability, redundancy and availability in your IT infras-tructure*. 2016, ISBN: 978-1519073846.

[21] httpbin - HTTP Request & Response Service, [Online]. Available: http://httpbin.org (visited on 09/22/2019).