Multi-Cloud Application Monitoring

Pier Francesco Contino
Università Degli Studi di Roma Tor Vergata
Roma, Italia
pfcontino@gmail.com

Alex Ponzo
Università Degli Studi di Roma Tor Vergata
Roma, Italia
ponzo93@gmail.com

ABSTRACT

The use of multiple cloud computing services makes the monitoring task more difficult, because of the multiple vendor-specific monitoring tools.
Although it's complexity multi-cloud still offers many quality, so it's essential to provide a monitoring system that fits well in this environment. Our project provides a distributed, fault tolerant aggregation tool for multi-cloud monitoring, that measures infrastructure level metrics as well as application level metrics.

KEYWORDS

Cloud computing, Monitoring, Alerting, Multi-cloud, Distributed, Fault tolerance, Amazon Web Service, Google Cloud Platform

1 INTRODUCTION

We use the term multi-cloud when an application uses cloud services provided by different cloud vendors, thus having a heterogeneous architecture.
When you use a multi-cloud infrastructure for your application you find yourself with multiple different monitoring systems, each one with his vendor-specific metrics [5]. To unify this different systems you have to deploy multiple monitor agents to gather the metrics, transfer them in an aggregation component and then retrieve them by using a graphic interface or an alert system.
Many times can be necessary to measure application level metrics to add them to the monitoring system, or you may want metrics that are not directly provided by the cloud provider monitor system; so you can find yourself constrained to add to the agent an external monitoring tool.
We have implemented a monitoring system that does the previously described things and that can recover from a fault on a node.
For the testing of the monitoring system we have also implemented a multi-cloud word count distributed application based on the map reduce paradigm that uses the remote procedure call for the inter-process communication.

2 SYSTEM ARCHITECTURE

The system architecture of the monitoring system is mainly decentralized, except for the aggregator component that is a single node connected with all agent nodes.

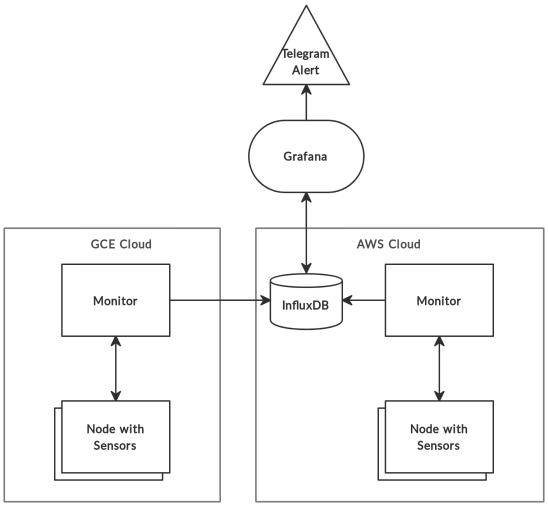

Figure 1: Monitoring Architecture

Agent

The heart of the system architecture is the agent, the element in charge of requesting the metrics from the instances monitored as well as the application level metrics resident on the agent node.
The granularity of the monitoring can be modulated by the configuration file, so that the user can choose the right trade-off from resource overhead and monitoring accuracy.
The agent monitoring tools in our case are:

- AWS CloudWatch
- GCE StackDriver
- Prometheus
- Node exporter

The application level metrics are read from a log file.

Aggregator

This is the component that gather the metrics from the various agents and makes it persistent, always available and uniform them in a single point of access for the user graphic

interface module.
We implemented this as a InfluxDB database installed on a single node, and we used a go library to post data on it from the agents.
We also used it to query the last timestamp of the saved metrics in order to recover the agent state after a crash.
This centralized aggregator can block temporarily the monitoring system in case of a fault on its hosting node, but the node recovery limits this time frame and the metrics on the non-faulty agents are preserved.

Metric Visualizer

In a monitoring system is important to make the metrics representation as clear as possible, so we used a graphic interface for data visualization on the client node.
In particular we make use of Grafana for this task, due to its easiness of configuration and because it provides several useful functions. Here the system finally retrieves data organized in timeseries from the DB, and expose it to the user in many different forms.

