Cluster-level Logging *of* Containers *with* Containers

SATNAM SINGH

LOGGING CHALLENGES OF CONTAINER BASED CLOUD DEPLOYMENTS

This article shows how cluster-level logging infrastructure can be implemented using open source tools and deployed using the very same abstractions that are used to compose and manage the software systems being logged. Collecting and analyzing log information is an essential aspect of running production systems to ensure their reliability and to provide important auditing information. Many tools have been developed to help with the aggregation and collection of logs for specific software components (e.g., an Apache web server) running on specific servers (e.g., Fluentd[4] and Logstash.[9]) They are accompanied by tools such as Elasticsearch[3] for ingesting log information into persistent storage and tools such as Kibana[7] for querying log information.

Collecting the logs of components realized using containers such as those from Docker[2] and orchestrated by systems such as Kubernetes,[8] however, is more challenging because there is no longer a specific program and a specific sever. This is because a component consists of many anonymous instances (replicas) that are scaled up and down in number depending on the system load.

Furthermore, there is no specific server because each replica is run on a server chosen by the orchestrator.

This article looks at how to overcome these challenges by describing how cluster-level log aggregation and inspection can be implemented on the Kubernetes orchestration framework. A key aspect of the approach described here is its exploitation of the same abstractions that are used to compose and manage the system to be logged to also build the logging infrastructure itself. This approach makes use of using existing open source tools such as Fluentd, Elasticsearch, and Kibana, which are deployed inside containers and orchestrated to collect the logs of the other containers running in a cluster.

A BRIEF INTRODUCTION TO KUBERNETES

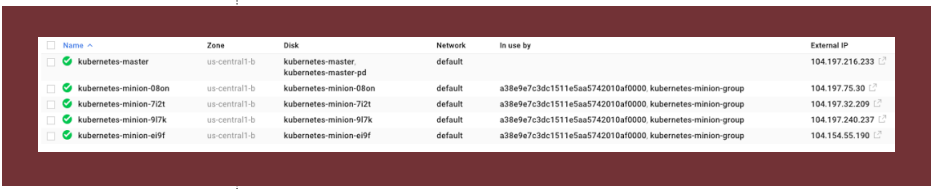
This article describes just enough of the Kubernetes system to help motivate a log collection and aggregation scenario for a simple application. A comprehensive description of the Kubernetes container orchestrator can be found on its website,[8] and an overview article on Borg, Omega and Kubernetes is available on *acmqueue*.[1]

The Kubernetes system can orchestrate components of applications on a variety of public clouds as well as private clusters. In this article an application is deployed on a Kubernetes cluster created on a collection of VMs (virtual machines) running on the public cloud Google Compute Engine.[5] A cluster could have been created using Google Container Engine (GKE),[6] which automates many aspects of cluster creation and management. To emphasize the provider-agnostic nature of the approach, we illustrate the explicit creation of a Kubernetes cluster that performs

log collection and aggregation with open-source tools. Either explicit cluster creation or creation using a cluster management system like GKE allows us to perform log collection and aggregation with open-source tools, although GKE allows for tighter integration with Google's proprietary cloud logging system.

Figure 1 shows the deployment of a four-node Kubernetes cluster that is used for the example application described in this article. This cluster has four worker VMs called `kubernetes-minion-08on`, `kubernetes-minion-7i2t`, `kubernetes-minion-9l7k`, and `kubernetes-minion-ei9f`. A fifth Kubernetes master VM orchestrates work onto the other VMs. For work scheduled on this cluster by Kubernetes, however, you should remain oblivious to the name or IP address of the particular node that is used to run the applications since this is one of the details that is abstracted by Kubernetes. *You don't know the name of the machine running our program.* Furthermore, the components of the application will scale up and down in size as the system evolves and deals with failure, so one logical component may execute across many different machines. *The name of the machine(s) running your program may change.*

