 FIT5225 Assignment 1 Report – iWebLens application

Full Name: Hao Xu

Student Number: 32767919

Tutor Name: Mohammad Goudarzi and Hasanul Ferdaus

Tutorial Time: 10:00 am – 12:00 pm, Monday

1. Abstract

This author conducted image objects detection experiments from local client and cloud client to the web-system, iWebLens. In this report, the result shows that the average response time will decrease by increasing the number of pods and threads without “OOMKilled” pods state. The author discussed 3 different challenges and the way to address them.

1. Result & Discussion

In this section, I will show the results of experiments conducted by local client and cloud client with different number of threads sent by client and different number of pods on the server-side limited on 0.5 CPU and 512 MiB memory. Also, I conducted an extra experiment with pods limited on 0.5 CPU.

1. Pods with 0.5 CPU and 512MiB memory

Figure 1: Plot of Local Client Test on Pods with 0.5 CPU and 512 MiB memory

Figure 2: Plot of Cloud Client Test on Pods with 0.5 CPU and 512 MiB memory

* 1. Normal behavior pods without “OOMKilled” state

Firstly, Let’s focus on pods more than 5 and threads less than 40, where the “OOMKilled” state does not occur. By looking at the plot of both local-client test, we can see several trends.

We can fix the number of threads to analysis the effects of different pods. When the client sends a single-thread request, the average response time slightly decrease with the increased number of pods, but the decrease quantity of time is little. For example, in local client test, the response time of 1 thread with multiple pods are all around 0.37 seconds. However, when the client sends a multiple-thread request, the average response time decrease a lot with the increased number of pods. For example, in cloud client test, when threads are 10, the average response time of 5 pods is 0.08, and it reduced to 0.05 when number of pods increasing to 15. That’s because Kubernetes will automatically balance tasks among pods in two worker nodes, to increase the speed of processing requests and decrease the resources pressure of each pod.

When the number of pods is fixed, we can conclude that the average response time will decrease when the number of threads is increased. For example, in Figure 2, when number of pods is 15 and the client send single-thread request, the average response time is around 0.2 second. And when the client sends 40 threads request at the same time, the average response time is decreased to 0.03 second. That’s because after Kubernetes balance tasks to different pods, these requests can be processed concurrently. Also, each pod is running a whole python flask application inside the container which is independent to each other, so that all pods could be regarded as an independent web service to process the request. This shows one of the key advantages to use Kubernetes to do container orchestration, which is to easily scale up our web-based application to handle more client requests.

* 1. Pods with “OOMKilled” state

However, we can see there are several odd points on both local client plot and cloud client plot. For example, by looking at the detail data of both local client and cloud client test, when there is only 1 pod but the local client send 40 threads request at the same time, the average response time is around 1.5 second which is really high. That is because on the web service, this single pod experiences several different lifecycle states, which are “Running”, “OOMKilled”, “CrashLoopBackOff” and back to “Running”. That’s because of resource starvation. When we deploy this single pod with our deployment YAML file, we manually set the memory request and limit to “512MiB”. So, when the client concurrently sends so many threads, say 20 threads, to the web service, so many local variables will be created in this pod’s container’s memory, even though I tried to reduce the number of local variables and use global variables as many as possible. If the pod’s container memory limit is reached, “OOMKilled” state will occur, and the pod may crash. If the pod is starting, crashing, starting again, and then crashing again, the state will become “CrashLoopBackOff”.

There are several ways to address this issue. The firstly, assign more resources to each pods so that the memory limit will be increased. The second way is to use less local variables and more global variables so that the memory limit will not be reached so quickly.

