Design and Implementation of Network Performance Testing System under Container Networks

Group 5: LUO Tian, WANG Haocheng, WANG Haowei,
WEI Zhanjun and ZHANG Yao

Abstract

Kubernetes is one of the most important infrastructures in cloud computing. The network virtualisation feature makes the containers running on Kubernetes form a large virtual network. In this paper, we will launch a research based on container network in Kubernetes to explore the communication principle between containers in the virtual network. The core part of this paper is to design and implement a semi-automated network performance testing system in Kubernetes and use this system to carry out container network performance testing experiments and produce results.

***Keywords*: Kubernetes, Docker, CNI, iPerf, AWS**

1 Introduction

In the era of cloud computing, virtualization technology was introduced to solve the problem of low resource utilization. It allows us to run multiple virtual machines (VM) on a single physical server, making better use of physical server resources. Container is a technology that is more lightweight than VM [1]. Similarly, each container has its own file system, CPU, memory, process space, etc., but the operating system kernel can be shared between containers on the same physical server. This characteristic reduces some performance losses and further improves the utilization of physical resources.

Kubernetes (K8S) is an open-source container orchestration platform for automating the deployment, scaling, and management of containerized applications [2]. A cluster often consists of thousands of physical servers in large enterprise scenarios. Applications are deployed, upgraded, and offline almost every moment. With the help of Kubernetes, developers do not need to pay attention to the underlying hardware information of the cluster. They only need to hand over the packaged application to Kubernetes; then, it can automatically schedule a node in the cluster and start the application container.

As the adoption of Kubernetes grows, it becomes increasingly important to understand and evaluate the network performance within a Kubernetes cluster. Network performance plays a crucial role in the overall efficiency and reliability of containerized applications, especially in scenarios involving inter-Pod communication within the same node and across different nodes.

This project aims to develop a tool for testing and evaluating the network performance of Pods in a Kubernetes environment. By utilizing well-known network diagnostic tools such as ping, traceroute, and iperf [3], this project assesses the communication efficiency between Pods. These tools provide a comprehensive view of network characteristics and help identify potential issues. Ping is a fundamental tool that measures the round-trip time (RTT) between hosts, providing insights into network latency. Traceroute helps determine the network path taken by packets from the source to the destination, identifying potential routing issues or bottlenecks. Iperf is a powerful tool for measuring network bandwidth, allowing the assessment of throughput and performance characteristics. The outcomes of this project are instrumental for developers and system administrators in optimizing their Kubernetes network configurations and in ensuring robust and efficient service delivery.

2 Technical Research

2.1 Background Analysis

Network virtualization is one of the essential technologies in Kubernetes. In order to meet the need for mutual communication between applications, in Kubernetes, each Pod is assigned its virtual IP address. These Pods form a sizeable virtual network in Kubernetes, an overlay network. However, these virtual IP addresses do not exist in the underlying physical network of the cluster (i.e., the underlay network), resulting in Pods having no way to communicate directly. Therefore, we must use third-party plugins to achieve interoperability between overlay networks. In order to solve this problem, Kubernetes provides the standard Container Network Interface (CNI), which defines a set of standard interfaces for transmitting overlay network data in the underlay network. Any network plugin that implements the CNI interface can be integrated into Kubernetes. There are currently many CNI plugins based on different technologies, such as Flannel, Calico, Cilium, Kube-OVN, etc. In this section, we choose Flannel as the object of study, and we will specifically study how Flannel enables Pods to communicate under the same node as well as across nodes in VXLAN [4] mode.

2.2 Solution Study

Assume that there is a network structure as shown in Figure 1. Node 1 and 2 are the two underlying hosts of K8S. Pod 1 and 2 are running on Node 1, while Pod 3 is on Node 2. Node 1 and 2 can communicate with each other through the physical switch.

2.2.1 Pod communication on the same host

Pod communication on the same host will use cni0, cni0 is a virtual bridge created by Flannel. Pod 1 and 2 are connected to this bridge using veth pair. A veth pair can be understood as a virtual network cable. Any data generated at one end will appear at the other end at the same time. The veth pair's end of the Pod is regarded as the eth0 network card in the container, which is the default network card of the container.

We can see that Pod 1 and 2 are in the same subnet, so just like the ordinary layer 2 network communication, the packet sent by Pod 1 to Pod 2 comes to Pod 2 through cni0, and vice versa.

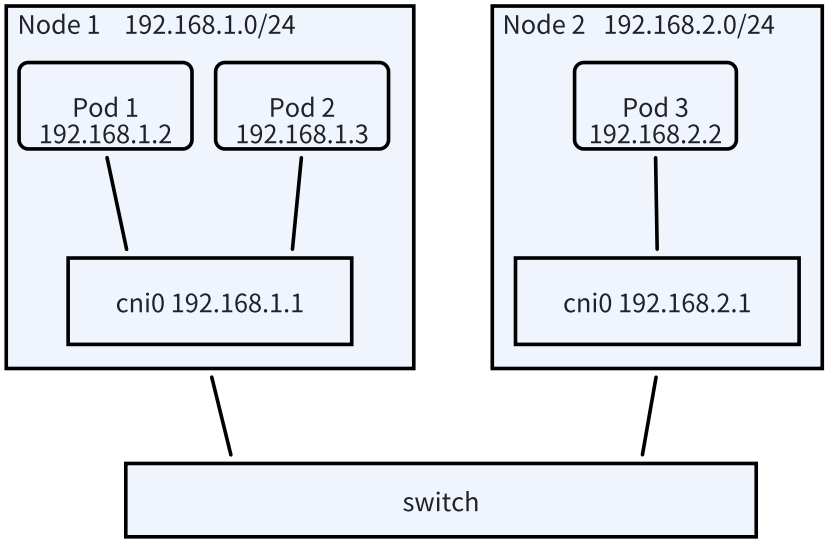


Figure 1: Network structure

2.2.2 Pod communication on different hosts

What happens if Pod 1 wants to send a packet to Pod 3? It can be a little complex. Because Pod 1 and Pod 3 are not in the same subnet, Pod 1 must use the default gateway to communicate. Pod 1 will send the packet to 192.168.1.1 first, which is the default gateway of Pod 1 and also the IP address of cni0. Actually, cni0 is also regarded as the network card on Node 1, so the packets sent to cni0 will enter the host's kernel protocol stack.

After Pod 1's packet reaches Node 1's protocol stack, Node 1 learns that this packet should be sent to Node 2 through its destination IP. So Node 1 will make a VXLAN encapsulation based on the original packet, the source IP address of VXLAN packet is the IP address of Node 1, while the destination IP address of VXLAN packet is the IP address of Node 2. Then, Node 1 will send the VXLAN packet through the physical network. Finally, the VXLAN packet came to Node 2 and was decapsulated, and the original packet was sent to Pod 3.

During this process, Pod 1 does not perceive the existence of the VXLAN packet and the underlying physical network. It completes the communication just like ordinary cross-subnet communication.

3 System Design

From the above analysis, we can conclude that the communication between Pods in K8S needs to be implemented with the help of CNI, and compared to the native Linux protocol stack, there are some extra encapsulation and de-encapsulation operations in the packet processing process, thus causing a certain degree of performance loss. In order to investigate the network performance of inter-pod communication, we design a semi-automated network performance testing system under K8S, and this section will elaborate the design principle and solution of the system.