FIGURE 1: **KUBERNETES CLUSTER RUNNING ON GOOGLE COMPUTE PLATFORM**



Name	Zone	Disk	Network	In use by	External IP
kubernetes-master	us-central1-b	kubernetes-master, kubernetes-master-pd	default		104.197.216.233
kubernetes-minion-08on	us-central1-b	kubernetes-minion-08on	default	a38e9e7c3dc1511e5aa5742010af0000, kubernetes-minion-group	104.197.75.30
kubernetes-minion-7i2t	us-central1-b	kubernetes-minion-7i2t	default	a38e9e7c3dc1511e5aa5742010af0000, kubernetes-minion-group	104.197.32.209
kubernetes-minion-9l7k	us-central1-b	kubernetes-minion-9l7k	default	a38e9e7c3dc1511e5aa5742010af0000, kubernetes-minion-group	104.197.240.237
kubernetes-minion-ei9f	us-central1-b	kubernetes-minion-ei9f	default	a38e9e7c3dc1511e5aa5742010af0000, kubernetes-minion-group	104.154.55.190

Consequently, in the Kubernetes model it does not make sense to think of a specific program P running on a specific machine M. It is far more idiomatic to identify parts of the system by making queries over labels that are attached to anonymous entities created by the Kubernetes orchestrator, which will return the currently running entities that match the query. This allows us to talk about a dynamically evolving infrastructure without mentioning the names of specific resources.

A Music Store Application

A Kubernetes deployment of a hypothetical music store application is used to help describe how cluster-level container logs can be collected. The application has several front-end microservices that accept HTTP requests to a web interface for browsing and buying music. These front-end services work by communicating with a back-end MySQL instance and a Redis cluster that provide the persistent storage needed by the application. A persistent disk hosted on Google Compute Engine also provides the storage needed by the MySQL database.

LOGGING PODS

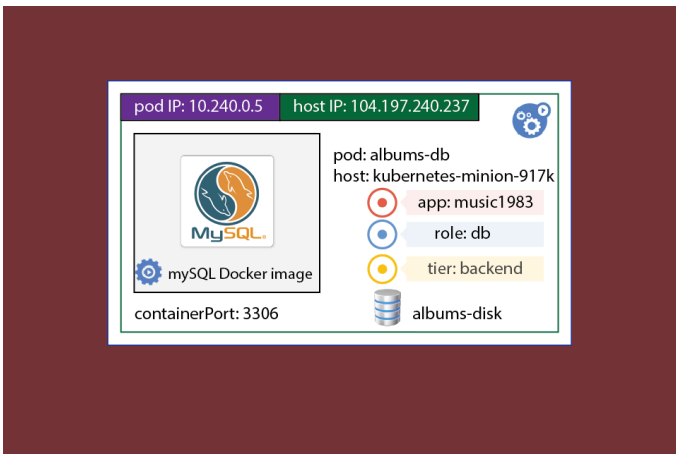
The basic unit of deployment in Kubernetes is a pod. A pod is the specification of resources that should always be allocated together as an atomic unit onto the same node along with other information that a cluster orchestrator can use to manage the pod's behavior. The music store application uses one pod to describe the deployment of the MySQL instance as shown in the YAML file (albums-db-pod.

yaml [https://github.com/satnam6502/logging-acm-queue/blob/master/albums-db-pod.yaml]):

```
apiVersion: v1
kind: Pod
metadata:
  name: albums-db
  labels:
    app: music1983
    role: db
    tier: backend
spec:
  containers:
  - name: mysql
    image: mysql:5.6
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: REDACTED
    ports:
    - containerPort: 3306
    volumeMounts:
    - name: mysql-persistent-storage
      mountPath: /var/lib/mysql
  volumes:
  - name: mysql-persistent-storage
    gcePersistentDisk:
      pdName: albums-disk
      fsType: ext4
```

This specification can be used to create a deployment of the music store database:

FIGURE 2: **A DEPLOYMENT OF THE MYSQL ALBUMS DATABASE**




```
$ kubectl create -f albums-db-pod.yaml
```

Figure 2 illustrates a deployment of this pod, which has the name `albums-db` and a pod IP address of 10.240.0.5. It runs on the Google Compute Engine VM called `kubernetes-minion-917k`, contains a Docker image of a MySQL instance, and uses a persistent disk on Google Compute Engine called `albums-disk`. Three labels identify the application `music1983`, the role `db`, and the tier `backend`. The pod exposes the port 3306 serviced by the MySQL Docker instance for use by other components in the same cluster through the address 10.240.0.5:3306.