Alarm System

Thanks to Grafana it is possible to check if some values of the metrics are out of the norm or out of prefixed ranges (for example it could be useful to know if the CPU utilization of an instance is over 80/90% for over than 5 minutes) and inform the user, so that he can make decisions depending on the situation. We have exploited the Telegram App, using a Bot to update the user on the system state. In case of alert, the system sends a message to a predefined chat, including the cause of the alert and an image of the graph with the actual values; the system keeps then monitoring that value and inform the user in case it returns to normality.

3 VENDOR MONITORING SERVICES

AWS CloudWatch

This is the service that the AWS cloud platform offers for the monitoring, it is a complete monitoring system but we used it only to request the metrics, as suggested by the multicloud nature of the project.
The distributed nature of this service allows us to monitor every node of the system without deploying a monitor on each instance, since we used EC2 instances in our project we collected metrics relative to those.
We used this service in the free tier mode so we could not measures all metrics and the measuring interval was constrained (the minimum measurement interval was five minutes) and the data is not immediately available for the request[1].
To collect the metrics we used the AWS Go SDK in our agent application.

GCE StackDriver

This is the monitoring service offered by Google to gather metrics of the Compute Engine instances; by default those instances already scrape some metrics (like CPU utilization), but an installation of the StackDriver Agent is necessary to get many other ones (like memory used) [3]. The combination of these two mechanisms allows to obtain a large scale of host level metrics on the Google Cloud VMs. As for AWS CloudWatch this service represents both a monitor and a sensor, allowing to retrieve metrics on all other instances of the same network, this can be extremely useful if you program to switch the monitoring and monitored instances in the future. Similarly to the Amazon corresponding service, to collect the metrics in our agent application we used the GCE Go SDK.

4 PROMETHEUS

This is an open-source standalone time-series based monitoring system; its most important components are: a main server which scrapes and stores time series data and special-purpose exporters for services which parse metrics in a format readable by the server. In our project we used Node Exporter, which allows to scrape some host level metrics (like number of OS context switches), exposing them to the Prometheus server.
We decided to install Prometheus on every instance too, to preserve the idea of keeping the possibility to switch monitor and monitored ones in the future.

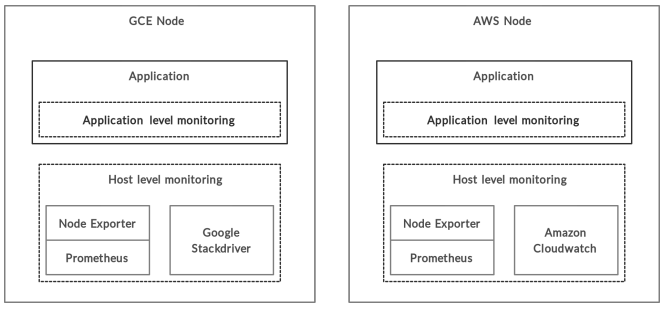

Figure 2: Monitors architecture

5 MONITORED METRICS

Our system is configurable to scrape nearly every metric of the used monitoring tools: StackDriver (mostly for GCE), CloudWatch (for AWS), Prometheus (equipped with Node Exporter), and other compute services metrics (like the Google Compute ones); at the moment we have selected only few metrics from each service to scrape only some of the more significant according to our scope (CPU usage, memory used, network traffic, etc.).

The list of monitored metrics is available in the json files dedicated to the respective monitoring tools:

- metrics_gce.json
- metrics_ec2.json
- prometheus_metrics.json

this files are all located in the "configure" folder and this list of metrics can be expanded at will to scrape every wanted metric available by simply modifying them.
In addition to these we have equipped our application in order to scrape some metrics at this level too; at this scope the monitored system must necessarily be aware of the operation, so if you want to add some application level metrics it is unavoidable to add some features to its code (if you prefer it is possible to create an ad-hoc Prometheus exporter for your application metrics too).
The obtained metrics will be stored by the agent on the DB every 5 minutes, so you may need to wait a little before you can see them; once they are available you can easily add a panel on the Grafana dashboard integrating the needed metrics or setting some alerts.

Word Count Specific metrics

In the word count application we measured various metrics both at the global level and at the node level, also we measured the metrics relative to the specific phase of the word count.
So we have the metric structured as in the table 1.