3.1 Architecture

The following diagram (Figure 2) illustrates the basic components of the project:

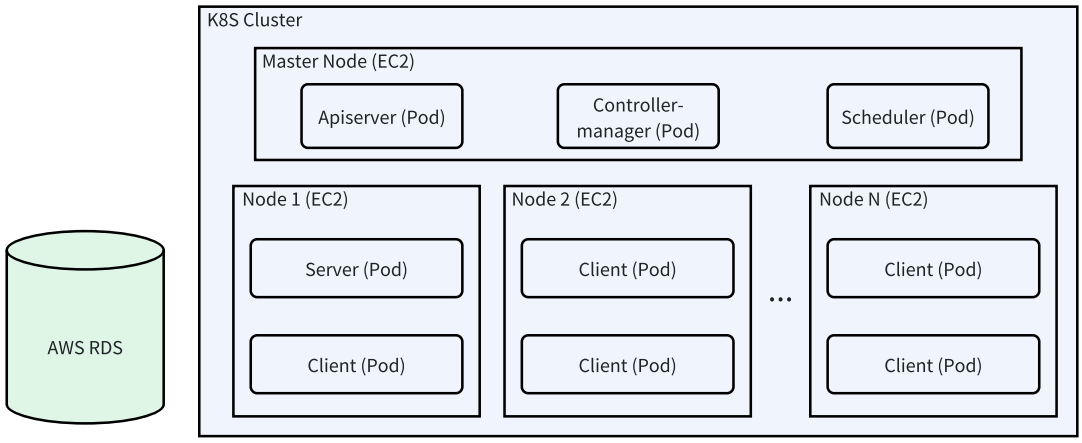


Figure 2: Architecture of our system

There are two types of nodes in a K8S cluster, Master and Node, which are both EC2 VMs on the cloud. Among them, Master is the control plane of the cluster. Apiserver, Controller-manager and Scheduler are K8S's own components that run on Master as Pods. Our project is deployed on Nodes, where Client is the executor of the network performance tests, which is deployed as a Pod on each Node. In order to measure the network communication performance on the same host, the number of Clients deployed on a Node is greater than 1. Server is the scheduler of network performance test tasks, which is responsible for managing the lifecycle of all Clients in the cluster, dispatching the test tasks and collecting the results. Outside the K8S cluster, we also deployed a database as a data persistence storage unit. Server stores all Client states, task states and results in the database so that the data will not be lost after the entire project is restarted.

Figure 3 is a simplified sequence diagram of this system, and we will elaborate on the interaction flow of the various modules in the system in the following sections:

At the beginning of this workflow, the user interacts with the Apiserver using the kubectl tool to deploy the Server and Client to the cluster. Once the Client is up, it starts sending heartbeats to the Server at a rate of 1 per second. Then user can query the details of the Client that is currently online in the cluster through the API provided by the Server, including the IP and UID of the Pod. The user can dispatch tasks to the Server based on the Client information obtained. The metadata of the task include the test target, the type of the task, the parameters of the task, etc. The Server will dispatch the task to the corresponding Client periodically based on the test target. The Client will report the results to the Server after executing the task. Finally, the user can request the execution result of a task from the Server.

3.2 Client Design

The design of Client is divided into three layers: communication layer, job layer and execution layer. The communication layer consists of the HTTP module, which receives calls from the job layer, sends HTTP requests to the Server, and parses the responses back

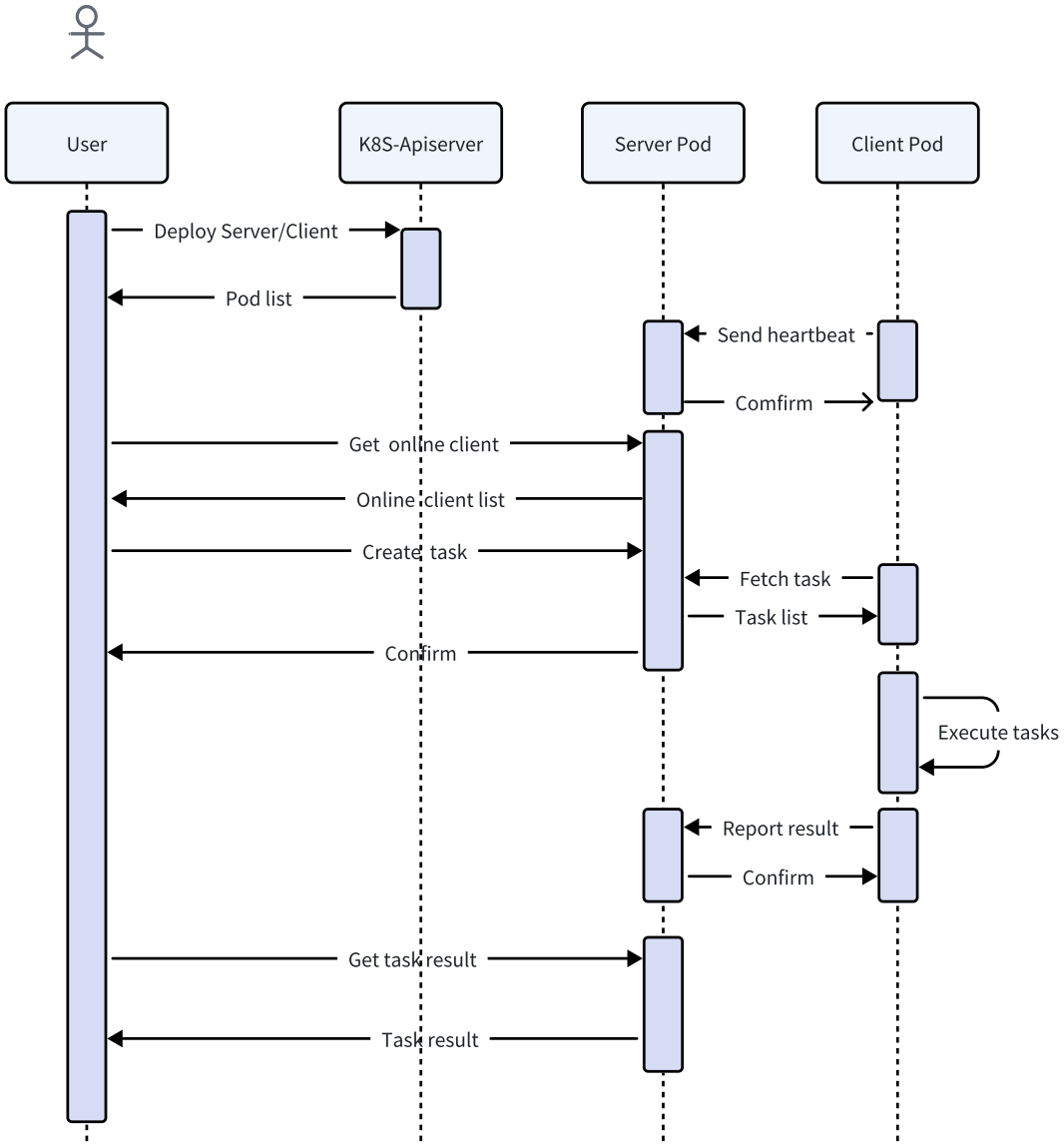


Figure 3: Sequence diagram

to the job layer. Three jobs are defined in the job layer, the heartbeat sending job, the task fetching job, and the task reporting job. The heartbeat sending job sends its own global context (They are read from the Pod environment variables when the Client starts, and environment variables are automatically injected when K8S starts the Pod) to the Server once a second. The task fetching job also fetches tasks from the Server at a rate of once a second, using the Pod UID as its unique identifier (it is impossible to have Pods with the same UID in the cluster), and the Server returns to it the tasks that need to be executed based on the UID. After the Client fetches the tasks, it passes them to the task execution pool via a channel. The task execution pool enables the Client to execute multiple tasks in parallel. The task report job will continuously poll the task execution pool and send the status and results of the completed task back to the Server.