Inside the Kubernetes cluster, you can connect to this database, populate it, and make queries. For example:

```
$ mysql --host=10.240.0.5 --user=NAME --password=REDACTED albums
mysql> select * from pop where artist = 'Pink Floyd';
+------------+----------------------+-----------+----------+
| artist     | album                | inventory | released |
+------------+----------------------+-----------+----------+
| Pink Floyd | Dark Side of the Moon |        57 |     1973 |
| Pink Floyd | The Wall              |       103 |     1983 |
+------------+----------------------+-----------+----------+
2 rows in set (0.08 sec)
```

The logs for a pod can be extracted using the Kubernetes command-line tool:

```
$ kubectl logs albums-db
…
2016-03-01 00:43:20 1 [Note] InnoDB: 5.6.29 started; log sequence
number 1710893
2016-03-01 00:43:20 1 [Note] Server hostname (bind-address): '*';
port: 3306
2016-03-01 00:43:20 1 [Note] IPv6 is available.
…
```

This command fetches the logs for the currently running MySQL Docker image. You can ask Kubernetes to report the Docker container ID for the running MySQL instance:

```
$ kubectl describe pod albums-db | grep "Container ID"
    Container ID:   docker://38ab5c9e9aa8004e9b61f19885…
```

Does this solve the problem of collecting logs from an application deployed on a Kubernetes cluster? One problem is that during the lifetime of a pod the underlying Docker container (or containers) that is deployed may

terminate and new replacement containers created (e.g., to deal with a container that has failed in some way). The following induces a failure by sabotaging the MySQL container and seeing how Kubernetes responds. This is done by SSH'ing to the Google Compute Engine VM that is running the container in order to kill the MySQL Docker container.

```
$ gcloud compute --project "kubernetes-6502"
ssh --zone "us-central1-b" "kubernetes-minion-
9l7k"
$ sudo -s
# docker ps
CONTAINER ID        IMAGE
38ab5c9e9aa8        mysql:5.6
# docker kill 38ab5c9e9aa8
38ab5c9e9aa8
# docker ps
CONTAINER ID
abdfca342daa        mysql:5.6
```

Soon after, an agent running on the node that is part of the Kubernetes system noticed the container was no longer running. In order to drive the current state of the system to the desired state, a new Docker instance of MySQL is created with a container ID that starts with abdfca342daa. Checking the logs of `albums-db` now reveals:

```
$ kubectl logs albums-db
2016-03-01 01:33:25 0 [Note] mysqld (mysqld 5.6.29) starting as
process 1 ...
…
2016-03-01 01:33:25 1 [Note] InnoDB: The log sequence numbers
1710893 and 1710893 in ibdata files do not match the log se-
quence number 1710903 in the ib_logfiles!
2016-03-01 01:33:25 1 [Note] InnoDB: Database was not shutdown
normally!
2016-03-01 01:33:25 1 [Note] InnoDB: Starting crash recovery.
…
2016-03-01 01:33:25 1 [Note] InnoDB: 5.6.29 started; log sequence
number 1710903
2016-03-01 01:33:25 1 [Note] Server hostname (bind-address): '*';
port: 3306
```

The logs are now for the currently running container (abdfca342daa), and the logs for the previous instance of the MySQL container (38ab5c9e9aa8) have been lost. The lifetime of these logs is determined from the lifetime of the underlying Docker container rather than the lifetime of the pod. What is really needed is a mechanism for collecting and storing all the log information that was generated by every container instance that runs as part of this pod's execution lifecycle.

