Design and Implementation of Dynamic Expansion of Vehicle Real-Time Data Transmission Architecture Based on Kubernetes

Chen Chen\textsuperscript{a}, Jianjun Cai\textsuperscript{b}, Nver Ren\textsuperscript{c}, Kunhao Li\textsuperscript{d}

Automotive Data Center, China Automotive Technology&Research Center Co.Ltd, Tianjin China
\textsuperscript{a}chenchen@catarc.ac.cn,\textsuperscript{b}caijianjun@catarc.ac.cn, \textsuperscript{c}rennver@catarc.ac.cn, \textsuperscript{d}likunhao@catarc.ac.cn

Abstract—In order to deal with the overload problem of smart car server, a dynamic capacity expansion technology of vehicle real-time data transmission architecture based on Kubernetes cluster is proposed. The technology uses Docker to split the architecture into multiple micro-services, and uses the HPA feature of Kubernetes to periodically monitor server CPU resource utilization, trigger expansion rules, and automate the increase and decrease of nodes to achieve capacity expansion. It has been proved by experiments that the dynamic expansion technology of the real-time data transmission architecture based on Kubernetes can realize the dynamic expansion of the architecture. It solves a series of problems caused by server service overload caused by the sudden increase of data sent by the car in a short period of time, and ensures the stability of the service.

1. INTRODUCTION
A series of technological changes which include the rapid development of network and chip technologies, the iterative upgrading of new energy vehicles, the popularity of 4 g / 5 g, and the increasing capabilities and requirements of various network terminals enables automotive terminals to transmit more data in shorter time, accurate positioning of road vehicles in real time, capturing abnormalities become possible, and also makes smart driving possible. This paper studies an expansive solution which realize a dynamic expansion technology using Kubernetes cluster to deal with a series of problems caused by server service overload aroused by sudden burst of data sent by the car, ensuring the stability of the service and the architecture.

From the bottom layer, Docker is used to split the architecture into multiple micro-services and package them into multiple different images. Kubernetes differentiates the service architecture into multiple service deployments, and then calls from inside Kubernetes to ensure the security of the service. At the same time, it relies on its HPA for real-time expansion, probe and other technologies for real-time monitoring of Service and Pod. Virtualization technology can greatly improve the utilization of server resources, virtualize server resources, and manage them in combination with micro-service solutions, so that services can be matched with containers and service clusters can be quickly replicated and expanded.

2. BASIC CONCEPT
Docker is an open source application container engine for container-level virtualization. For Kubernetes clusters, Docker is a mirrored repository for managing images. Developers need to periodically package
the code into a mirrored deployment. On the Docker private service, it is provided to be called by Kubernetes. Kubernetes will pull the corresponding image in the Docker repository according to the user's configuration file (yaml file) to form Deployment, Service, Pod, and so on. The Service consists of multiple Pods with the same configuration. The Pod is generated by the rules of the Deployment.

To facilitate the management of containerized applications, Kubernetes abstracts some concepts based on the Docker container. These concepts exist as resource objects in Kubernetes. Kubernetes uses resource objects to describe clusters. The expected state, these objects are persistent entities, you can use the tools provided by Kubernetes to add, delete and modify something. Kubernetes is an open source system for auto-deploying, extending and managing containerized applications. This article mainly uses its horizontal extension mechanism, in addition to its service discovery and load balancing mechanism, automatic deployment and rollback mechanism, and self-healing mechanism, key and configuration management mechanisms. The Kubernetes cluster consists of Maser and Node, which is scheduled by the master load. Node is responsible for the implementation of the service, and Pod is the implementation of the Service. A group of cluster management related processes consisting of etcd, API Server, Controller Manager, and Scheduler runs on the Master node, which implements management functions such as resource management, elastic scaling, Pod scheduling, system monitoring, and error correction for the entire cluster. Each component runs Kubelet, Proxy, and Docker daemon, and is responsible for managing the life cycle and service proxy of the Pod on this node.

