POLITECNICO DI TORINO

01NWDBH - Mobile and Sensor Networks

Lab 04

Kubernetes

Group ID:
05

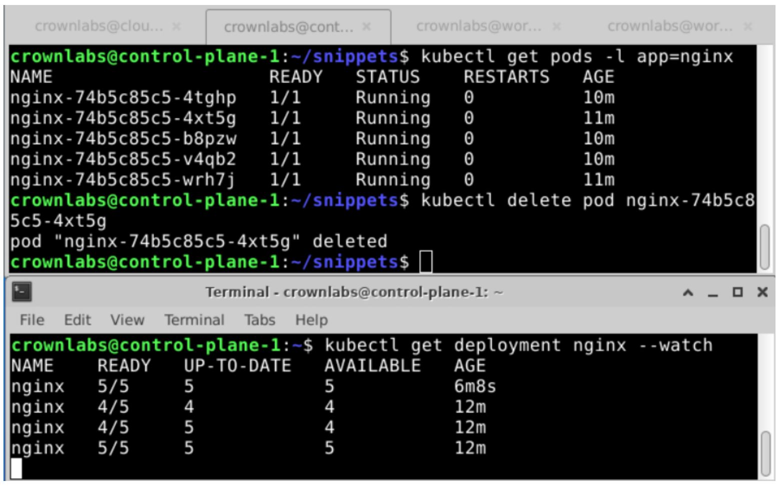
Students:

Valerio Collina	333919
Ammar Hussein	329829
Md Ismail Hossain	342704
Md. Ataur Rabby	347363

Academic year 2024/2025

Q1: Comment the result of the watch command.

A1: After deleting one nginx pod with kubectl delete pod, Kubernetes immediately recreated a new one to maintain the desired 5 replicas. This behavior was observed in real time using kubectl get deployment nginx –watch. This confirms that the Deployment controller ensures the specified number of replicas is always running. This behavior demonstrates the self-healing nature of Kubernetes Deployments.



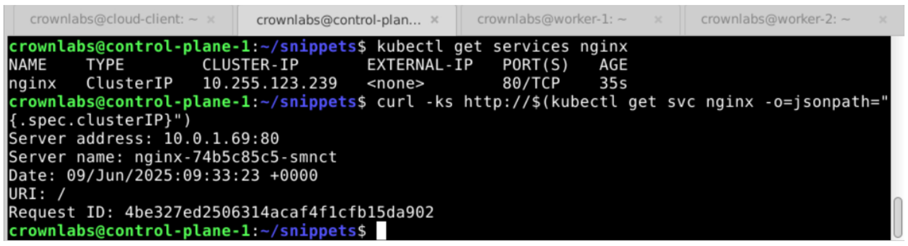
```
crownlabs@control-plane-1:~/snippets$ kubectl get pods -l app=nginx
NAME                     READY   STATUS    RESTARTS   AGE
nginx-74b5c85c5-4tghp    1/1     Running   0          10m
nginx-74b5c85c5-4xt5g    1/1     Running   0          11m
nginx-74b5c85c5-b8pzw    1/1     Running   0          10m
nginx-74b5c85c5-v4qb2    1/1     Running   0          10m
nginx-74b5c85c5-wrh7j    1/1     Running   0          11m
crownlabs@control-plane-1:~/snippets$ kubectl delete pod nginx-74b5c8
5c5-4xt5g
pod "nginx-74b5c85c5-4xt5g" deleted
crownlabs@control-plane-1:~/snippets$
```

```
crownlabs@control-plane-1:~$ kubectl get deployment nginx --watch
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
nginx   5/5     5            5           6m8s
nginx   4/5     4            4           12m
nginx   4/5     5            4           12m
nginx   5/5     5            5           12m
```

Figure 1: Deleting an nginx pod and real-time output of `kubectl get deployment nginx --watch` showing the pod count dropping to 4/5 and then returning to 5/5 as Kubernetes recreates the pod.