Logging Pods Managed by Replication Controllers

Although a single pod in a Kubernetes cluster can be specified and deployed, it is far more idiomatic to specify a replication controller that creates many replicas of a pod. Here is an example of a replication controller that

specifies the deployment of two Redis slave pods (redis-slave-controller.yaml [https://github.com/satnam6502/logging-acm-queue/blob/master/redis-slave-controller.yaml]):

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: redis-slave
  labels:
    app: music1983
    role: slave
    tier: backend
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: music1983
        role: slave
        tier: backend
    spec:
      containers:
      - name: slave
        image: redis
        resources:
          requests:
            cpu: 300m
            memory: 250Mi
        ports:
        - containerPort: 6379
```

This specification declares a replication controller called `redis-slave`, which has three user-defined labels of metadata that are attached to each pod it creates. The labels identify the name of the overall application `music1983`, the role of the Redis instance `slave`, and this pod as being a member of the `backend` tier. The initial number of replica pods is set to two, although this number may be dialed up or down later. Each pod to be replicated consists of a Redis Docker container, an exposed port 6379 over which the Redis protocol operates, and some resource requests for CPU and memory utilization that are communicated to the scheduler. This specification can be given to the Kubernetes command-line tool to bring the Redis slave pods to life:

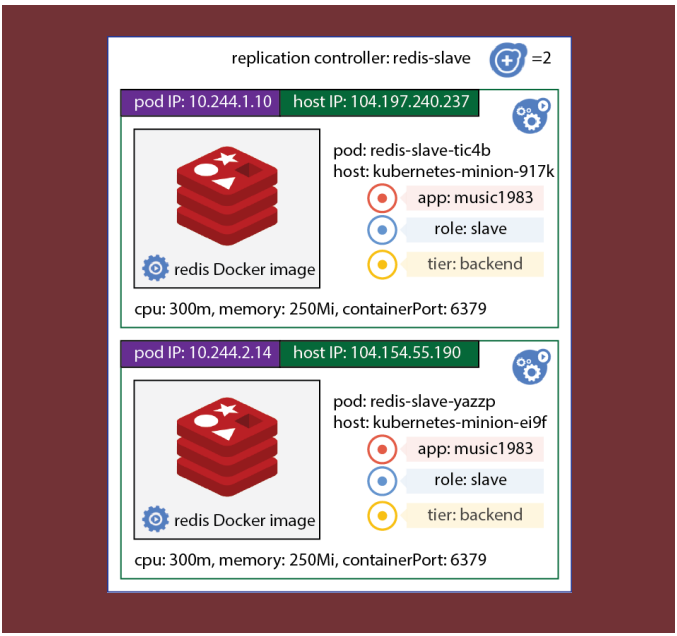
```
$ kubectl create -f redis-slave-controller.yaml
```

Figure 3 shows a sample deployment of such a Redis slave controller with two pods running on two different Google Compute Engine VMs. The pods have automatically generated names: redis-slave-tic4b and redis-slave-yazzp. Do not get too attached to the name of any specific pod, since pods may come and go as a result of failure or changes in the cardinality of the replication controller.

Logging Pods Captured by a Service Specification

Each pod has its own IP address, and the IP address of the host VM is also shown, although this address is never of any interest to the Kubernetes application running on the cluster. If you can't utter the name of a specific pod, then how can you interact with it? Label selectors can define an

FIGURE 3: **DEPLOYMENT OF THE REDIS SLAVE REPLICATED PODS**



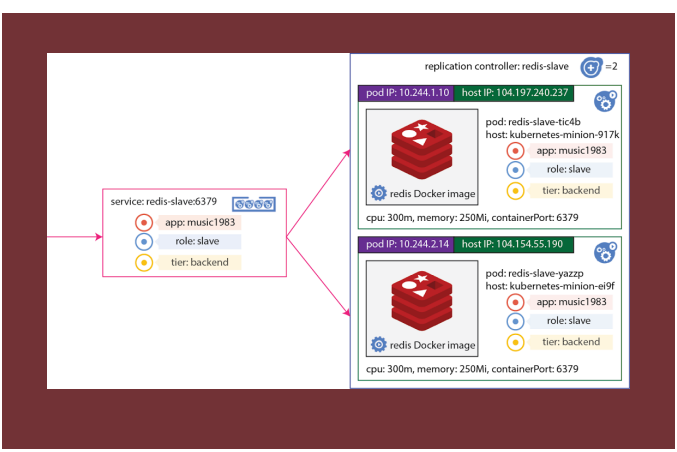
entity called a *service*, which introduces a *stable name* for a collection of resources. Requests sent to the stable name provided by the service are automatically routed to a pod that matches the net cast by the service label selectors. Here is the definition of a service identifying pods that provide the Redis slave functionality (redis-slave-service.yaml [https://github.com/satnam6502/logging-acm-queue/blob/master/redis-slave-service.yaml]):