Kubernetes initiates an RC creation request through Kubectl, and the API Server writes the request to etcd. The Controller Manager gets and analyzes the RC event according to the interface of the API Server's listening resource. The RC generates the Pod, and then the Scheduler performs scheduling and selects the host Node, the result is recorded by etcd. Kubectl submits a new mapping creation request through the Pod's Service. The Controller Manager uses the Label to query the relevant Pod instance to generate Endpoints information. The API Server writes the information to etcd, and then the Proxy process uses the API Server query to listen to the Service object and its corresponding Endpoints, thereby achieving load balancing and traffic forwarding of service nodes with high concurrent architecture. The Kubernetes automatic scaling policy is a kind of responsive scaling, which essentially calculates the amount of scaling based on the current load of the Pod copy. For the expansion phase, when the Pod replica load is too high to trigger the HPA to expand, HPA calculates the expected number of copies and notifies the replica controller. The replica controller calculates the number of Pods to be created according to the expected value, and hands it to the corresponding component to perform specific creation jobs. The purpose of Kubernetes auto-reduction is to reduce the overall resource utilization of the replica set and reduce the deployment cost of the application by removing redundant copies when the Pod replica set load is low. The automatic retractor notifies the replica controller of the expected number of copies, and the replica controller checks the true value of the current number of copies. If the actual value is greater than the expected value, the replica controller deletes the excess copy.

HPA is a major feature of Kubernetes and is the core of this article. It dynamically monitors minute-level dynamics by periodically monitoring certain parameters in the Pod (for example, CPU usage, Memory usage, or TCP connections). At the same time, when the load is reduced, Kubernetes will gradually recycle the free containers. This technology is huge for a project that is concurrent with millions of terminals. It can make users allocate hardware resources more reasonably.

3. EXPANSION PRINCIPLE
Under the premise of maintaining the concurrent throughput of the system, the server resources are reduced, the containerized service is used to achieve lightweight storage of the hard disk resources, and the server CPU situation monitoring is automated to reduce nodes and maximize resource utilization.
The essence of HPA is the resource utilization of the periodic monitoring system. When the monitoring result exceeds the expected range, the expansion rule is triggered, and the pod operation is added/deleted to the corresponding Service. The initialized pod will become Ready state in a short time, then become ready to use the load balancing technology to integrate the new Pod into the Service to achieve capacity expansion.

3.1 Capacity calculation algorithm

\[ f(x) = \begin{cases} 
  x_{\text{Max}}, & x \geq x_{\text{Max}} \\
  \text{ceil} \left( x \cdot \left( \frac{\text{CV}}{\text{DV}} \right) \right), & x_{\text{Min}} < x < x_{\text{Max}} \\
  x_{\text{Min}}, & x \leq x_{\text{Min}} 
\end{cases} \quad (1) \]

where:

- \( f(x) \) = desiredReplicas
- \( x \) = currentReplicas
- \( CV \) = currentMetricValue
- \( DV \) = desiredMetricValue

Kubernetes has an error value \( \alpha \) that triggers a capacity expansion event if and only if \((\text{CD}/\text{DV}) > 1 + \alpha \) or \((\text{CD}/\text{DV}) < 1 - \alpha \). The default value of \( \alpha \) is 0.1, that is, the range of initial non-trigger expansion is \([0.9, 1.1]\), which is adjusted by modifying --horizontal-pod-autoscaler-tolerance.

3.2 Time required for expansion

\[ t_{\text{init}} = \sum_{i=1}^{4} t_i \quad (6) \]

t1 is the time taken by Kubernetes to detect that the resource exceeds the threshold, t2 is the time required to calculate the required number of pods, t3 is the creation time of the pod, and t4 is the time taken for the pod to become ready. Among them, t1 accounts for the highest proportion of the expansion time, and more than 50% of the time is occupied by t1.

3.3 Contraction related matters

When the amount of data is reduced, the related pod will reduce the load. When the corresponding pod is not called for a long time, Kubernetes will recycle the useless Pod. The time required for each Pod to trigger the contract is 5 minutes by default. It can be done by adjust parameters.

--horizontal-pod-autoscaler-downscale-stabilization.

When shrinking, the corresponding pod is first removed from the Service, and then the recovery operation is performed to complete the reduction.
4. TEST VERIFICATION
Objective: To simulate the actual situation, simplify the actual situation and focus on testing the Netty part, you need to make the jar file into a Docker image, and then deploy the corresponding Kubernetes service.

4.1 Containerization of services
Since Kubernetes is a Docker service, it is necessary to package the java program into a Docker image. The DockerFile content is as follows. Note the time zone setting and port exposure issues. The port exposure is used for Kubernetes to call, and the time zone involves application services.