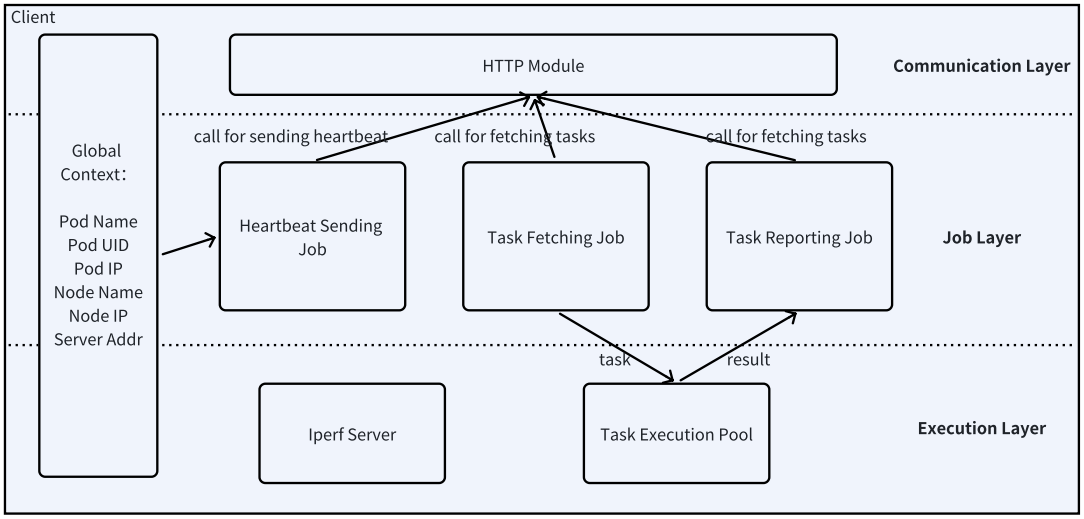


Figure 4: Client layers

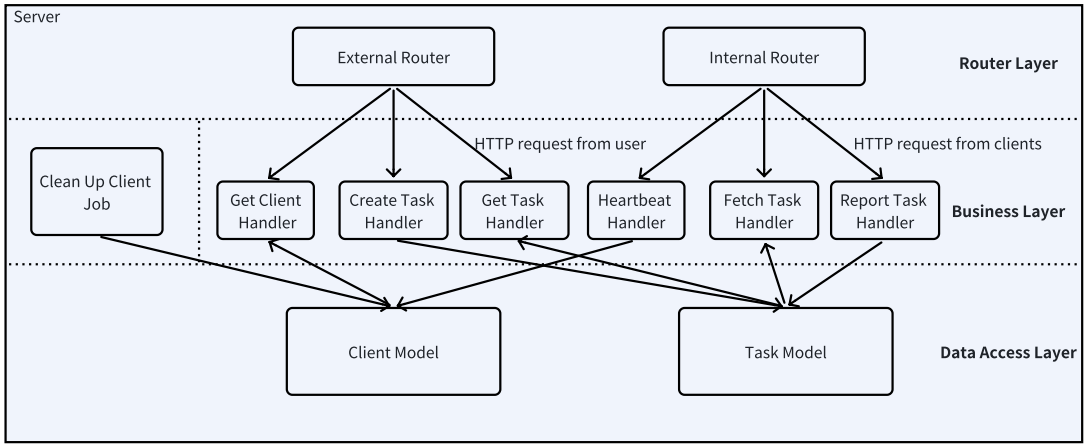


Figure 5: Server layers

3.3 Server Design

The core component of Server is Gin, which is an HTTP framework under the Go language. The Server is also designed as a three-layer structure: the Router Layer is responsible for receiving incoming HTTP requests and assigning them to the corresponding handlers based on the request paths; the requests received by the Server are divided into internal requests from the Client and external requests from the user, all APIs are listed in Table 1; The Business Layer is responsible for the specific request execution logic, such as converting the Client's heartbeat request to the corresponding Client model in the database, and converting the user's request for creating a task to the corresponding Task model in the database. The Data Assess Layer interacts directly with the database, and is responsible for a variety of data operations, such as adding, deleting, updating, and checking.

In Business Layer, in addition to the API Handler, there is another timed Job: Clean Up Client Job which is similar to the design idea of Job in Client. This job is executed

Table 1: All APIs in our systems

Type	Method	Path	Parameters	Function
Internal (For Client)	POST	/api/v1/internal/heartbeat	Client Global Context	Receive self-reported information from client
	GET	/api/v1/internal/get_tasks	Pod UID	Pull the tasks that need to be executed
	POST	/api/v1/internal/report_task	Task Status, Task Result	Report tasks result
External (For User)	GET	/api/v1/external/clients	-	Get the list of online clients
	POST	/api/v1/external/task/create	Task Metadata	Create network test tasks
	GET	/api/v1/external/task	Task ID	Query task results

every 3 seconds, it will mark all the clients in the database whose last heartbeat time is more than 10s away from the current time as offline. The reason for this is that there is no way to notify the Server when a Client goes offline in the cluster, so data cleaning needs to be done periodically. In addition, when a user requests a list of clients from the Server, the Server will only return the clients that are currently online (there is no meaning to return offline clients).

4 Experiment Result

In this section, we deployed the above system in a K8S cluster on AWS. We used 3 AWS EC2 servers, one as Master and two as Node. We deployed Flannel as CNI in the cluster with VXLAN mode enabled. After the deployment, we measured the network communication performance between Pods using Ping and Iperf programs, and the test results are shown in Table 2.

Table 2: Results

Communication mode	TCP bandwidth (Mbps)	Average RTT (ms)
Pods within the Node	2687	0.091
Pods across the Node	796	0.375
Node to Node directly	948	0.223

5 Conclusion

In this project, we explored the communication principles of container networks in Kubernetes, designed a container network performance testing system and implemented it. We built our own Kubernetes cluster and deployed our system on it for network performance testing. The test results show that container network technology introduces a performance loss (16%) compared to physical networks, and how to optimise this part is a key issue for academia and industry to focus on.

Due to the lack of front-end developers in our group this semester, our system can only interact with users in the form of an API, and in our future work, we will try to develop a set of front-end pages to improve the user experience.

References

[1] M. Eder, "Hypervisor-vs. container-based virtualization," *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)*, vol. 1, 2016.

[2] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, 2016.

[3] J. Dugan, S. Elliott, B. A. Mah, J. Poskanzer, and K. Prabhu, "iperf - the ultimate speed test tool for tcp, udp and sctp," https://iperf.fr.

[4] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright, "Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks," Tech. Rep., 2014.

Appendix A: Software Artifact

Abstract

In this artifact, we provide the project's source code repository address, container image repository address. We demonstrate the process of building a Kubernetes cluster on EC2 and deploying our project in the cluster. Finally, we demonstrate how to perform semi-automated network performance testing with our project and show the results.

Description

Our system is developed in Go and runs as a container on Kubernetes, with the relevant code, image build files, and K8S configuration files hosted on a GitHub repository[1]. In addition, we uploaded the compiled client image[2] and server image[3] to DockerHub.