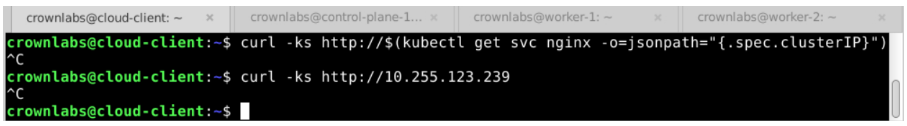
Q2: Run the curl command both in the client VM and in the control plane one. This command is working only on the control plane (master) node, why?

A2: Running the curl command on the control plane node showed the NGINX welcome page. This means the ClusterIP service works inside the cluster. Running the same command on the client VM failed because ClusterIP services are internal-only and inaccessible from external nodes, such as the client VM. This confirms that ClusterIP services are internal-only and not accessible from external nodes like the client VM.



```
crownlabs@control-plane-1:~/snippets$ kubectl get services nginx
NAME    TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
nginx   ClusterIP   10.255.123.239   <none>        80/TCP    35s
crownlabs@control-plane-1:~/snippets$ curl -ks http://$(kubectl get svc nginx -o=jsonpath="
{.spec.clusterIP}")
Server address: 10.0.1.69:80
Server name: nginx-74b5c85c5-smnct
Date: 09/Jun/2025:09:33:23 +0000
URI: /
Request ID: 4be327ed2506314acaf4f1cfb15da902
crownlabs@control-plane-1:~/snippets$
```

Figure 2: Output of curl on the control plane node showing the NGINX welcome page.

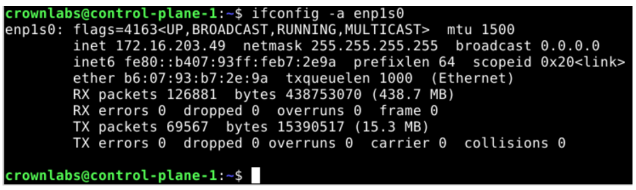


```
crownlabs@cloud-client:~$ curl -ks http://$(kubectl get svc nginx -o=jsonpath="{.spec.clusterIP}")
^C
crownlabs@cloud-client:~$ curl -ks http://10.255.123.239
^C
crownlabs@cloud-client:~$
```

Figure 3: Curl command failing on the client VM due to ClusterIP service being inaccessible from outside the cluster.

Q3: Which is your `IP_MASTER_NODE`? Which output do you get when running the curl command from the management workstation? Does it work, unlike in the previous case of a ClusterIP service? Do you get a similar output when accessing the service via your web browser? Finally, why does it work?

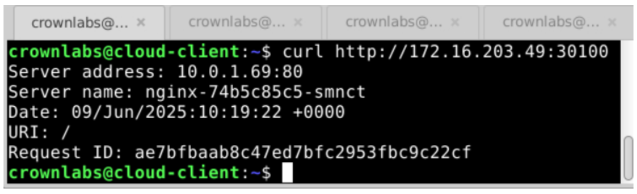
A3: The master node IP is `172.16.203.49`, found using `ifconfig` on the appropriate network interface `enp1s0`. The command `curl http://172.16.203.49:30100` from the client VM returned the NGINX welcome page successfully. Accessing the same URL from the browser also worked. This is because the NodePort service exposes the application on port 30100 on all cluster nodes, making it reachable from outside the cluster.



```
crownlabs@control-plane-1:~$ ifconfig -a enp1s0
enp1s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.16.203.49  netmask 255.255.255.255  broadcast 0.0.0.0
        inet6 fe80::b407:93ff:feb7:2e9a  prefixlen 64  scopeid 0x20<link>
        ether b6:07:93:b7:2e:9a  txqueuelen 1000  (Ethernet)
        RX packets 126881  bytes 438753070 (438.7 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 69567  bytes 15390517 (15.3 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

crownlabs@control-plane-1:~$ 
```

Figure 4: Master node IP address.