```
apiVersion: v1
kind: Service
metadata:
```

```
  name: redis-slave
  labels:
    app: music1983
    role: slave
    tier: backend
spec:
  ports:
  - port: 6379
  selector:
    app: music1983
    role: slave
    tier: backend
```

The deployment of this service is illustrated in figure 4. The service defines a DNS (Domain Name System)-resolvable name within the cluster `redis-slave`, which accepts requests on port 6379 and then forwards them to

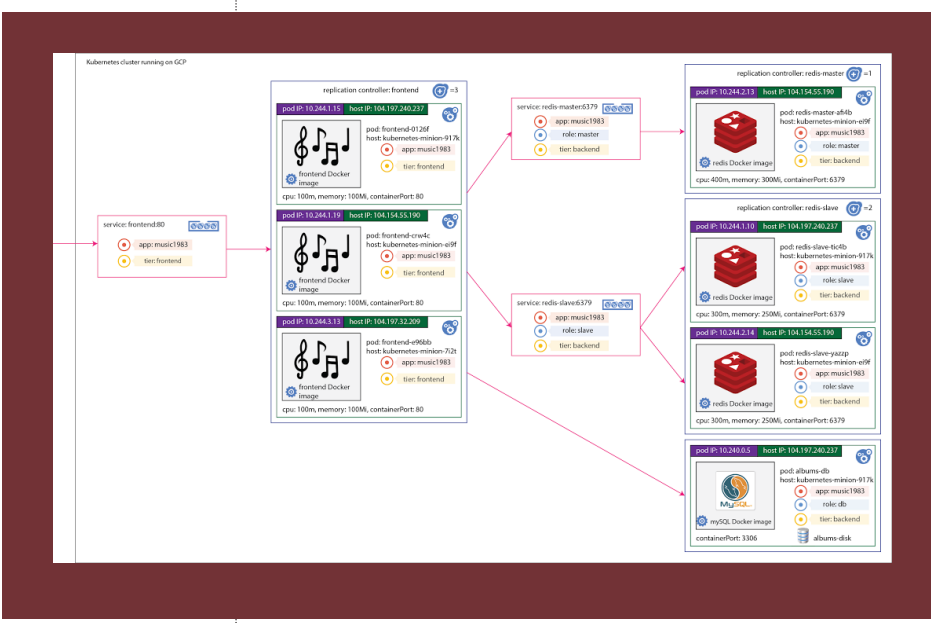
FIGURE 4: **SERVICE MAPPING REQUESTS TO REDIS READ SLAVES**



any pod that matches its label selectors (i.e., any pod that has the app label set to `music1983`, the role label set to `slave`, and the tier label set to `backend`). Now consumers of the `redis-slave`-read-replicated pods are insulated from the names of the specific pods that are used to service their requests as well as the names of the specific nodes on which these pods are running.

Figure 5 shows the deployment of a music store website made up of several front-end microservices that accept external requests and render a web user interface. These front-end services store information in a key/value store implemented by several instances of the Redis key/

FIGURE 5: **A KUBERNETES DEPLOYMENT OF A MUSIC STORE SERVICE**



value store. The system is designed to make it easy to independently scale up the capacity for (a) serving web traffic; (b) reading from the key/value storage system; (c) writing to the key/value storage system. As more users connect to the music store website, the number of front-end microservices can be dialed up. Typically, you expect many more relatively cheap read operations than expensive write operations to the key/value store. To process read operations as quickly as possible, reads from Redis slave instances (two in this case) are serviced and a separate pool of Redis microservices deployed as masters that perform write operations (initially just one in this case).