FROM java:8
# ADD adc-netty-2.2.0.jar netty.jar
RUN bash -c 'touch /netty.jar'
# set timezone
ENV TZ=GMT+8
RUN ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && echo $TZ > /etc/timezone
# expose port
EXPOSE 80
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/netty.jar"]

4.2 Initialize Deployment and Service

Figure 2. Service and Pod Initialization.
Using the configuration file of Kubernetes to generate Deployment and Service, this article shows a simplified version, which needs to be configured according to the project needs, and the CPU initiation needs 200m. Deployment is used by subsequent Pods.

apiVersion: extensions/v1beta1
text: Deployment
metadata:
annotations:
development.kubernetes.io/revision: "1"
geneneration: 1
labels:
run: netty
name: netty
selfLink: /apis/extensions/v1beta1/namespaces/default/deployments/netty
spec:
replicas: 1
selector:
matchLabels:
run: netty
template:
The configuration of the Kubernetes deployment service is as follows. It is necessary to associate the container that has been packaged into the docker image and expose the port 80.

Service can be understood as a service consisting of all Netty in the whole architecture. Its characteristic is that for the whole, he only exposes one port, the actual business processing is handled by Pod, the data is processed by Pod, and the inter-framework is performed by Service. Data transfer, as well as the default load balancing algorithm of Kubernetes, can also be customized development.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: netty
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    run: netty
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
```

Execute the command to generate Deployment and Service.

Since the initial setting for Deployment for Pod is 1, the cluster is automatically generated at the same time.

```
pod: netty-7c444996b7-hvkb6
```

![Figure 3. Get Service and Pod Status.](image)

4.3 *Init Horizontal Pod Autoscaler*

The condition in the example is 20% cpu for one expansion, the initial 2 Netty, the upper limit 8 Netty.

```
kubectl autoscale deployment netty --cpu-percent=20 --min=2 --max=8
```
4.4 Carry out the load and verify the dynamic expansion effect
The condition in the example is 20% cpu for one expansion, the initial 2 netty, the upper limit of 8 netty.

Under the kubectl get hpa -w command, the netty information will continue to be printed. It can be seen that when the load starts, the load of a single node reaches 174% cpu, then the dynamic expansion mechanism takes effect, and the netty is amplified to 8 and the load is simultaneously loaded. Dropped to 60%.

4.5 Stop the load and verify the dynamic shrinkage effect
When the load is reduced, the netty expanded after a period of time will be recycled, and the dynamic expansion is completed.

Analysis
Figure 7. Count and Load.

0-3 minutes, is the time for Kubernetes initialization, generates 1 Pod.
At 3:30, complete HPA initialization, Pod becomes 2.
At 4:07, start the load.
At 4:33, the number of pods changed to four, at which point the load was 174%.
At 4:53, the Pod becomes 8 and then the minute is reduced to 63%, and the expansion is completed.
The load is stopped at 6:13 minutes, triggered at 11:15, and recycled at 11:40.

5. CONCLUSION
The tested overall architecture can be expanded in 1 minute when bursts of traffic into the service, and can be reduced in 5 minutes to achieve the desired results, and has been put into use in a certain architecture.

- Providing a more economical solution for real-time quantity collection of vehicles under 10,000-class vehicle terminals, so that system resources can be more rationally applied;
- Realizing the integration of Kubernetes cluster and service architecture, and virtualizing the overall architecture container;
- Exploring the principle of Kubernetes dynamic expansion.

REFERENCES
[1] J. Sheng Lebiao, You Weiqian, Zhang Yuqian, and Zhou Qinglin, “High availability and load balancing design of Kubernetes cluster,” Electronic Technology and Software Engineering, vol.7, pp. 1-3, 2019.
[2] D. Zhou Jiawei, “Design and implementation of Kubernetes cross-cluster management,” Zhejiang University, 2017.
[3] J. Zhang Jian and Xie Tianwei, “Research on platform-as-a-service architecture based on Docker,” Information Technology and Informatization, vol.10, pp.131-134, 2014.
[4] J. Wang Junxiang and Guo Lei, “Enterprise-class container cloud platform solution based on Kubernetes and Docker technology,” Journal of Shanghai Institute of Shipping and Transportation, 3rd, vol.41, pp. 51-57, 2018.
[5] D. Liu Huiyang, “Research and implementation of container monitoring and scheduling based on Docker,” South China University of Technology, 2016.
[6] D. Yang Jingqi, “Research on scalability of cloud business platform,” Beijing University of Posts and Telecommunications, 2014.
[7] D. Yang Pengfei, “Research and implementation of resource dynamic scheduling based on Kubernetes,” Zhejiang University, 2017.