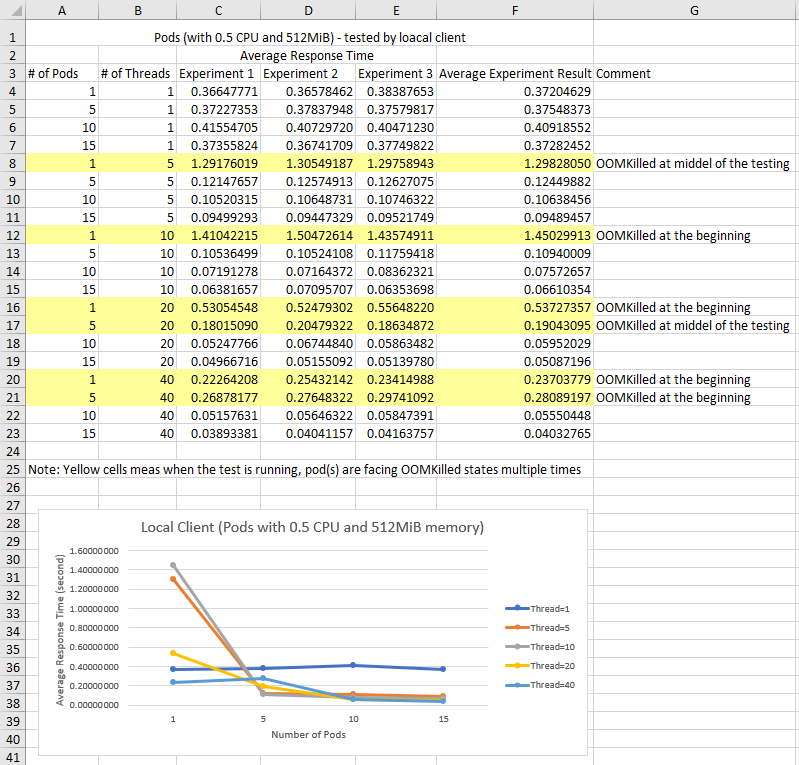


Figure 3: Detail Data of Local Client Test on Pods with 0.5 CPU and 512 MiB memory

Graphical user interface, application, table, Excel

Description automatically generated

Figure 4: Detail Data of Cloud Client Test on Pods with 0.5 CPU and 512 MiB memory

1. Pods with 0.5 CPU

After doing experiments above, I conducted extra experiments without assigning any memory resources limitations on each pod. Here are the detail data collected and plots for both local client testing and cloud client testing.

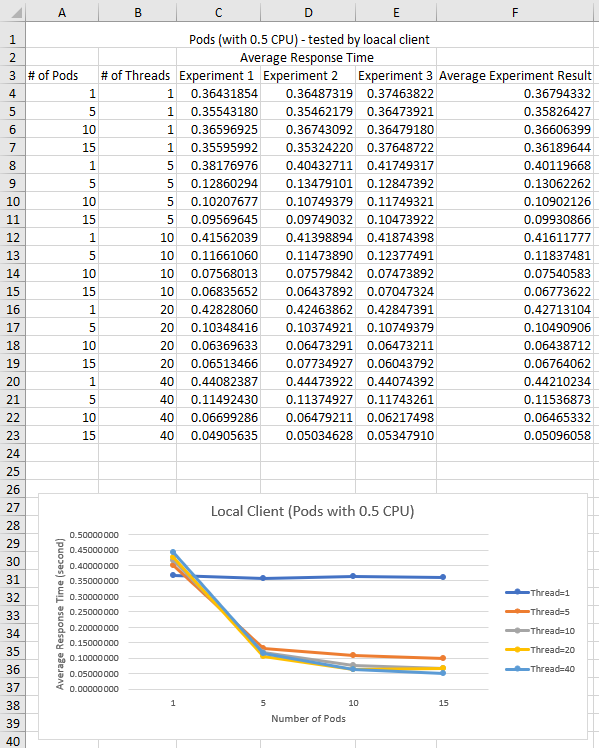
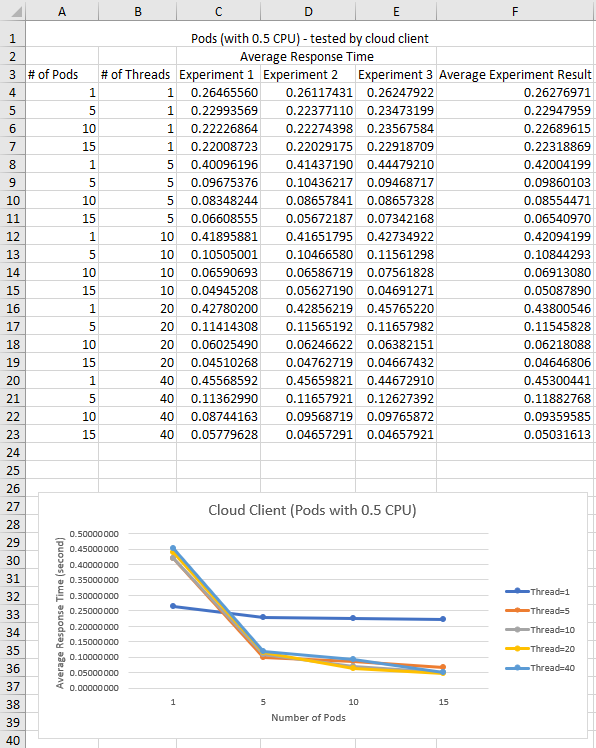
 