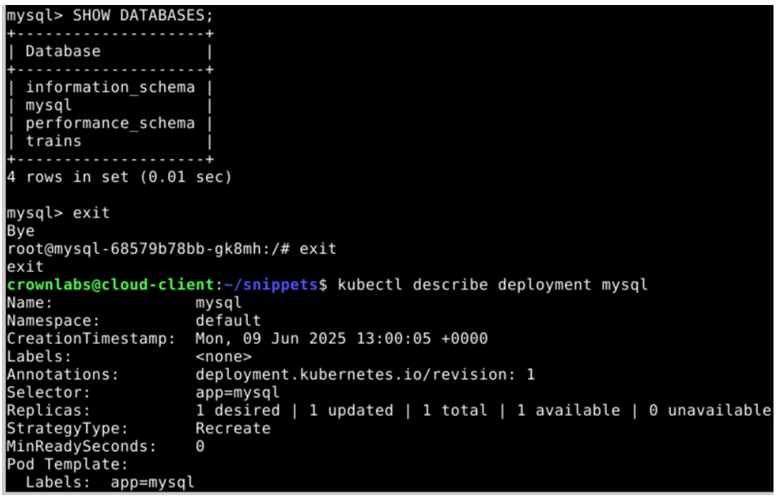


```
crownlabs@cloud-client:~$ curl http://172.16.203.49:30100
Server address: 10.0.1.69:80
Server name: nginx-74b5c85c5-smnct
Date: 09/Jun/2025:10:19:22 +0000
URI: /
Request ID: ae7bfbaab8c47ed7bfc2953fbc9c22cf
crownlabs@cloud-client:~$ 
```

Figure 5: Output of successful `curl` command accessing NGINX via NodePort.

Q4: Comment this result. In a real scenario, what could be the advantage of data persistence (consider a possible real-world application example, that offers some services to the users)? Analyze also the output of kubectl describe deployment mysql. Which is the namespace? Which is the name of the PVC? Which is the folder, local to the pod, that is mapped to the PV for persistent data storage? Finally, which events are listed? From the events, how many replicas of the MySQL database have been deployed?

A4: The `trains` database remains intact after deleting and recreating the MySQL pod, confirming that data is stored on a PersistentVolume. This allows data to persist across restarts — essential for applications like booking or e-commerce systems. The deployment is in the `default` namespace. The PVC is `mysql-pv-claim`, and it maps to `/var/lib/mysql` inside the pod. Events show that only one replica of the MySQL pod was created.



```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| trains             |
+--------------------+
4 rows in set (0.01 sec)

mysql> exit
Bye
root@mysql-68579b78bb-gk8mh:/# exit
exit
crownlabs@cloud-client:~/snippets$ kubectl describe deployment mysql
Name:               mysql
Namespace:          default
CreationTimestamp:  Mon, 09 Jun 2025 13:00:05 +0000
Labels:             <none>
Annotations:        deployment.kubernetes.io/revision: 1
Selector:           app=mysql
Replicas:           1 desired | 1 updated | 1 total | 1 available | 0 unavailable
StrategyType:       Recreate
MinReadySeconds:    0
Pod Template:
  Labels:  app=mysql
```

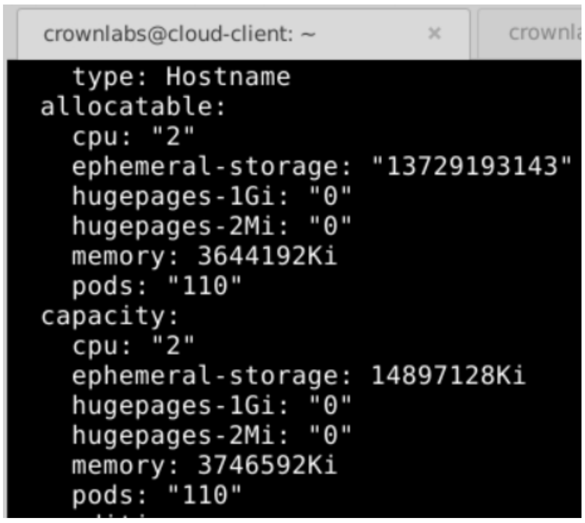
Figure 6: Summary of kubectl describe deployment mysql output, confirming data persistence and deployment details.

Q5: Identify and comment the fields stating the available resources.

A5: The node YAML includes two key fields describing resources:

- `capacity`, which is the total amount of CPU, memory, and storage available on the node.
- `allocatable`, showing how much of those resources can actually be used by pods after reserving some for system processes.