Metrics	Scope	Phase
Workers	Total	Total
Throughput	Total	Total
Throughput	Worker	Map / Reduce
Word Elaborated	Total	Total
Word Elaborated	Worker	Map / Reduce
Elaboration Time	Worker	Map / Reduce
Elaboration Time	Worker	Reduce

Table 1: Word count metrics

6 FAULT RESISTANT NODE

To make a monitoring agent resistant to a fault we simply discover a faulty node and we restore it as soon as possible.

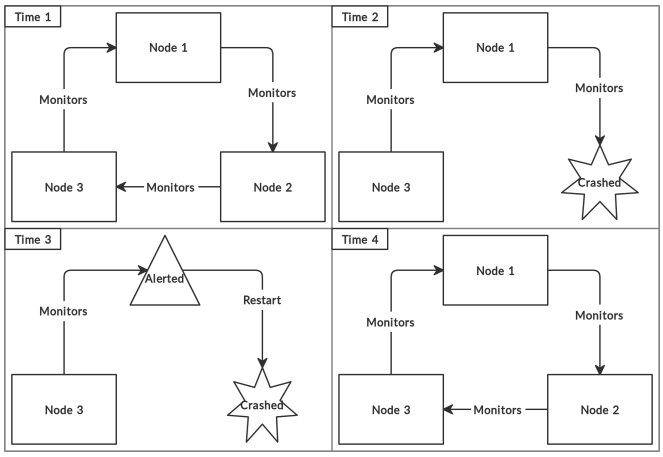

Figure 3: Zookeeper ring model

Noticing faulty nodes

For this purpose we used the group membership concept, so that we notice a failure when a member leaves the group; the implementation is based on ZOOKEEPER ephemeral nodes [2].
The *ephemeral nodes* exists as long as the session that created the node is active, when the session ends the node is deleted automatically by zookeeper.
Zookeeper offers also a *watcher*: a system that gives a signal when a node changes, we utilized it to observe changes in the membership group. To reduce the number of signals received by a node we used an observation scheme based on a ring, where each node observes the following one (figure 3). With this scheme the failure of a node prompt only one request to restore the faulty node avoiding concurrency problems and thus not requiring a lock on the ephemeral nodes.
The zookeeper architecture needs a *minimum of 3 servers* (deployed on independent instances) to ensure correct functioning in the replicated mode and need an odd number of server to reach the quorum. Having 2n+1 servers we can support the failure of up to n servers, so in our base configuration example of 3 servers we can tolerate at most 1 server fault.
The actual implementation consists of two membership group one for the google cloud instances and one for the amazon web service instances. [1]

Restoring faulty nodes

To revive a dead agent we issue a request to the service that hosts it to restart the given instance. The consequence of this operation is the lost of the state of every application resident

[1]This is due to the dynamic nature of the public ip addresses that makes difficult the monitor configuration

on that instance, so we recover the state of the monitoring agent by issuing a request to the database.
The only state of our agent is the last time of the request of the metrics, so we simply request the last timestamp resident on the aggregator database and then request to the monitoring services all the metrics not yet retrieved.
Since AWS EC2 and GCE instances can have transient state (like *stopping* and *pending*) before restarting and instance we first check if the instance is in that state to prevent unnecessary additional restart or even errors in the API call.
To launch the monitoring agent automatically when the instance start (or restart) we attached a startup script to its metadata.[2]

7 IMPLEMENTATION DETAILS

The monitor and the test application were written in the *Go language* utilizing in the word count application the synchronization mechanism offered by it and its rpc native support. It's json parsing functionalities were useful for the configuration of the whole system. The monitors and the application use several cloud services (table 2) offered by different cloud providers thus requiring different configuration and starting scripts, but using the same executable thanks to interface mechanism.

Provider	Service	Use
AWS	EC2	App and Monitoring
AWS	CloudWatch	Monitoring
AWS	S3	Application
GCE	Compute engine	App and Monitoring
GCE	StackDriver	Monitoring

Table 2: Service used

The application and the monitoring system share only some utilities packages and the application metrics package that defines the methods to log and to read application metrics from the log.
We also used some libraries to interface to the various service/framework (table 3) used by the monitoring system.