Collecting the logs of the front-end service pods brings up another life-cycle issue. It is not enough to just collect the logs from each of the three currently running pods (even when collecting the logs of multiple invocations of the front-end Docker image), because pods themselves may be terminated and then reborn (possibly on a different host machine). In certain situations, there may briefly be more than three front-end pods or perhaps fewer than three. If this occurs, the Kubernetes orchestration system will notice and create or kill pods to drive the system to the declared state of having just three front-end pods. As front-end pods come and go, you want to collect all of their logs, so the log-collection activity has a lifetime that is associated with the front-end replication controller rather than the lifetime of a specific pod.

Using Fluentd to collect node-level logs

The open-source log aggregator Fluentd is used to collect

the logs of the Docker containers running on a given node. Trying to run an instance of a Fluentd collector process directly on each node (i.e., GCE VM) generates the same deployment problems that pods were created to solve (e.g., dealing with failure and performing updates). Consequently, node-level log aggregation of Docker containers is actually implemented from a Docker container that runs as part of a pod specification. This meta-approach allows the logging layer to benefit from the same advantages afforded to the application layers by the Kubernetes model for managing deployment and life cycle events. For example, the rolling update mechanism of Kubernetes can update the pods running on each node so they use an updated version of the log-aggregation software while the cluster is still running.

The Fluentd collectors do not store the logs themselves. Instead they send their logs to an Elasticsearch cluster that stores the log information in a replicated set of nodes. Again, rather than running this Elasticsearch cluster directly "on the metal," you can define pods that specify the behavior of a single Elasticsearch replica, then define a replication controller to specify a collection of Elasticsearch nodes that contain the replicated log information and provide a query interface, and finally define a service that provides a stable name for balancing queries to the Elasticsearch cluster.

The complete specification of the Fluentd node-level collector pods is shown here (fluentd-es.yaml [https://github.com/kubernetes/kubernetes/blob/master/cluster/saltbase/salt/fluentd-es/]):

