Performance of Distributed Systems
2020-2021

Practical Assignment 3

Kris Foster

An assignment submitted in part fulfilment of the degree of

MSc. Advanced Software Engineering

Lecturer: Prof. Liam Murphy & Prof. John Murphy

UCD School of Computer Science
University College Dublin

July 11, 2021

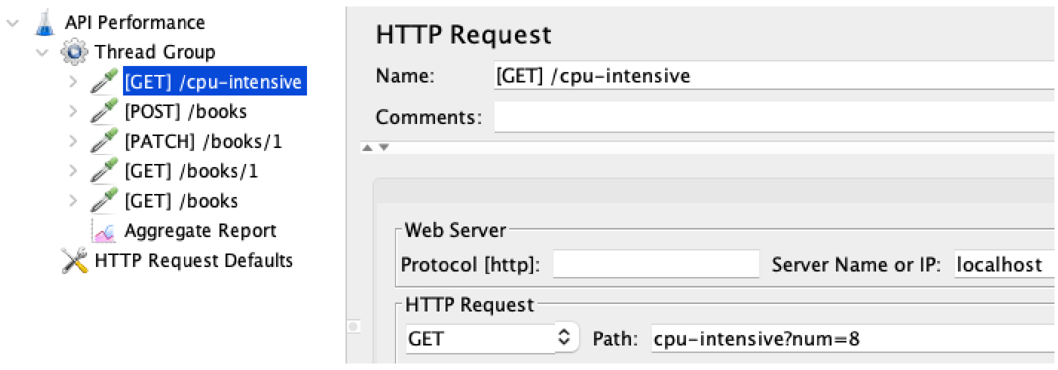
Introduction

For this assignment, I decided to use the Kotlin & NodeJS applications provided in the specification. I started by running the applications locally & getting familiar with the API endpoints they exposed. As discussed in the specification, the applications expose API endpoints to perform CRUD operations (create, read, update, delete) on a SQL database & another endpoint which performs a CPU intensive operation.

I wanted to deploy the applications on kubernetes (container-orchestration system) so I could easily scale up/down the application using kubernetes pods. A pod is a single instance of the application. A load balancer is used to route each request to one of the pods. As the applications would be running inside containers, I could also enforce a limit on how much CPU the application could use. I wanted to explore how these two variables (horizontal scaling & CPU limit) impacted performance.

Test Plan/JMeter Setup

I decided to create & run the test plan using Apache JMeter. I decided that the test plan should have 5 threads (users), which would all hit 5 endpoints (Figure 1.1). This would be repeated 20 times, for a total of 500 requests. I considered a request as failed if the response time was $>$ 1000ms. These failed requests & would be considered as 1000ms in the response time averages.



Name: [GET] /cpu-intensive

Comments:

Web Server

Protocol [http]:

Server Name or IP: localhost

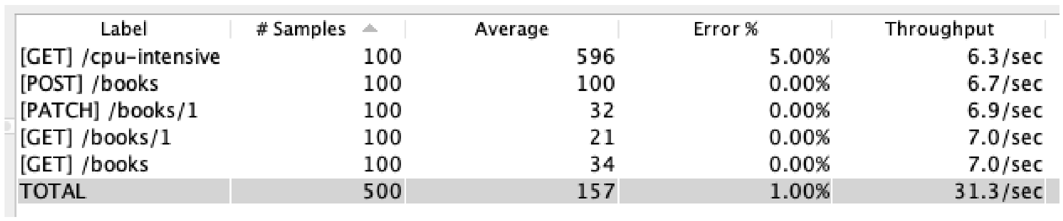
HTTP Request

GET

Path: cpu-intensive?num=8

Figure 1.1: Test Plan Setup

I added an aggregate report listener for each type of HTTP request (e.g. [POST] /books) & another listener which would report on all 500 requests in the test plan. The aggregate report shows useful information e.g. average response time, error % & throughput (Figure 1.2).