Figure 5:Data and Plots of Both Local Client Test and Cloud Client Test on Pods with 0.5 CPU

We can see the trends is similar to what we conclude above without those odd “OOMKilled” warning state points. The average response time decreases with the increased number of pods on fixed thread value. Also, the average response time decreases with the increased number of threads on fixed thread value. So, Kubernetes can balance tasks between two nodes and among multiple pods properly to increase the processing speed by handling the multi-thread client requests concurrently.

1. Challenges and Solutions

In this section, I will give examples of 3 different common challenges in cloud computing and how it is addressed in my system.

1. Scalability

Scalability is talking about that a system should be able to handle the growth of the number of users. In my system, for example, the number of uses to use this iWebLens app may be increased a lot at a particular time so that the pressure on my web service will be huge and the CPU or memory resource of 2 worker node VMs can be ran out. To address this issue, firstly we can scale up our system by creating more VMs playing as worker node role. After that, we can easily join these worker nodes to master node as what we did in the project, so that they can help to afford user’s pressure, since Kubernetes Cluster gives us an easy way to combine worker node into the master node. Secondly, we can scale out, meaning that, when we create our VMs, we can assign more CPUs and memory to a single VM so that it has more resources to be used by client users.

1. Concurrency

Concurrency means that multiple clients can access the same resources at the same time. In our system, for example, multiple users may send multiple images by multi-thread requests to our iWebLens app at the same time. To address this potential issue, I use container and multi-thread programming technology. Firstly, my python-based application used Flask package which can handle multi-thread client request. Only variables related to neural network part stay in local variables and other variables such as yolo-tiny-configs, yolo-tiny-label are global variables that can be shared by clients. Secondly, I encapsulate and package my application into a container by using Docker, so that Kubernetes can create multiple pods including a container where an application can independently run in it. So, when multiple users send requests to my system concurrently, Kubernetes will automatically assign these tasks to different pods to process them at the same time without slowing down the process time.

1. Failure Handling

Failure Handling meaning that the system is able to handle different types of failure, such as wrong data type sent by client, or failure of the web service. In my system, for example, client may send data which is not an image, or client send too many request to crash my web service. To address these kinds of issue, we can use detecting, tolerating and recovery methods. Firstly, my python app can detect the data type sent by clients and throw an exception if the data type is wrong. Secondly, if the webservice is down, we will show clients the information saying, “timeout of connection please try again later”. Finally, my Kubernetes application can automatically restart the pods if they meet some crash states such as “OOMKilled” and “CrashLoopBackOff”.

1. References

https://kube.academy/paths/kubernetes-core-concepts

<https://sysdig.com/blog/debug-kubernetes-crashloopbackoff/>

https://komodor.com/learn/how-to-fix-oomkilled-exit-code-137/