A scalable distributed system based on microservices for collecting pod logs from a Kubernetes cluster

Flávio Gomes da Silva Lisboa
Universidade Tecnológica Federal do Paraná
Curitiba, Brasil
https://orcid.org/0000-0002-9396-7944

***Abstract*—This article presents the architecture of a distributed log collection system for Kubernetes clusters. Initially, we present the motivation for creating the system. Then, we present an overview of the system, consisting of several microservices. Next, we approach the implementation of each of the microservices. All system components are free and open software. This article is an example of how a distributed system with microservices can be heterogeneous in relation to the programming languages used.**

***Keywords*—container; kubernetes; microservice; log.**

I. INTRODUCTION

Kubernetes is an open source container orchestration system created by Google. It allows a complex software system to be distributed in units called Pods, which are sets of containers with shared storage and network resources. Kubernetes has an abstraction called Deployment, which allows you to define a desired state for a Pod. This includes maintaining Pod replicas to ensure that the microservice it contains is always available. From liveness tests, a Deployment can restart a Pod, without the need for human intervention.

When a Pod is created, it is assigned an IP address. When it restarts, this address changes. Kubernetes can restart a Pod as many times as necessary, which is convenient to spare full-time technicians from monitoring applications that should be available 24 by 7. It is not desirable, however, for a Pod to be constantly restarted, as this suggests that there is a frequent fault condition.

Eventually, this failure condition can go unnoticed for a long time due to the automatic Pod restart performed by Deployment. Over time, a change combined with this fault condition can generate a state where the Pod is constantly restarted, as the liveness rule is never satisfied. The problem we want to solve here is the increasing complexity of a failure in a distributed system, characteristic of systems hosted on Kubernetes clusters.

To avoid this kind of situation, we can monitor the pod logs to check the frequency of changing IP addresses. In this article we present a scalable and distributed open source system for collecting logs from Kubernetes pods, filtering only the events involving IP change.

II. ARCHITECTURE OF THE SYSTEM

The system described here is called **podips**. It consists of the following components, all of them implemented with free and open source softwares:

- The **functional core**, formed by microservices **podips-reader**, **podips-queue** and **podips-writer**;
- The **monitoring panel**, which is the microservice **podips-monitor**;
- The **continuity guarantor**, which is the microservice **podips-cronjob**.

A Golang program (**podips-reader**), continuously reads the pod modification events of a Kubernetes cluster. At each reading, it sends these events to a queue implemented in ActiveMQ (**podips-queue**). A Python program (**podips-writer**) watches the queue and sends the read data to a Fluentd server, a data collector that aggregates pod logs with other logs in a non-SQL database for auditing. The reading and writing programs were originally written in Python. The reason the **podips-reader** was rewritten in Go is because the Kubernetes client in Python had failures in reading the cluster, which were not resolved. The **podips-writer** was kept in Python because trying to write to Fluentd with the driver in Go failed.

To ensure system availability, the **podips-reader** and **podips-writer** program deployments have *liveness probes* configured, to verify that queue writing and reading are taking place. Verification is based on changing status files at an interval of 5 minutes. That is, if the status files are not updated in at least 5 minutes, the deployments will be automatically restarted by Kubernetes.

To ensure that there is no loss of messages on the producer side (**podips-reader**), a PostgreSQL database is used, which stores messages that cannot be sent to the queue. This bank is continually read to check for messages and re-attempted to send them. This ensures that messages will not be lost until the queue is available again. Messages are stored in a table *messages* of a database *queue*. This table must have the fields *id* (serial) and *message* (text with up to 1000 characters). It is possible to manage the database through a PostgreSQL web client installed in the environment, via the URL /phppgadmin.

It's important to say that if **podips-writer** can't send the message to Fluentd, it sends the message back to the queue.

To help observe system availability, **podips-monitor** provides a web interface, which allows you to check the status of Kubernetes read, queue write and read, and send data to Fluentd. The **podips-reader** and **podips-monitor** programs send HTTP messages to **podips-monitor**, which updates internal status files with the date and time of the operation. The podips system operates on two Kubernetes clusters. So there are two URLs for **podips-monitor**.

To ensure that there is always at least one pod changing within the 5 minute interval, there is the program in Node.JS **podips-cronjob** which is registered as cronjob, running every 5 minutes. The reason **podips-cronjob** is written in Node.JS is because the policy on Kubernetes clusters where podips is installed restricts the programming languages that can be used in cronjobs.

The Figure 1 illustrates the relationship among the microservices of podips, focusing the flow of messages. Every microservice os podips is deployed in the same cluster Kubernetes they are reading.