Label	# Samples	Average	Error %	Throughput
[GET] /cpu-intensive	100	596	5.00%	6.3/sec
[POST] /books	100	100	0.00%	6.7/sec
[PATCH] /books/1	100	32	0.00%	6.9/sec
[GET] /books/1	100	21	0.00%	7.0/sec
[GET] /books	100	34	0.00%	7.0/sec
TOTAL	500	157	1.00%	31.3/sec

Figure 1.2: Test Plan Report

Now that this test plan was created, I could deploy both applications with different configurations & run the test plan against them.

Deployment Setup

I wanted to deploy the applications on kubernetes so they could be easily horizontally scaled & other parameters could also be configured (e.g. CPU limit). I decided to add as much automation around the deployment so I could efficiently spin up applications with different configurations & run the test plan against them.

I started by creating two kubernetes deployment files (node-express-deployment.yml & kotlin-spring-deployment.yml). The code block below (Listing 1.1) shows an example of what these files looked like. The **replicas** key defines how many pods to deploy & the **cpu** key is used to enforce a CPU limit in the container. I also created two service.yml files (Listing 1.2), which would provide load balancing in-front of the replicas in the deployment.

```
apiVersion: apps/v1
kind: Deployment
spec:
  replicas: 1
  template:
      containers:
        - name: node-express
          image: kriscfoster/node-express
          resources:
            limits:
              cpu: "2000m"
...
```

Listing 1.1: node-express-deployment.yml

```
apiVersion: v1
kind: Service
spec:
  type: LoadBalancer
  ports:
    - port: 5000
      targetPort: 5000
  selector:
    app: node-express
...
```

Listing 1.2: node-express-service.yml

I then created a useful script (Listing 1.3) to deploy both applications, which would build a docker image & deploy it on kubernetes using the content from the deployment.yml file discussed above.

```
# build container
docker build ./node-express -t kriscfoster/node-express

# starting services
kubectl apply -f ./kubernetes/postgres-deployment.yml
kubectl apply -f ./kubernetes/postgres-service.yml
kubectl apply -f ./kubernetes/node-express-deployment.yml
kubectl apply -f ./kubernetes/node-express-service.yml
```

Listing 1.3: start-node-express.sh

All of the work carried out for automating the deployments with different configurations can be seen at `https://github.com/kriscfoster/PerformanceComparison`.

Results & Discussion

Applications Running in Single Pod

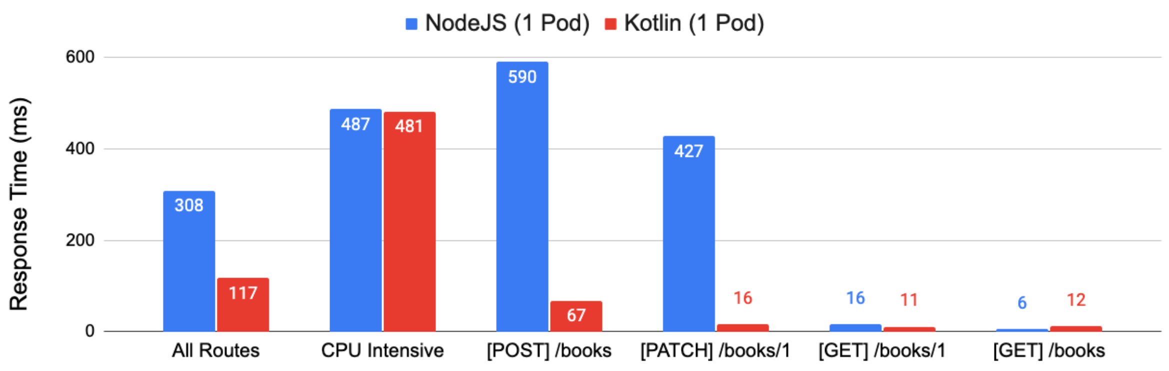


Figure 2.1: Results for single instance of both applications

I started by running the test plan against a single instance of both the NodeJS & the Kotlin application (results in Figure 2.1). 0.6% of requests to the NodeJS application failed (response took longer than 1000ms) & the Kotlin application responded to all requests within 1000ms. The average response time across all requests was 308ms for the NodeJS application & 117ms for the Kotlin application. The response times of the CPU intensive endpoint were actually very similar across both applications.