Hardware dependencies

3 $\times$ AWS EC2 (1 Master, 2 Node). Recommended Configuration:

- vCPU: ≥ 2
- RAM: $\geq$ 8GB
- Disk: $\geq$ 50GB

1 $\times$ AWS RDS. Recommended Configuration:

- vCPU: ≥ 2
- RAM: RAM: $\geq$ 4GB
- Disk: Disk: $\geq$ 100GB

Software dependencies

OS:

- Ubuntu 20.04 LTS (amd64)

Software:

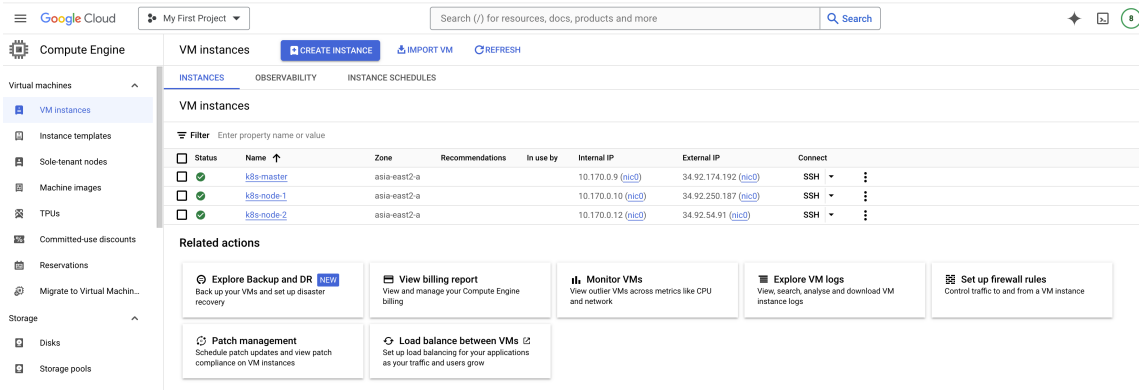
- Docker
- apt-transport-https
- ca-certificates
- curl

[1] https://github.com/SunSetPilot/cs5296-project
[2] https://hub.docker.com/repository/docker/ahussp/cs5296-client/general
[3] https://hub.docker.com/repository/docker/ahussp/cs5296-server/general

Installation

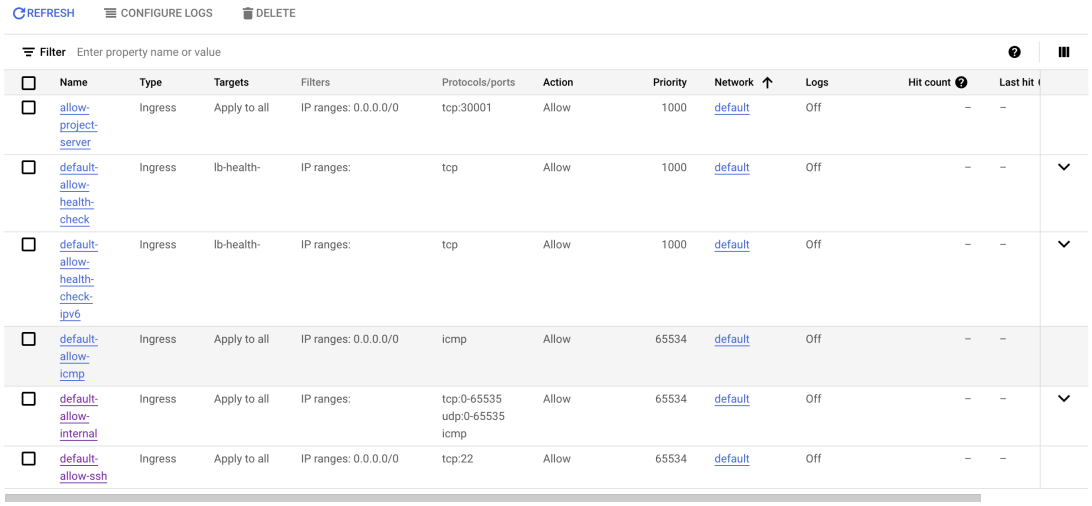
In this section we describe how to build a Kubernetes cluster using EC2. Since our AWS account is no longer available at the time of writing this document, we use Google Cloud instead. The steps are basically the same except that creating instances and configuring the network security group is different.

1. Create 3 VM instances

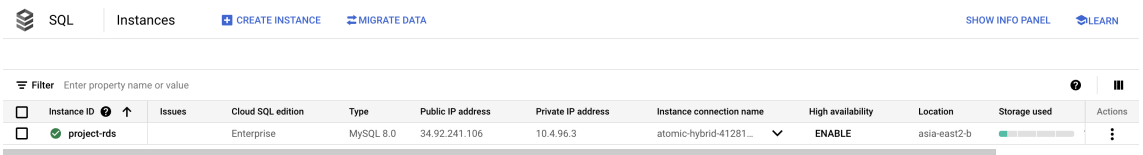


2. Add VPC firerules rules

Allow ingress rule 0.0.0.0/0 for TCP port 30001 (for user to access the Server)



3. Create MySQL instance



4. Install bridge-utils on each EC2 instance

```
weizhanjun@k8s-master:~$ sudo apt-get install -y bridge-utils
Reading package lists... Done
Building dependency tree
Reading state information... Done
Suggested packages:
  ifupdown
The following NEW packages will be installed:
  bridge-utils
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 30.5 kB of archives.
After this operation, 112 kB of additional disk space will be used.
Get:1 http://asia-east2.gce.archive.ubuntu.com/ubuntu focal/main amd64 bridge-utils amd64 1.6-2ubunt
u1 [30.5 kB]
Fetched 30.5 kB in 0s (562 kB/s)
Selecting previously unselected package bridge-utils.
(Reading database ... 62149 files and directories currently installed.)
Preparing to unpack .../bridge-utils_1.6-2ubuntu1_amd64.deb ...
Unpacking bridge-utils (1.6-2ubuntu1) ...
Setting up bridge-utils (1.6-2ubuntu1) ...
Processing triggers for man-db (2.9.1-1) ...
weizhanjun@k8s-master:~$ sudo modprobe br_netfilter
weizhanjun@k8s-master:~$ lsmod | grep br_netfilter
br_netfilter           28672  0
bridge                307200  1 br_netfilter
```