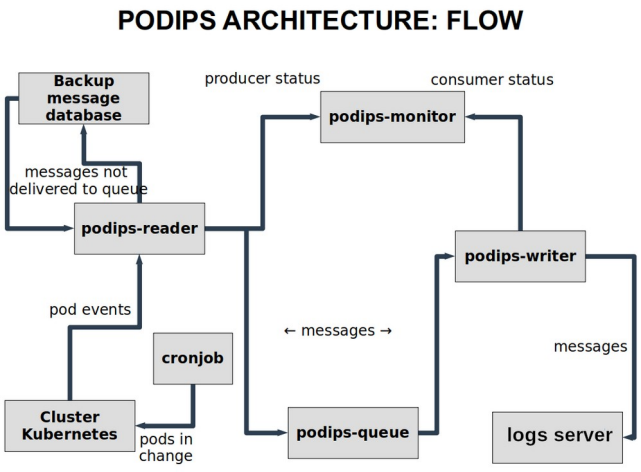

Fig. 1. Architecture of podips focusing the data flow

Next, we will detail each of the system's microservices.

III. READER OF LOGS FROM KUBERNETES

The microservice **podips-reader** is a Go program composed by one only source code file in Go language (podips-reader.go) and a configuration file (dbconfig.ini). It is licensed under LGPL-2.1 and is available at https://github.com/fgsl/podips-reader. This program can be executed inside or outside a Kubernetes cluster, but it is recommended to run it inside the cluster to take advantage of the automatic restart functionality.

The program configuration can be done by environment variables or by configuration files. Environment variables take precedence over configuration files. In this way, the environment determines the behavior of the program, and it is not necessary to package configuration files when publishing the program as a microservice.

The *main* function of this program consists of a repeat loop that reads the status of the pods every ten seconds. If the reading is successful, another loop is executed, to go through the collection of events occurred with the pods, and send to a queue the events that refer to the IP change.

When events are read successfully, an HTTP message is sent to a success monitoring endpoint and a local status file is created. If there is a failure, another message is sent to a failure monitoring endpoint and the local status file is destroyed. This allows Kubernetes to restart the program pod if the status file does not exist. Of course, this check has to be done in a time frame within which we tolerate read failures.

The **podips-reader** program defines a structure called *PodInfo* to encapsulate the data that must be sent to the queue. This structure prevents the various functions in the program from having multiple arguments and makes the program more readable.

The program is modularized into several functions for ease of understanding and maintenance. Within the repeat loop that processes the events of read pods, an object of type *PodInfo* is populated by the *getPodStateTerminatedAndKind* and *getPodStatusAndSendLog* functions. The first checks if the pod has finished while the second checks if the event should be sent to the log. It is the second function that sets the sendLog attribute of the PodInfo object and thus determines whether the program will try to send the log to the queue.

If the program fails to send a message to the queue, the *saveMessageToDatabase* function is invoked to persist the message to a relational database. The main loop in the main function calls the *readMessagesFromDatabase* function every iteration to check for persisted messages and tries to send them again. The order in which the messages are sent is not important, as the date and time of the event is part of the content. Thus, the logs viewer tool can sort the messages correctly.

IV. WRITER OF LOGS INTO FLUENTD

The microservice **podips-writer** is a Python program composed by one only source code file, licensed under LGPL-2.1 and available at https://github.com/fgsl/podips-writer. It is an event-oriented program. It defines a *LoggerListener* class that is a listener for the message queue's *on_message* and *on_error* events. A repeat loop keeps the program running, waiting for the queue to notify you of the receipt of messages. When a message is received by the queue, the program tries to read it and send it to Fluentd.

The *LoggerListener* class tries to send logs to Fluentd when it takes them off the queue. On success, it sends a

success message to the monitoring endpoint. In case of failure, it sends a failure message to the monitoring endpoint and puts the message back on the queue.

V. Monitor of Reading and Writing

A web application called **podips-monitor** serves as a monitoring tool for podips-reader and podips-writer programs. These two programs write various messages to the system log, which allows you to follow in real time what they are doing, but podips-monitor presents a synthetic view indicating whether the read and write operations are successfully occurring at any given time.

The **podips-monitor** application is a microservice written in PHP with the Mezzio framework. This framework works with the concept of middlewares, defined in the PSR-15 specification [1]. Mezzio lets you connect URLs with classes designed to process HTTP requests in a pipelined structure. Mezzio works with the whitelist concept at the HTTP method level, so that permission to send an HTTP message must be explicitly granted for each method (GET, POST, PUT, DELETE...). This implies that a Mezzio application only admits requests to an endpoint that are configured with the specified HTTP methods.

The **podips-monitor** application uses a semaphore-based color system to indicate the states of the main operations of the podips system. Figure 2 shows an example of the podips-monitor homepage, which displays four frames: Kubernetes read, queue write, queue read, and Fluentd write. If podips receives a 500 status message it displays in red, showing that there was a failure. Messages are stored until another is sent, and podips-monitor always displays the stored message. If the difference between the message recording time and the current time is greater than the minimum accepted interval time, the message is displayed in yellow. Otherwise it is displayed in green.