The results for the remaining endpoints were interesting. For the NodeJS application, whatever requests came after the CPU intensive requests also took a suspiciously long time. After further testing, my theory for this is that these following requests needed to wait for the CPU intensive requests to complete before they could be served. This is because NodeJS runs in a single thread. The same did not impact the Kotlin application as much because Kotlin/Spring uses multi-threading. This allowed following requests to be served by spawning different threads while the CPU intensive requests were still being served.

We could be able to solve this issue in the NodeJS application by using a library to introduce worker processes (e.g. throng - `https://www.npmjs.com/package/throng`). This would allow us to spawn multiple worker processes so more than one request could be handled concurrently.

The CPU intensive requests were definitely causing the performance issues in my test plan. I know this because when the CPU intensive requests were temporarily removed from the plan, all requests for all applications completed in < 20ms. It was nice that our applications included this CPU intensive endpoint, as it provided some data to compare the applications with. If we didn't have this endpoint, the results for all applications/configurations would have been very similar.

Scaling Applications Horizontally

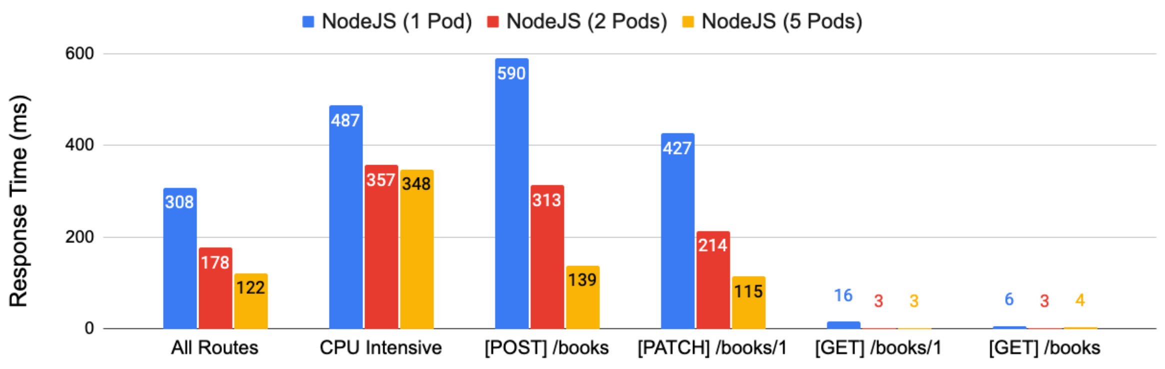


Figure 2.2: Results when NodeJS application was scaled to 2 & 5 pods

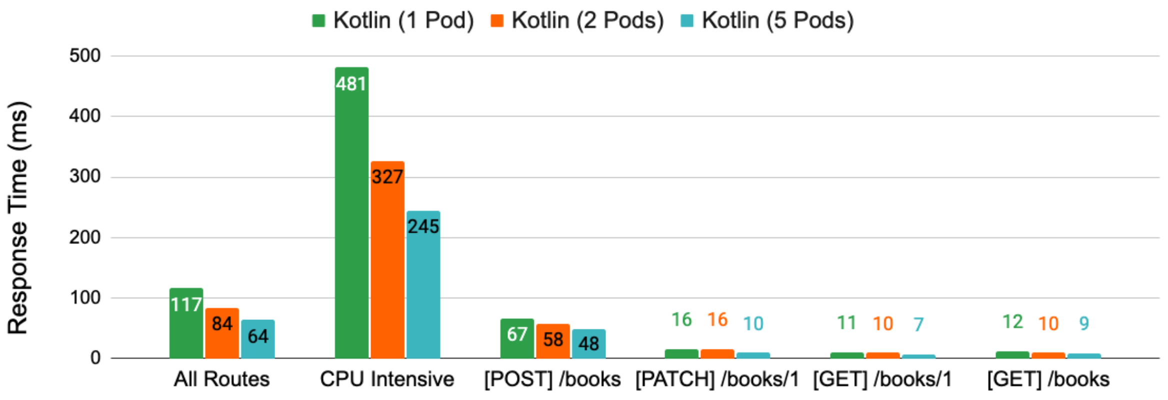


Figure 2.3: Results when Kotlin application was scaled to 2 & 5 pods

I then scaled the deployments of both applications up to two pods and again to five pods and ran the test plan against each of these configurations. As expected, this reduced the response times for both applications across all routes. None of the deployments using 2 or 5 pods had any errors as all response times were within the 1000ms response time threshold.