For example, the node has 2 CPUs and 3746592Ki of memory in total, but only 3644192Ki is allocatable. Storage also follows the same reservation principle.



```
    type: Hostname
  allocatable:
    cpu: "2"
    ephemeral-storage: "13729193143"
    hugepages-1Gi: "0"
    hugepages-2Mi: "0"
    memory: 3644192Ki
    pods: "110"
  capacity:
    cpu: "2"
    ephemeral-storage: 14897128Ki
    hugepages-1Gi: "0"
    hugepages-2Mi: "0"
    memory: 3746592Ki
    pods: "110"
```

Figure 7: YAML view of node resources

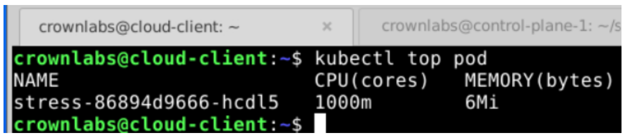
Q6: Compare CPU consumption before and after editing the deployment. Explain the difference between resource requests and resource limits.

A6: At first, the pod could use the full CPU because no limits were defined. After editing the deployment, we set: `requests.cpu = 100m` and `limits.cpu = 250m`.

This means:

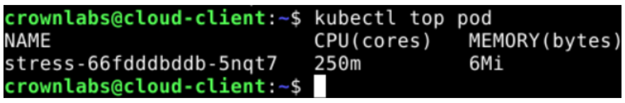
- `request` is the minimum CPU the pod is guaranteed (here, 0.1 CPU).
- `limit` is the maximum CPU the pod can use (here, 0.25 CPU).

As a result, the CPU usage dropped from 1000m to 250m. This can be seen using the `kubectl top pod` command. This demonstrates how resource limits prevent pods from over-consuming CPU, improving fairness and stability in multi-tenant clusters.



```
crownlabs@cloud-client:~$ kubectl top pod
NAME                        CPU(cores)   MEMORY(bytes)
stress-86894d9666-hcdl5     1000m        6Mi
crownlabs@cloud-client:~$
```

Figure 8: Before editing: pod consumes 1000m (1 CPU).

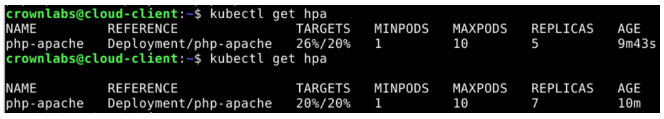


```
crownlabs@cloud-client:~$ kubectl top pod
NAME                        CPU(cores)   MEMORY(bytes)
stress-66fdddbddb-5nqt7     250m         6Mi
crownlabs@cloud-client:~$
```

Figure 9: After editing: pod limited to 250m.

Q7: How many replicas do you get in your case? Is it a reasonable number considering the CPU utilization target you imposed with –cpu-percent and the high percentage you saw the first time you ran kubectl get hpa? Why?

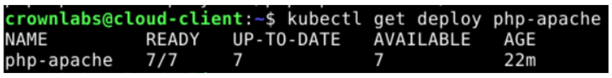
A7: In our case, the number of replicas reached **7**. This is reasonable given the CPU utilization target of 20% set during the creation of the Horizontal Pod Autoscaler (HPA). At the beginning, the CPU usage was very high (above 100%), which triggered the autoscaler to incrementally scale the number of pods. After a short time, it stabilized to 7 replicas, bringing the average CPU load closer to the target threshold. This behavior confirms the correct functioning of HPA: it adapts the number of pods based on real-time CPU metrics to meet the performance objectives without overloading the cluster.



```
crownlabs@cloud-client:~$ kubectl get hpa
NAME         REFERENCE                  TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache      26%/20%   1         10        5          9m43s
crownlabs@cloud-client:~$ kubectl get hpa

NAME         REFERENCE                  TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache      20%/20%   1         10        7          10m
```

Figure 10: HPA output showing 7 replicas and 14% average CPU usage vs. 20% target.



```
crownlabs@cloud-client:~$ kubectl get deploy php-apache
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
php-apache   7/7     7            7           22m
```

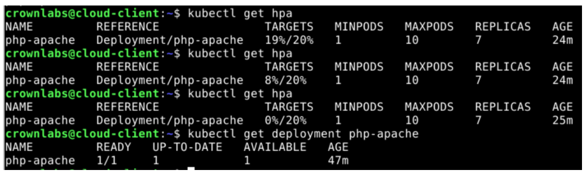
Figure 11: Deployment status showing all 7 replicas are ready and available.