Description	Repository
Zookeper Client	github.com/samuel/go-zookeeper/zk
InfluxDb Client	github.com/influxdata/influxdb1-client/v2
AWS SDK Library	github.com/aws/aws-sdk-go
GC Api Library	cloud.google.com/go/monitoring/apiv3

Table 3: Libraries used

[2]See add_startup.sh script

Used software platform

The EC2 instances used run on the Amazon Linux AMI and the Compute engine instance were based on Ubuntu 18.04 LTS, the scripts used for the deploying use bash, the ssh and the scp protocols.

How to install the monitor

On the local machine to run the script you should install git, jq, konsole,[3] awscli,[4] gcloud.[5]
The scripts are divided in two folders: script_aws and script_gce. The configuration files are saved in the directory /configuration. Can be useful to add a static ip to the aggregator node instance to simplify the grafana configuration.
The first node of the aws node is the master of the word count application. AWS and GCE *must* have at least 3 zookeeper servers each.

(1) create the instance on aws manually or by running the scripts create_ec2.sh
(2) create the instance gce manually or by running the scripts create_wm.sh
(3) set name to the instances created
(4) set aws IAM roles for the EC2 instances to enable
  - ec2fullaccess
  - s3fullaccess
  - cloudwatchfullaccess
(5) Set a security group (AWS) and set the firewall rules (GCE) with the following inbound open ports:
  - 2888,3888,2181 (zookeeper)
  - 22 (ssh)
  - 1050-1060 (rpc)
  - 9090, 9100 (prometheus)
(6) set names of the instances in monitor.json
(7) set the name of the GCE project in gce_project_id.json
(8) run /aws_script/dependency.sh to install the project dependency and the project itself
(9) run /gce_script/setup_environment.sh to install the project dependency and project itself
(10) to configure the metrics monitored modify
  - metrics_ec2.json
  - metrics_gce.json
  - metrics_prometheus.json
(11) to configure monitoring run configure_monitoring.sh
(12) to add monitoring at startup run add_startup.sh
(13) install grafana on the client [6]
(14) Set the grafana datasource on influxdb on http://ADDR:8086

. After configuration you should restart all instances to activate monitoring or run the start_monitoring.sh script to

[3]https://konsole.kde.org/
[4]https://aws.amazon.com/cli/
[5]https://cloud.google.com/sdk/gcloud/
[6]https://grafana.com/

start it manually.

8 TEST APPLICATION

To test the described above system we have developed a multicloud application, based on the Map-Reduce paradigm [4], that realizes a **distributed word count**; in this segment we are going to describe the used architecture.
Our application exploits a master-worker model to organize and split out the load, and a client-server one to serve the user; it also takes advantage of the Amazon S3 service to store the files and the results.

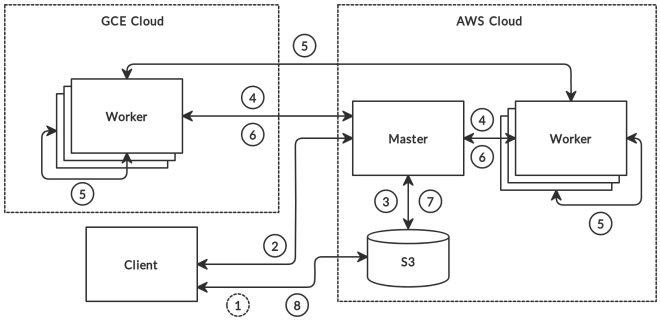

Figure 4: Application architecture

The word count service takes place in the following way:

(1) Client stores the files on the S3 storage.
(2) Client contacts Master asking for a list of files to be processed.
(3) Master retrieves indicated files from the storage service.
(4) Master distributes files to Workers using a Round-Robin algorithm and starts the *Map phase*.
(5) Workers process the received files, counting words inside them and executing a *Pre-Reduce phase*, informing Master upon termination; once the Master has received notification from all of the Workers it instruct them to start the *Shuffle phase*. Nodes exchange data on a predetermined way.
(6) Workers perform the *Reduce phase* and send elaborated data back to the Master.
(7) Master aggregates results, store them on the S3 service and informs the Client.
(8) Client retrieves the results from the remote storage and finally shows them to the user.