5. Install Kubelet, Kubeadm, Kebuctl on each EC2 instance

```
sudo mkdir /etc/apt/keyrings
curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.28/deb/
   Release.key | sudo gpg --dearmor -o /etc/apt/keyrings/
   kubernetes-apt-keyring.gpg
echo "deb [signed-by=/etc/apt/keyrings/kubernetes-apt-
   keyring.gpg] https://pkgs.k8s.io/core:/stable:/v1.28/
   deb/ /" | sudo tee /etc/apt/sources.list.d/kubernetes.
   list
sudo apt-get update
sudo apt install -y kubelet kubeadm kubectl
```

```
weizhanjun@k8s-master:~$ sudo apt install -y kubelet kubeadm kubectl
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  conntrack cri-tools ebtables kubernetes-cni socat
Suggested packages:
  nftables
The following NEW packages will be installed:
  conntrack cri-tools ebtables kubeadm kubectl kubelet kubernetes-cni socat
0 upgraded, 8 newly installed, 0 to remove and 0 not upgraded.
Need to get 87.6 MB of archives.
After this operation, 335 MB of additional disk space will be used.
Get:1 http://asia-east2.gce.archive.ubuntu.com/ubuntu focal/main amd64 conntrack amd64 1:1.4.5-2 [30
.3 kB]
Get:2 http://asia-east2.gce.archive.ubuntu.com/ubuntu focal/main amd64 ebtables amd64 2.0.11-3build1
 [80.3 kB]
Get:3 http://asia-east2.gce.archive.ubuntu.com/ubuntu focal/main amd64 socat amd64 1.7.3.3-2 [323 kB
]
Get:4 https://prod-cdn.packages.k8s.io/repositories/isv:/kubernetes:/core:/stable:/v1.28/deb  cri-to
ols 1.28.0-1.1 [19.6 MB]
Get:5 https://prod-cdn.packages.k8s.io/repositories/isv:/kubernetes:/core:/stable:/v1.28/deb  kubern
etes-cni 1.2.0-2.1 [27.6 MB]
Get:6 https://prod-cdn.packages.k8s.io/repositories/isv:/kubernetes:/core:/stable:/v1.28/deb  kubele
t 1.28.9-2.1 [19.5 MB]
Get:7 https://prod-cdn.packages.k8s.io/repositories/isv:/kubernetes:/core:/stable:/v1.28/deb  kubect
l 1.28.9-2.1 [10.3 MB]
Get:8 https://prod-cdn.packages.k8s.io/repositories/isv:/kubernetes:/core:/stable:/v1.28/deb  kubead
m 1.28.9-2.1 [10.1 MB]
Fetched 87.6 MB in 5s (16.3 MB/s)
Selecting previously unselected package conntrack.
(Reading database ... 62180 files and directories currently installed.)
```

6. Install Docker on each EC2 instance

```
sudo apt install docker.io
sudo systemctl start docker
sudo systemctl enable docker
```

```
weizhanjun@k8s-master:~$ sudo systemctl start docker
weizhanjun@k8s-master:~$ sudo systemctl enable docker
weizhanjun@k8s-master:~$ docker --version
Docker version 24.0.5, build 24.0.5-0ubuntu1~20.04.1
weizhanjun@k8s-master:~$
```

7. Configuring the Kubernetes Master

```
sudo kubeadm config images pull
```

```
weizhanjun@k8s-master:~$ sudo kubeadm config images pull
I0427 13:26:02.723421    7420 version.go:256] remote version is much newer: v1.30.0; falling back to: stable-1.28
[config/images] Pulled registry.k8s.io/kube-apiserver:v1.28.9
[config/images] Pulled registry.k8s.io/kube-controller-manager:v1.28.9
[config/images] Pulled registry.k8s.io/kube-scheduler:v1.28.9
[config/images] Pulled registry.k8s.io/kube-proxy:v1.28.9
[config/images] Pulled registry.k8s.io/pause:3.9
[config/images] Pulled registry.k8s.io/etcd:3.5.12-0
[config/images] Pulled registry.k8s.io/coredns/coredns:v1.10.1
```

```
sudo kubeadm init --pod-network-cidr=10.244.0.0/16
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

After running the above command, we will get the command for Node to join the cluster.

```
To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

Alternatively, if you are the root user, you can run:

  export KUBECONFIG=/etc/kubernetes/admin.conf

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 10.170.0.9:6443 --token 1j1yvr.8vhfv7vui39w79d4 \
        --discovery-token-ca-cert-hash sha256:1b641b0f6493103bc7511799d371bb4341349e0157995b0562436f09a60226ff
```

8. Add the other 2 nodes to the cluster

```
weizhanjun@k8s-node-2:~$ sudo kubeadm join 10.170.0.9:6443 --token 1j1yvr.8vhfv7vui39w79d4 --discovery-token-ca-cert-hash sha256
:1b641b0f6493103bc7511799d371bb4341349e0157995b0562436f09a60226ff
[preflight] Running pre-flight checks
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -o yaml'
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Starting the kubelet
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...

This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.
```

View cluster node information.

```
weizhanjun@k8s-master:~$ kubectl get nodes
NAME         STATUS     ROLES           AGE     VERSION
k8s-master   NotReady   control-plane   5m16s   v1.28.9
k8s-node-1   NotReady   <none>          30s     v1.28.9
k8s-node-2   NotReady   <none>          20s     v1.28.9
```

9. Install the CNI.

```
kubectl apply -f https://github.com/flannel-io/flannel/
   releases/latest/download/kube-flannel.yml
```

```
weizhanjun@k8s-master:~$ kubectl apply -f https://github.com/flannel-io/flannel/releases/latest/download/kube-flannel.yml
namespace/kube-flannel created
serviceaccount/flannel created
clusterrole.rbac.authorization.k8s.io/flannel created
clusterrolebinding.rbac.authorization.k8s.io/flannel created
configmap/kube-flannel-cfg created
daemonset.apps/kube-flannel-ds created
```

After a successful CNI installation, we can see that all nodes are in the Ready state.

```
weizhanjun@k8s-master:~$ kubectl get nodes
NAME         STATUS   ROLES           AGE     VERSION
k8s-master   Ready    control-plane   7m53s   v1.28.9
k8s-node-1   Ready    <none>          3m7s    v1.28.9
k8s-node-2   Ready    <none>          2m57s   v1.28.9
```

At this point, the Kubernetes cluster has been set up, and below we will demonstrate how to deploy our project to the cluster. The configuration files you use are placed in the ”etc” folder in the git repository directory.

10. Deploying MySQL Endpoint and MySQL Service

Since MySQL is not deployed inside the cluster, the MySQL Endpoint and MySQL Service need to be deployed so that the Pod can access the MySQL database.

Remember to change the MySQL service IP address in the configuration file.

```
kubectl apply -f mysql-endpoints.yaml
kubectl apply -f mysql-service.yaml
```

```
weizhanjun@k8s-master:~$ vim mysql-endpoints.yaml
weizhanjun@k8s-master:~$ kubectl apply -f mysql-endpoints.yaml
endpoints/mysql created
weizhanjun@k8s-master:~$ vim mysql-service.yaml
weizhanjun@k8s-master:~$ kubectl apply -f mysql-service.yaml
service/mysql created
weizhanjun@k8s-master:~$
```

11. Deploying Server and Client to a Cluster