```
apiVersion: v1
kind: Pod
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
spec:
  containers:
  - name: fluentd-elasticsearch
    image: gcr.io/google_containers/fluentd-
elasticsearch:1.11
    resources:
      limits:
        cpu: 100m
    args:
    - -q
    volumeMounts:
    - name: varlog
      mountPath: /var/log
    - name: varlibdockercontainers
      mountPath: /var/lib/docker/containers
      readOnly: true
  terminationGracePeriodSeconds: 30
  volumes:
  - name: varlog
    hostPath:
      path: /var/log
  - name: varlibdockercontainers
    hostPath:
      path: /var/lib/docker/containers
```

This specifies a node-level collector that runs a

specially built Fluentd image configured to send logs to an Elasticsearch cluster using the DNS name and port elasticsearch-logging:9200 (which is itself implemented as a Kubernetes service). The specification also describes how the location of the Docker logs on the node-level file system are mapped to the file system inside the Docker container run by the pod. This allows the logs of all the Docker containers on the node to be collected by this Fluentd instance running inside this container.

When a Kubernetes cluster is configured to use logging with Elasticsearch as the data store, the cluster creation process instantiates a log-collector pod on each node. These pods can be observed in the kube-system namespace:

```
$ kubectl get pods --namespace=kube-system
NAME                                        READY STATUS  RESTARTS AGE
fluentd-elasticsearch-kubernetes-minion-08on 1/1   Running 0        16d
fluentd-elasticsearch-kubernetes-minion-7i2t 1/1   Running 0        16d
fluentd-elasticsearch-kubernetes-minion-917k 1/1   Running 0        16d
fluentd-elasticsearch-kubernetes-minion-ei9f 1/1   Running 0        16d
…
```

A special process on each node makes sure that one of these log-collection pods is running on each node. If a log-collector pod fails for any reason, a new one is created in its place. These pods collect the logs of the locally running Docker containers and ingest them into an Elasticsearch Kubernetes service running in the kube-system namespace.

Using Elasticsearch to Store and Query Cluster Logs

A cluster created using Elasticsearch for the storage of logs will by default instantiate two Elasticsearch instances. The specification for these Elasticsearch logging pods can be found at es-controller.yaml [https://github.com/kubernetes/kubernetes/blob/master/cluster/addons/fluentd-elasticsearch/es-controller.yaml], which describes a replication controller for the Elasicsearch instances as well as the actual configuration of the Elasticsearch logging pods. These can be observed in the kube-system namespace:

```
$ kubectl get pods --namespace=kube-system
NAME                            READY    STATUS     RESTARTS   AGE
elasticsearch-logging-v1-7rmo3  1/1      Running    0          16d
elasticsearch-logging-v1-v7lmv  1/1      Running    0          16d
…
```

The node-level log-collection Fluentd pods do not speak directly to these Elasticsearch pods. Instead, they connect to the DNS name and elasticsearch-logging:9200, which is implemented by an Elasticsearch Kubernetes service es-service.yaml [https://github.com/kubernetes/kubernetes/blob/master/cluster/addons/fluentd-elasticsearch/es-service.yaml]:

```
apiVersion: v1
kind: Service
metadata:
  name: elasticsearch-logging
  namespace: kube-system
```

```
    labels:
      k8s-app: elasticsearch-logging
      kubernetes.io/cluster-service: “true”
      kubernetes.io/name: “Elasticsearch”
spec:
  ports:
  - port: 9200
    protocol: TCP
    targetPort: db
  selector:
    k8s-app: elasticsearch-logging
```

You can observe this service running in the kube-system namespace:

```
$ kubectl get services --namespace=kube-system
NAME                    CLUSTER_IP    EXTERNAL_IP    PORT(S)
elasticsearch-logging   10.0.8.117    <none>         9200/TCP
…
```

Elasticsearch can be queried for the logs of all pods that are captured by the label selectors for the front-end service. A local proxy allows you to connect to the cluster with administrator privileges, which are required to retrieve the logs of running containers. You query for just the logs of containers that are marked with a `container_name` field of `frontend-server`.