Implementation details

In our implementation only the master knows in advance the configuration of the nodes, so it's his job to inform the workers of the coordinates of their siblings. Before doing that it check if the nodes are alive by issuing a request on all workers, if the request time out he removes them from the list of the workers used in the imminent request.
The master does not keep track of the nodes state during the following operations, so the application is not resistant to a crash of a worker during this time frame.
The inter-process communication it's based on Go rpc providing us a synchronization decoupling layer for not blocking the nodes waiting for the results of the operations. Although this a barrier is needed to inform the workers of the end of the map phase as they can't start the reduce phase until they have all the results of the previous one.

Application deploy

In order to make the application work some setup is needed, in first place you need to manipulate some files to give the needed information to the system:

- create a file named "bucket.json" in "./configuration/generated", containing the name of your S3 bucket
- adapt file "word_count.json" in "./configuration" to your system nodes
- optional: in case of local testing adapt file "app_node.json" in "./configuration/generated"

In second place it is enough to start some script that will take care of automate the setup of application for you:

(1) ./script_gce/configure_app.sh to configure the gce part
(2) ./script_aws/configure_app.sh to configure the aws part

N.B.: the previous two script MUST be launched in that order; for the next two the order is invariant:

- ./script_gce/start_app.sh to start gce workers
- ./script_aws/start_app.sh to start master and aws workers

Now the Server part is ready, time to use the Client.
The Client of the application uses different flags to specify different operations and arguments for the steps (1) and (2) (in step (8) results are retrieved automatically).
These are the commands for operations:

- *load*: this command loads the files specified in the AWS S3 bucket at specified names
- *delete*: this command deletes the files specified by names from the AWS S3 bucket
- *list*: this command lists the files in the bucket
- *count*: this command executes the wordcount of the files in the bucket identified by given names

These are the commands for arguments specification:

- *names*: (used with load/delete/count) specifies the names for the S3 files in the bucket to use/load/delete

- *paths*: (used with load) specifies the paths for the local files to upload
- *serverAddr*: (used with count) specifies the address of the server for the rpc request

9 TESTING

To perform an adequate testing on our system we had to test separately different aspects:

- Fault tolerance: stopping a VM instance (both on GCE and AWS) causes it to restart immediately and the monitor agent to start scraping metrics again
- Alert: setting some thresholds adequately we have received a notification on Telegram thanks to the Bot when the query value has exceeded it (including the graph responsible of the alert); we have then received another notification when the value is returned under the threshold (we have tested pending alerts too)
- Load "balance" of the application: due to the nature of the round robin algorithm used by master to distribute the files to the workers we have experienced a fair balancing of the load in normal conditions; some exceptions being when number of files is minor then number of workers and when files have a significant variance in the number of words (in this cases some workers metrics like CPU utilization have grown rapidly)
- Consistency of the application: Metrics measured at workers level measured match the total metrics measured by the master.

10 LIMITATIONS

During the development of the project we have experienced some minor limitations that prevented us to achieve some of our prefixed goals or simply slowed us down, for example:

- S3 metrics are gathered only once a day (near midnight), so it results quite useless trying to monitor it
- CloudWatch in free tier mode allows us to scrape metrics with an interval of at least 5 minutes, so monitoring may result a bit delayed
- At first we were using very small VM on GCE and we had to switch to a slightly bigger one when memory size becomes too small to support the monitoring system
- Some unexpected problems with Prometheus configuration prevented it from work on Amazon in the same way it does on Google, so for this cloud we managed to modify the yaml configuration file via script instead of linking a file json in section "file_sd_configs".

REFERENCES

[1] Amazon. 2019. Amazon CloudWatch Pricing. (2019). Retrieved December 16, 2019 from https://aws.amazon.com/cloudwatch/pricing/

[2] Apache. 2019. Zookeeper overview. (2019). Retrieved December 16, 2019 from https://zookeeper.apache.org/doc/r3.5.6/zookeeperOver.html

[3] Google. 2019. Monitoring agent overview. (2019). Retrieved December 16, 2019 from https://cloud.google.com/monitoring/agent/

[4] IBM. 2019. What is MapReduce? (2019). Retrieved December 16, 2019 from https://www.ibm.com/analytics/hadoop/mapreduce

[5] NetworkComputing. 2019. Top 5 Challenges of Monitoring Multi-cloud Environments. (2019). Retrieved December 16, 2019 from https://www.networkcomputing.com/cloud-infrastructure/top-5-challenges-monitoring-multi-cloud-environments/