```
kubectl apply -f cs5296-server.yaml
kubectl apply -f cs5296-client.yaml
```

```
weizhanjun@k8s-master:~$ vim cs5296-server.yaml
weizhanjun@k8s-master:~$ kubectl apply -f cs5296-server.yaml
deployment.apps/cs5296-server-deployment created
service/cs5296-server-service created
weizhanjun@k8s-master:~$ vim cs5296-client.yaml
weizhanjun@k8s-master:~$ kubectl apply -f cs5296-client.yaml
pod/cs5296-client-pod-node-1-1 created
pod/cs5296-client-pod-node-1-2 created
pod/cs5296-client-pod-node-2-1 created
```

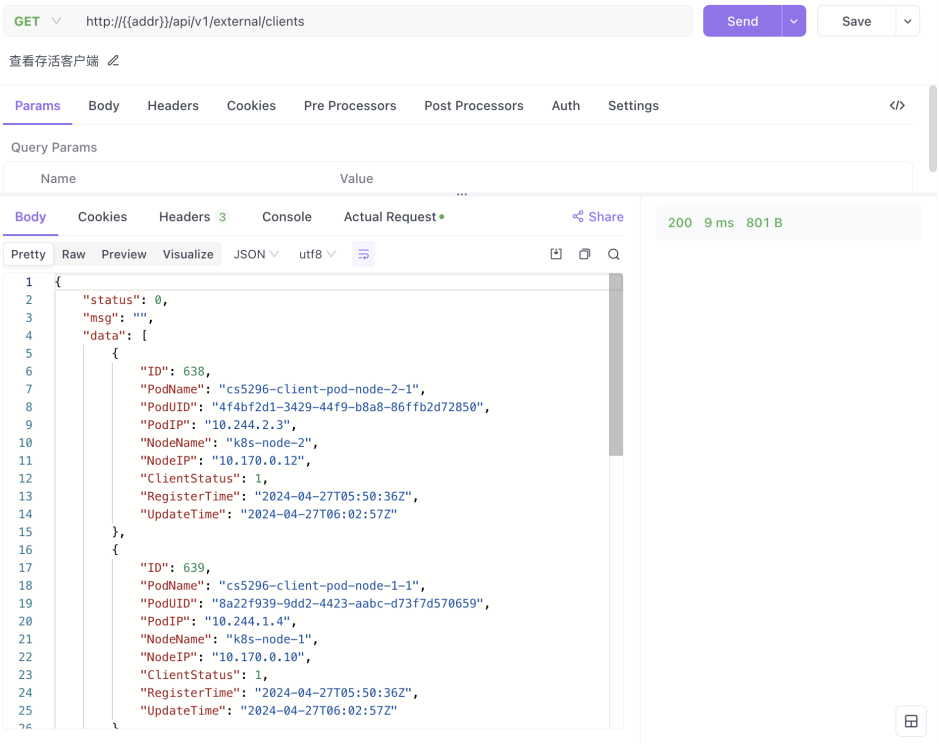
View Pods in the cluster.

