Lab No. 4: Scalability and Load Balancing

In this lab we will learn how load affects the quality of a service and how we can use load balancers to scale a service.

Part 1: Setup a Kubernetes Cluster

1. Provision a Kubernetes cluster by following the steps from Part 1 of Lab No. 3. (../lab03/lab03.html#lab03-part1). Name the virtual machine **lab04** and name your Kubernetes Cluster **lab04cluster** Make sure that you test the kubernetes cluster by deploying a test pod. (Any image is fine, as long as it lets you confirm that the Kubernetes cluster is running.

Part 2: Deploy a Simple Web Service

In this part of the lab you will deploy a very simple application that exposes a REST API with two endpoints:

- `POST /fibonacci` : takes a json payload with a single element `fibonacci_number` of type int. Example:

```
> curl -X POST -H 'Content-Type: application/json' http://<IP_ADDR>:8080/fibona
{"fibonacci_number":27,"value":196418}
```

- `GET /status` : returns internal state maintaned by the application (the size of a fibonacci number internal cache and a request counter)

```
> curl http://<IP_ADDR>:8080/status
{"cache_length":1001,"request_count":2847}
```

The application is hardcoded to listen in port 8080. The source of the application is the following:

```
package main

import (
        "encoding/json"
        "errors"
        "fmt"
        "io/ioutil"
        "log"
        "net/http"
)

type FibonacciRequest struct {
        FibonacciNumber int `json:"fibonacci_number"`
```

```
}

type FibonacciResponse struct {
        FibonacciNumber int    `json:"fibonacci_number"`
        Value           uint64 `json:"value"`
}

type StatusResponse struct {
        CacheLength  int `json:"cache_length"`
        RequestCount int `json:"request_count"`
}

var call_count = 0
var fibonacci_numbers = []uint64{0, 1}

func main() {
        http.HandleFunc("/status", StatusHandler)
        http.HandleFunc("/fibonacci", FibonacciHandler)
        log.Fatal(http.ListenAndServe(":8080", nil))
}

func StatusHandler(w http.ResponseWriter, r *http.Request) {
        if r.Method == http.MethodGet {
                sr := StatusResponse{len(fibonacci_numbers), call_count}
                jsonbytes, err := json.Marshal(sr)
                if err != nil {
                        w.WriteHeader(http.StatusInternalServerError)
                        w.Write([]byte("Unable to process request"))
                }
                w.Header().Set("Content-Type", "application/json")
                fmt.Fprintf(w, string(jsonbytes))
        } else {
                w.WriteHeader(http.StatusBadRequest)
                w.Write([]byte("Unsupported Method"))
        }
}

func FibonacciHandler(w http.ResponseWriter, r *http.Request) {
        if r.Method == http.MethodPost {
                body, err := ioutil.ReadAll(r.Body)
                if err != nil {
                        w.WriteHeader(http.StatusBadRequest)
                        w.Write([]byte("Unable to read request body."))
                } else {
                        call_count += 1
                        fr := FibonacciRequest{}
                        err = json.Unmarshal(body, &fr)
                        if err != nil {
                                w.WriteHeader(http.StatusBadRequest)
                                w.Write([]byte("Unable to unmarshal request JSON"))
                        } else {
                                if fr.FibonacciNumber > 92 {
                                        w.WriteHeader(http.StatusBadRequest)
                                        w.Write([]byte("Sorry, I can only get to F(9
```

```
                                } else {
                                        result, err := calc_fibonacci(fr.FibonacciNu
                                        if err != nil {
                                                w.WriteHeader(http.StatusBadRequest)
                                                w.Write([]byte(err.Error()))
                                        } else {
                                                fresp := FibonacciResponse{fr.Fibona
                                                jsonbytes, err := json.Marshal(fresp
                                                if err != nil {
                                                        w.WriteHeader(http.StatusInt
                                                        w.Write([]byte("Unable to pr
                                                }
                                                w.Header().Set("Content-Type", "appl
                                                fmt.Fprintf(w, string(jsonbytes))
                                        }
                                }
                        }
                }
        } else {
                w.WriteHeader(http.StatusBadRequest)
                w.Write([]byte("Unsupported Method."))
        }
}

func calc_fibonacci(number int) (uint64, error) {
        if number < 0 {
                return 0, errors.New("Negative, really?")
        }
        if number >= len(fibonacci_numbers) {
                result := uint64(0)
                for i := len(fibonacci_numbers); i <= number; i++ {
                        result = fibonacci_numbers[i-2] + fibonacci_numbers[i-1]
                        fibonacci_numbers = append(fibonacci_numbers, result)
                }
        }
        return fibonacci_numbers[number], nil
}
```