Figure 12: Autoscaler result: stable scaling after load injection.

Q8: Imagine a real scenario, what could be the advantages of exploiting Kubernetes Horizontal Pod Autoscaling feature? Describe the positive impact of using HPA, giving a simple real-world application example.

A8: Horizontal Pod Autoscaling (HPA) is very useful in real-world situations where the workload changes a lot. It lets Kubernetes automatically increase or decrease the number of pod replicas based on CPU usage or other metrics, so the system can handle more traffic when needed and save resources when the traffic is low.

For example, we imagine an app that collects data from air quality sensors distributed across the city. Normally, only a few sensors send data, so we use just a few pods to process it. But during pollution alerts or big events, many sensors send a lot of data at once. Then, HPA automatically increases the number of pods to handle the extra load quickly. When the alert finishes and data returns to normal, HPA scales the pods down again. This way, we save resources and keep the app working well without manual intervention.

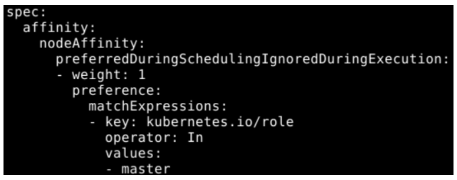


```
crownlabs@cloud-client:~$ kubectl get hpa
NAME         REFERENCE               TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   19%/20%   1         10        7          24m
crownlabs@cloud-client:~$ kubectl get hpa
NAME         REFERENCE               TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   8%/20%    1         10        7          24m
crownlabs@cloud-client:~$ kubectl get hpa
NAME         REFERENCE               TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   0%/20%    1         10        7          25m
crownlabs@cloud-client:~$ kubectl get deployment php-apache
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
php-apache   1/1     1            1           47m
```

Figure 13: HPA output after stopping the user load: CPU at 0% and replicas scaled back to 1.

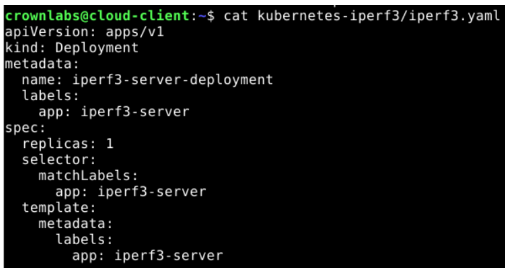
Q9:Analyze and discuss the YAML file, specifically highlighting which Kubernetes objects are going to be deployed and focusing on their specifications, e.g., affinity and tolerations. Furthermore, which kind of service is going to be deployed? Is it a NodePort or a ClusterIP?

A9: A9: We looked at the `iperf3.yaml` file. It has a Deployment with 1 replica of the `networkstatic/iperf3` container running on port 5201. The Deployment uses node affinity to **prefer** nodes labeled `kubernetes.io/role:` `master`, so it tries to run the pod on master nodes but can run elsewhere if needed. There is also a toleration to allow the pod to run on master nodes, even if those nodes have special taints that normally prevent pods from running there. The Service is of type **ClusterIP**, which means it only works inside the cluster and cannot be reached from outside.



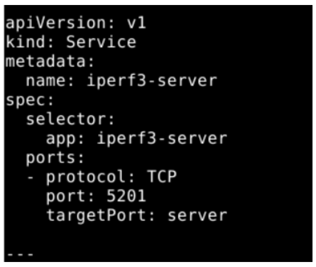
```
spec:
  affinity:
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: kubernetes.io/role
            operator: In
            values:
            - master
```

Figure 14: Affinity and tolerations



```
crownlabs@cloud-client:~$ cat kubernetes-iperf3/iperf3.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: iperf3-server-deployment
  labels:
    app: iperf3-server
spec:
  replicas: 1
  selector:
    matchLabels:
      app: iperf3-server
  template:
    metadata:
      labels:
        app: iperf3-server
```