```
weizhanjun@k8s-master:~$ kubectl get pods
NAME                                    READY   STATUS    RESTARTS   AGE
cs5296-client-pod-node-1-1              1/1     Running   0          2m37s
cs5296-client-pod-node-1-2              1/1     Running   0          2m37s
cs5296-client-pod-node-2-1              1/1     Running   0          2m37s
cs5296-server-deployment-646b5986cc-tgb48   1/1     Running   0          3m3s
weizhanjun@k8s-master:~$
```

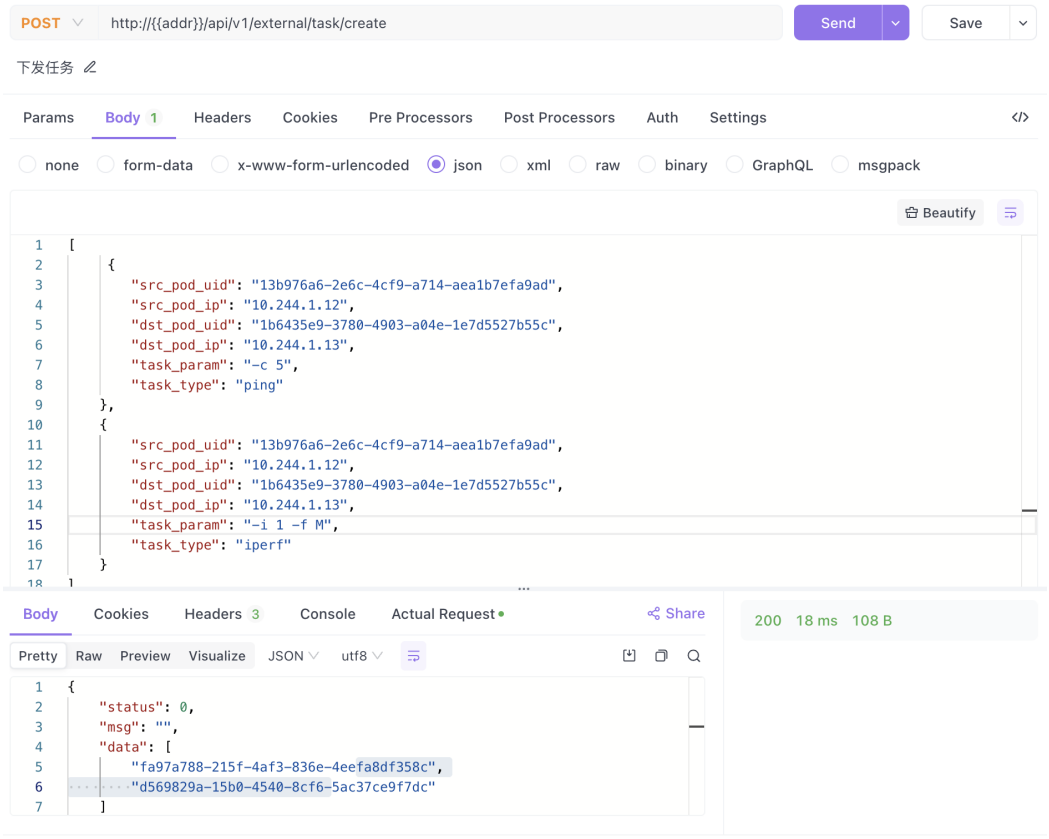
Experiment workflow

Since there is no front-end page for this system, we used the Apifox tool to carry out the experiments.

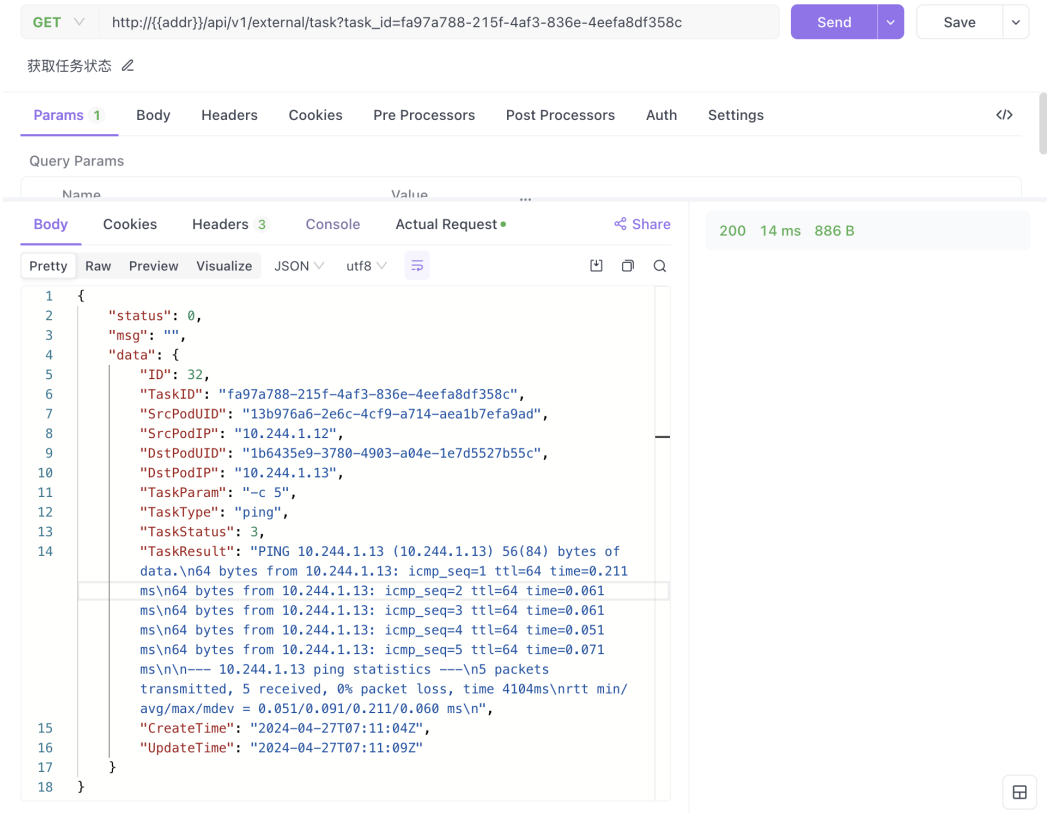
1. Get the list of online clients

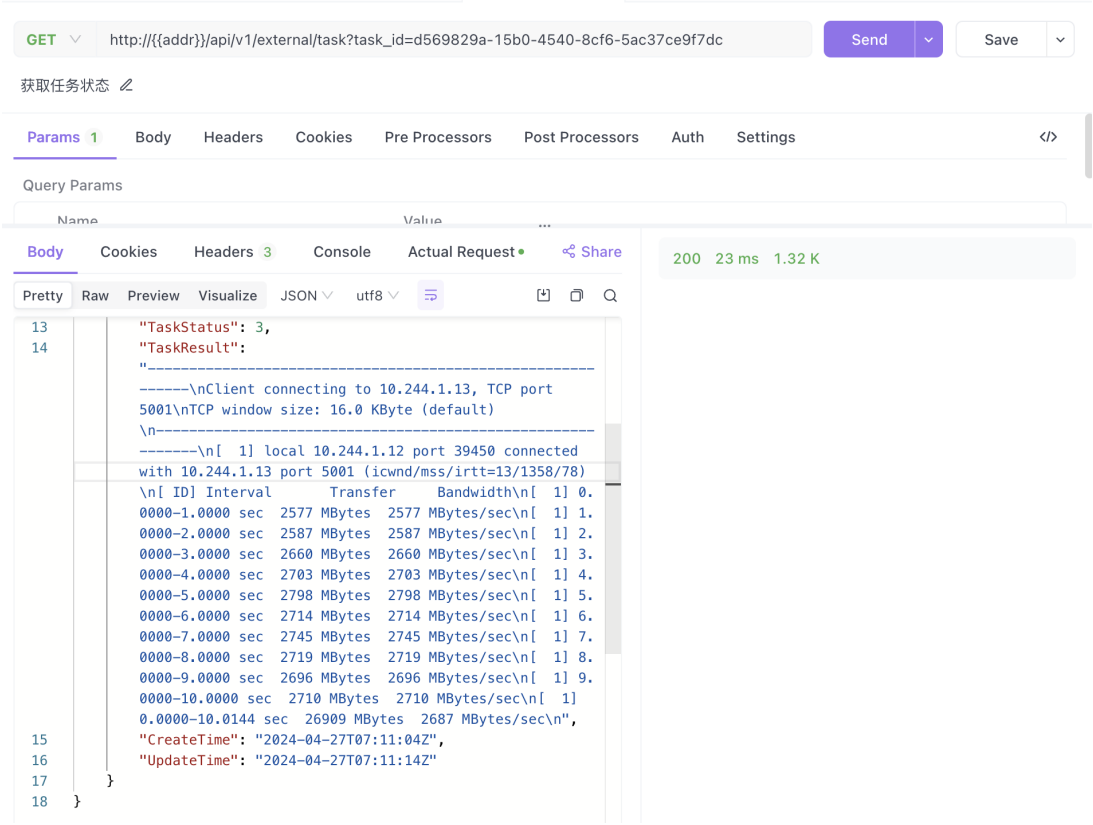


2. Dispatch same-node TCP bandwidth and RTT test task

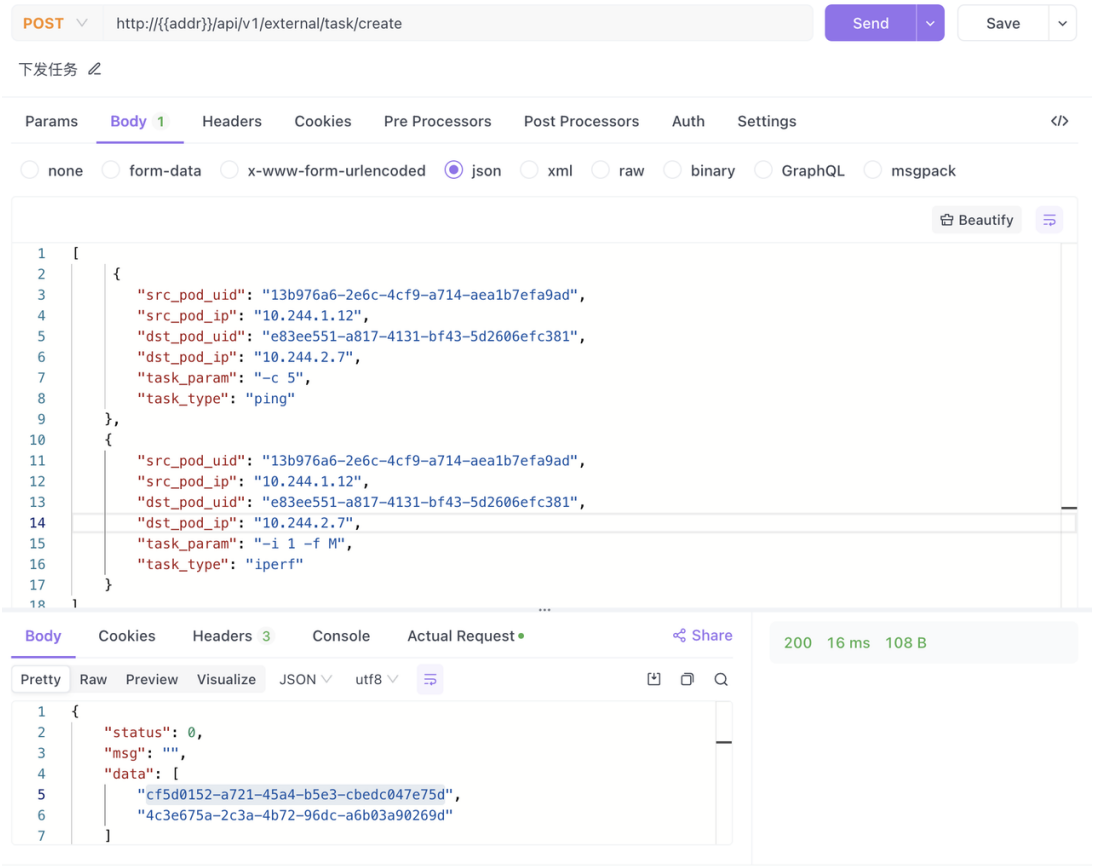


3. Get the results of the above tasks

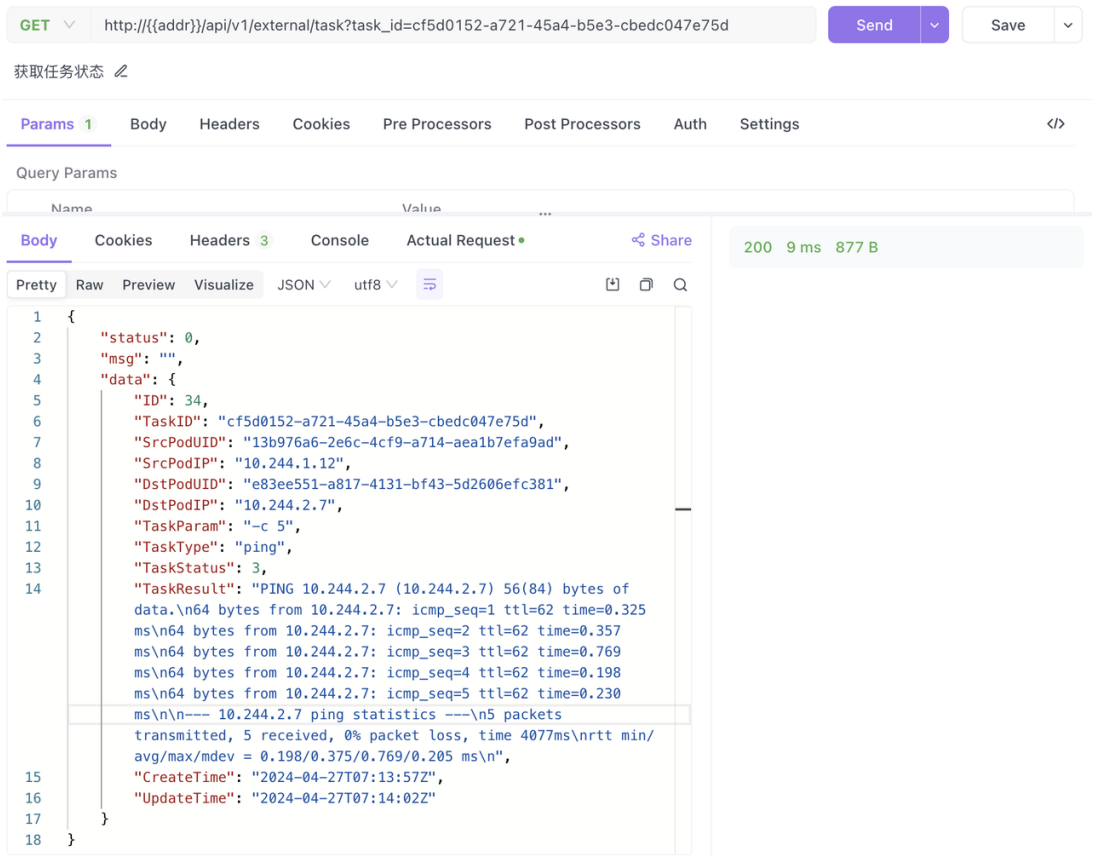


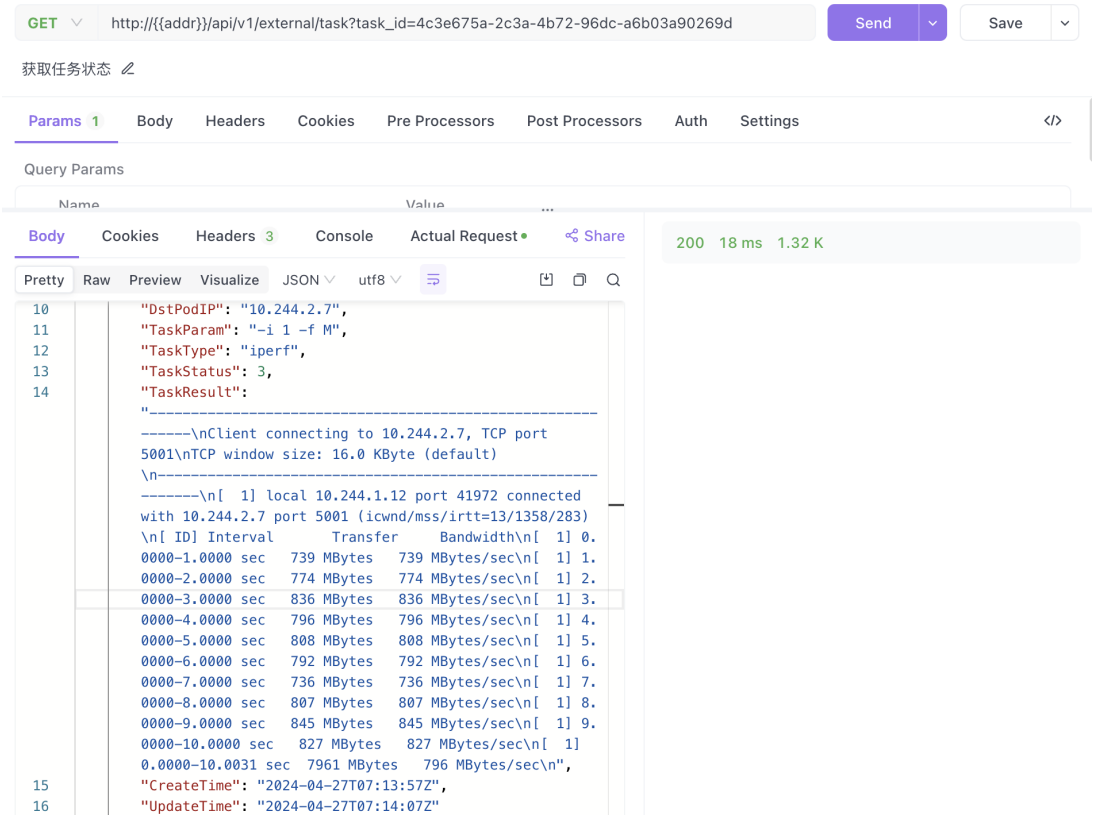


4. Dispatch cross-node TCP bandwidth and RTT test task



5. Get the results of the above tasks





6. Measurement of TCP bandwidth and RTT for direct communication between Nodes