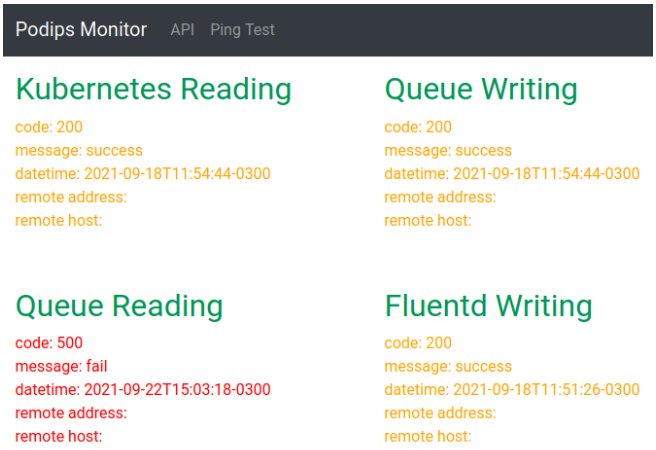

Fig. 2. Homepage of podips-monitor

Queue reading failure, as seen in Figure 2, can be a momentary problem, regularized in the next interaction with the queue. What cannot happen is the persistence of the failure. It is expected, in normal execution, that the red state will eventually occur, but that it will immediately be replaced by the green state.

VI. The Minimum Interval Guarantor

In order to be able to configure a minimum message sending interval for the podips-monitor, it is necessary to ensure that there is IP change in at least one pod in regular periods of time. That's why the system maintains a program in Node.JS that is invoked by a cronjob every 5 minutes. This program only makes an HTTP request to podips-monitor, to an endpoint that returns a JSON object with a summary of the status readings. I mean, it pretends to be a user accessing podips-monitor. Execution is quite fast, so the pod is destroyed as soon as the HTTP response is received. A new pod is then created every 5 minutes, which allows you to determine that the minimum tolerated interval for sending events by Kubernetes is 5 minutes.

VII. Mocking for Testing the Queue

To deploy podips, you need at least one Kubernetes cluster as the main data source and one Flutend server as the destination data source. Additionally, you need a relational database to ensure that messages are not lost if they cannot be immediately written to the queue. In the environment where podips is deployed PostgreSQL is used, but you can use any SQL database.

However, you can simulate the podips process using double components. In fact, you can create an environment that simulates the production and reading of messages, to understand what travels along the podips. The only real component you need is the queue system. We use ActiveMQ, but you can actually use another system that uses the Stomp protocol [2].

If you have an installed queuing system that supports Stomp, such as ActiveMQ, you can use the **mock-producer** and **mock-consumer** programs to simulate, respectively, the production of events by Kubernetes and the sending of messages to Fluentd.

The double **mock-producer** is a program Go licensed under LGPL-2.1 and is available at https://github.com/fgsl/mock-producer. The double **mock-consumer** is a program Python composed by one only source code file, licensed under LGPL-2.1 and available at https://github.com/fgsl/mock-consumer. These two programs, together with the queue, allow you to simulate how podip works on a Linux workstation. For the simulation, all can be run directly, without the need to be in containers.

VIII. The Production Environment

The production environment is composed of two Kubernetes clusters, one containing 68 distributed systems and the other containing 535 distributed systems. Not all have microservices architecture, but they are modularized into at least three pods. Each cluster has an instance of podips, which collects an average of 1000 messages per hour. All messages are aggregated by Fluentd into a single Elasticsearch database, whose content is viewed by support technicians via Graylog.

IX. The Possibilities of Podips

Although it was built to collect events that occurred with Kubernetes pods, the podips architecture can be repurposed for systems that need to collect logs in situations where there are multiple points of failure and where there must be a guarantee that no log messages are lost. As it is a free and

open source system, the reading and writing components can be rewritten to adapt to similar situations.

X. Future works

Although the components of podips are open source, for it to be easily reused it is desirable to have an installer, which deploys all components in a production environment in an automated way. Individual deployment is convenient in a microservices architecture, where system maintenance should be done so as not to make the entire system unavailable, but in order to see it up and running quickly, a user-friendly installer is really desirable.

There is a question about the podips evolution guidelines. It's a pretty stable system. The last publication was running for seven months, until the conclusion of this article, without the need for maintenance. The question is whether podips could not be integrated with one of the external systems it is related to, Kubernetes and Fluentd. But there is currently no idea how this integration could be implemented.

XI. Conclusion

We can observe that the system is composed of heterogeneous microservices in relation to the programming languages used. The system has programs written in Go, Java, Javascript, PHP and Python. It is an example that it is possible to build a distributed system based on microservices using different programming languages. This allows exploring the best of each language and not compromising the architecture in the face of specific language problems.

References

[1] PSR-15: HTTP Server Request Handlers. https://www.php-fig.org/psr/psr-15/

[2] Stomp: The Simple Text Oriented Messaging Protocol. https://stomp.github.io/

.