Figure 15: Deployment metadata

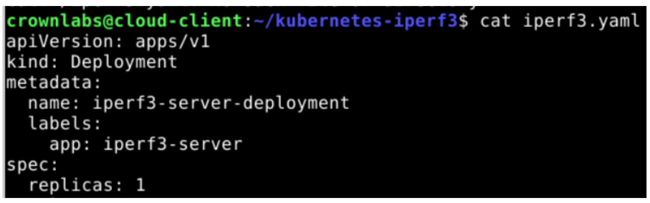


```
apiVersion: v1
kind: Service
metadata:
  name: iperf3-server
spec:
  selector:
    app: iperf3-server
  ports:
  - protocol: TCP
    port: 5201
    targetPort: server
---
```

Figure 16: ClusterIP service

Q10 part 1: Describe the role of each component (i.e. iperf3-server and iperf3-client). Moreover, explain why a Deployment, a Service and a DaemonSet are required to implement effectively this experiment. Furthermore, how many replicas are going to be deployed for the server? Where can you find this information in the YAML file?

A10 part 1: The `iperf3-server` runs as a Deployment with 1 replica (see `spec.replicas` in YAML). The `iperf3-client` runs as a DaemonSet to deploy one client pod on every node. The Service exposes the server internally to clients. Deployment manages server pods, DaemonSet ensures clients on all nodes, Service enables access.



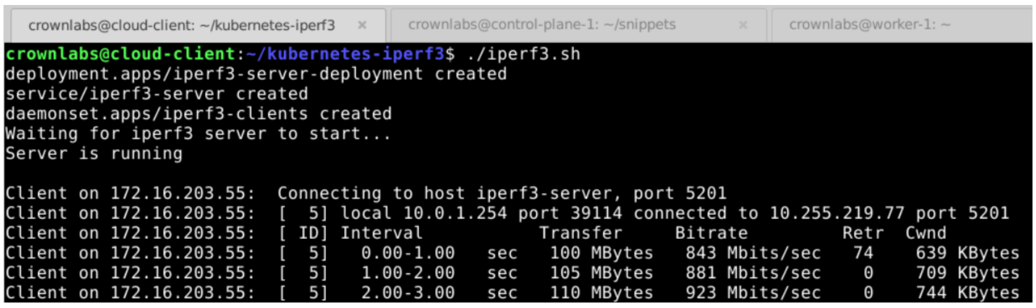
```
crownlabs@cloud-client:~/kubernetes-iperf3$ cat iperf3.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: iperf3-server-deployment
  labels:
    app: iperf3-server
spec:
  replicas: 1
```

Figure 17: Server Deployment snippet with `replicas: 1`

Q10 part 2: Attach a screenshot with the output of the experiment script. Comment the result and, in particular, discuss the reason why only one out of the three repetitions shows a higher bitrate.

A10 part 2: The experiment output (Fig. 18) shows throughput per client.

The variability in bitrate is typical in TCP-based tests due to slow start, retransmissions, and available bandwidth on each node.


crownlabs@cloud-client: ~/kubernetes-iperf3
crownlabs@control-plane-1: ~/snippets
crownlabs@worker-1: ~

```
crownlabs@cloud-client:~/kubernetes-iperf3$ ./iperf3.sh
deployment.apps/iperf3-server-deployment created
service/iperf3-server created
daemonset.apps/iperf3-clients created
Waiting for iperf3 server to start...
Server is running

Client on 172.16.203.55:  Connecting to host iperf3-server, port 5201
Client on 172.16.203.55:  [  5] local 10.0.1.254 port 39114 connected to 10.255.219.77 port 5201
Client on 172.16.203.55:  [ ID] Interval           Transfer     Bitrate         Retr  Cwnd
Client on 172.16.203.55:  [  5]   0.00-1.00   sec   100 MBytes   843 Mbits/sec   74    639 KBytes
Client on 172.16.203.55:  [  5]   1.00-2.00   sec   105 MBytes   881 Mbits/sec    0    709 KBytes
Client on 172.16.203.55:  [  5]   2.00-3.00   sec   110 MBytes   923 Mbits/sec    0    744 KBytes
```

Figure 18: iperf3.sh output: bitrate varies due to network dynamics