```
weizhanjun@k8s-node-1:~$ iperf -c 10.170.0.12 -i 1 -f M
------------------------------------------------------------
Client connecting to 10.170.0.12, TCP port 5001
TCP window size: 3.59 MByte (default)
------------------------------------------------------------
[  3] local 10.170.0.10 port 51936 connected with 10.170.0.12 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec   905 MBytes   905 MBytes/sec
[  3]  1.0- 2.0 sec   953 MBytes   953 MBytes/sec
[  3]  2.0- 3.0 sec   952 MBytes   952 MBytes/sec
[  3]  3.0- 4.0 sec   953 MBytes   953 MBytes/sec
[  3]  4.0- 5.0 sec   952 MBytes   952 MBytes/sec
[  3]  5.0- 6.0 sec   952 MBytes   952 MBytes/sec
[  3]  6.0- 7.0 sec   953 MBytes   953 MBytes/sec
[  3]  7.0- 8.0 sec   953 MBytes   953 MBytes/sec
[  3]  8.0- 9.0 sec   953 MBytes   953 MBytes/sec
[  3]  9.0-10.0 sec   952 MBytes   952 MBytes/sec
[  3]  0.0-10.0 sec  9478 MBytes   948 MBytes/sec
```

```
weizhanjun@k8s-node-1:~$ ping 10.170.0.12 -c 5
PING 10.170.0.12 (10.170.0.12) 56(84) bytes of data.
64 bytes from 10.170.0.12: icmp_seq=1 ttl=64 time=0.262 ms
64 bytes from 10.170.0.12: icmp_seq=2 ttl=64 time=0.231 ms
64 bytes from 10.170.0.12: icmp_seq=3 ttl=64 time=0.189 ms
64 bytes from 10.170.0.12: icmp_seq=4 ttl=64 time=0.226 ms
64 bytes from 10.170.0.12: icmp_seq=5 ttl=64 time=0.208 ms

--- 10.170.0.12 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4082ms
rtt min/avg/max/mdev = 0.189/0.223/0.262/0.024 ms
```