1. Create a docker image to run this application. Tag the image as `gcr.io/<YOUR_GCP_PROJECT>/fibonacci:latest` (replace your GCP project ID accordingly). Test your image locally, and once you have confirmed that it works, push the image to the Google Cloud Container Registry.
2. Create a Kubernetes Service that exposes the application using a **Load Balancer** type. (Same procedure as items 1 and 2 in Part 4 of Lab 3 (../lab03/lab03.html#lab03-part4)).
3. Wait until your service has been assigned an external IP Address. Once this is done, test your service before proceeding to the next section of the Lab. Do not scale your service since for next lab section, we need it to be running with only one replica.

Part 3: Measuring your instance performance

To measure the performance of our web service we are going to use *ApacheBench*, also known as `ab`, a popular benchmarking tool (https://httpd.apache.org/docs/2.4/programs/ab.html (https://httpd.apache.org/docs/2.4/programs/ab.html)).

1. SSH into the **lab04** instance. Run the following sequence of commands to install `ab`:

```
> sudo apt-get update
> sudo apt-get install -y apache2-utils build-essential
```

2. Since we are going to run our benchmarks with high concurrency levels, we need to increment the maximum number of open files that the kernel will allow. Run the following command (Note: this is not a persistent change, and you will need to run this command again if you log out or reboot **lab04**)

```
>> ulimit -n 25000
```

3. Create a file called `payload.json` with a single line with `{"fibonacci_number": 20}` as its content:

```
> echo '{"fibonacci_number": 20}' > payload.json
```

4. Run a test `ab` benchmark to verify installation (replace `<SERVICE_EXT_IP>` with the external IP address of the Kubernetes service that you deployed in the previous section):

```
>  ab -p payload.json -T application/json -m POST -n 100 -s 100 -r http://<SER\
```

   In the previous command, `ab` executed 100 requests (which is indicated by the `-n` argument) with a **concurrency level of 1** (which is the default value when the `-c` option is ommitted.).

5. In the `ab` benchmark that we run before, the `-c` option is used to simulate a given number of simultaneous connections (10 in that case), the `-n` option is used to determine the number of requests to issue(10 requests in the example). Experiment running `ab` with different concurrency levels, in batches of 10000 requests. Fill out the following table:

Concurrency	Requests	Requests per second (mean)	Longest request	98 percentile
50	10000			
100	10000			
500	10000			
1000	10000			
2000	10000			

Concurrency	Requests	Requests per second (mean)	Longest request	98 percentile
5000	20000			
10000	30000			
15000	45000			

Note

Apparently `ab` has a bug where if one of the requests times out, it exits with the message `apr_pollset_poll: The timeout specified has expired (70007)`, instead of recording it as a failed request. If you run into this problem try to run the test again.

What to turn in

Answer the following questions. Assume that you are not allowed to modify your current Kubernets infrastructure (i.e. you are not allowed to add more worker nodes to your Kubernetes cluster):

1. Using the previous observations, determine approximately at which concurrency level the performance of the fibonacci service begins to degrade.
2. Repeat the measurements made with `ab` when the service is scaled to 3, 6, 9 and 12 replicas. Save the data in a comma delimited file (csv) with the following columns: `replicas`, `concurrency`, `requests_per_second`, `longest_request`, `98_percentile`. You will need to upload this file during your Lab submission.
3. Did you get a noticeable improvement between 9 and 12 replicas? Explain this behavior.
4. What is the maximum throughput your service can provide? What is the maximum concurrency that you can support, while maintaining maximum throughput? How many replicas (approximately) do you need to support that concurrency level and throughput?
5. Suppose we expect our service to have a maximum number of 2000 concurrent users. We have a Service Level Agreement with our customers in which we guarantee a maximum response time up to 1.5 seconds, with a mean response of less than 300 milliseconds, with requests 98% of the time less than 600 ms. Can we fulfill those requirements? If so, how many replicas will be needed?

Back to top