```
$ kubectl proxy
<elsewhere>
$ curl -XGET “http://localhost:8001/api/v1/proxy/namespaces/
kube-system/services/elasticsearch-logging/_search?q=container_
name:frontend-server&_source=false&fields=log&pretty=true”
…
    }, {
      “_index” : “logstash-2016.02.26”,
      “_type” : “fluentd”,
      “_id” : “AVMa-COpcuStSsThKOM4”,
      “_score” : 2.8861463,
      “fields” : {
        “log” : [ “Slow read for key k103: 192 ms” ]
      }
…
    }, {
      “_index” : “logstash-2016.02.26”,
      “_type” : “fluentd”,
      “_id” : “AVMa-COpcuStSsThKONE”,
      “_score” : 2.8861463,
      “fields” : {
        “log” : [ “[negroni] Started GET /lrange/k336” ]
      }
…
    }, {
      “_index” : “logstash-2016.02.26”,
      “_type” : “fluentd”,
      “_id” : “AVMa-C_fcuStSsThKOOp”,
      “_score” : 2.8861463,
      “fields” : {
        “log” : [ “Slow write for key k970: 187 ms” ]
      }
    }, {
…
```

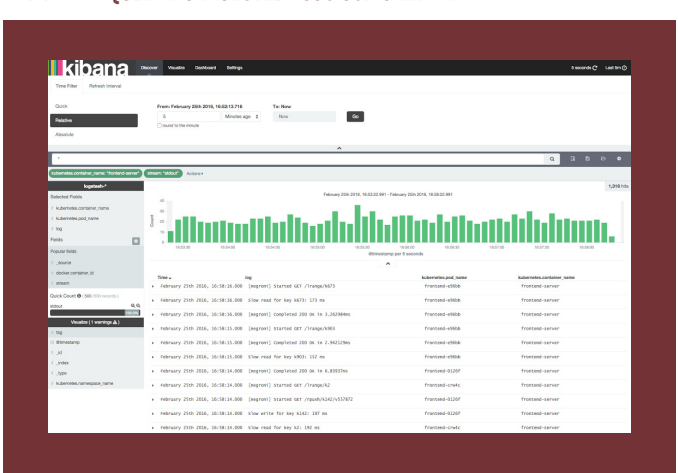
Since the Elasticsearch cluster is a collection of pods managed by a replication controller, it can deal with an increased query load to the logging system by simply increasing the number of replica nodes for the Elasticsearch logging instances. Each pod contains a replica of the ingested logs so if one pod dies for some reason (e.g., the machine it is running on fails), then a new pod will be created to replace it, and it will synchronize with the running pods to replicate the ingested logs.

VIEWING LOGS WITH KIBANA

The aggregated logs in the Elasticsearch cluster can be viewed using Kibana. This presents a web interface, which provides a more convenient interactive method for querying the ingested logs, as illustrated in figure 6.

The Kibana pods are also monitored by the Kubernetes

FIGURE 6: **QUERYING INGESTED LOGS USING KIBANA**



system to ensure they are running healthily and the expected number of replicas are present. The life cycle of these pods is controlled by a replication-controller specification similar in nature to how the Elasticsearch cluster was configured. The following output shows the cluster configured to maintain two Elasticsearch instances and one Kibana instance. If system load increases, a simple command can be issued to dial up the number of Elasticsearch and Kibana replicas. Furthermore, the number of Elasticsearch replicas can be scaled up independently of the number of Kibana instances, allowing you to respond to increases in different kinds of loads by scaling up only the subcomponents needed to meet that demand.

SUMMARY

Collecting the logs of containers running in an orchestrated cluster presents some challenges that are not faced by manually deployed software components. In particular, we cannot explicitly identify by name a particular container (or the name of the pod in which it is contained), nor the node that container is running on, because both of these may change during the lifetime of the deployed application. As application components (microservices) come and go, we need to gather and aggregate all the logs of the containers that work as part of the application during its life cycle. This challenge is addressed by the use of label-selector queries to identify which running activities belong to the application of interest at any given moment. Then these queries can be used (by way of a Kubernetes service) to query the logs of a dynamically evolving application.

The basic infrastructure needed to implement log aggregation and collection can itself be implemented using the same abstractions used to compose and manage the applications which need to be logged: pods, replication controllers, and services. This allows for adapting the capacity of the logging system and updating it while it is running as well as robustly dealing with failure. This also provides a model for developing other cloud computing system infrastructure components in a modular, flexible, reliable, and scalable manner.

References

1. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J. 2016. Borg, Omega, and Kubernetes. *Acmqueue* 14(1); http://queue.acm.org/detail.cfm?id=2898444.
2. Docker; www.docker.com.
3. Elasticsearch. Elastic; https://www.elastic.co/products/elasticsearch.
4. Fluentd; http://www.fluentd.org/.
5. Google Compute Engine; https://cloud.google.com/compute/.
6. Google Container Engine; https://cloud.google.com/container-engine/.
7. Kibana. Elastic; https://www.elastic.co/products/kibana.
8. Kubernetes; http://kubernetes.io/.
9. Logstash. Elastic; https://www.elastic.co/products/logstash.

Satnam Singh *(s.singh@acm.org) is a software engineer at Facebook working on mobile performance. Previously he worked at Google on the Kubernetes project.*