The average response time for the NodeJS application reduced from 308ms to 178ms when it was scaled from 1 to 2 pods. It reduced further to 122ms when scaled to 5 pods. This was because the load was being spread across multiple instances.

In addition to improving the performance of our system, having more than once instance would also make our application more resilient. For example, if one instance of the application failed & needed to restart, the remaining pods could handle the traffic during the downtime.

Enforcing a CPU Limit

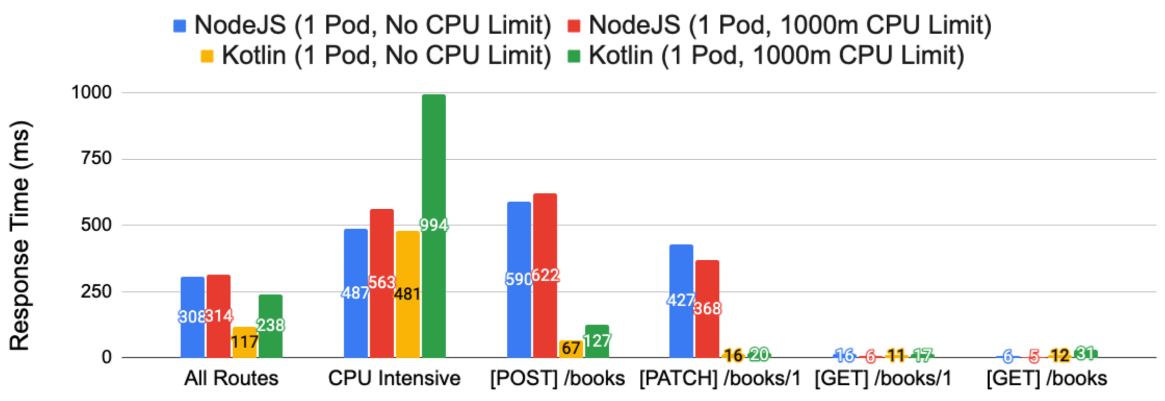


Figure 2.4: Results for both applications with CPU limit enforced

The results started to get interesting when I enforced a CPU limit (1000m/1 core) on the container running the application. In reality, this limit could allow us to run more pods (using lower CPU) on the kubernetes node. For example, if we had a 6000m (6 core) kubernetes node, we could either run three 2000m pods or six 1000m pods on the node.

This CPU limit had very little impact on the NodeJS application. The average response time increased marginally (from 308ms to 314ms). This showed that this was probably enough CPU for the NodeJS application under that load & adding more CPU wouldn't give us a large performance improvement.

However, enforcing this CPU limit had a huge impact on the performance of the Kotlin application (this showed that low CPU was causing a bottleneck there). The average response time for the CPU intensive route increased to 994ms & the error rate was 92% (these requests took > 1000ms). These performance issues were not seen on any of the other routes which showed that this particular endpoint was very CPU intensive. It also showed how the Kotlin/Spring application was utilizing multi-threading (a single slow request wasn't blocking the application from serving other requests).

These performance results showed me that the deployments of both of these applications would need to be configured in different ways to achieve their optimum performance. For example, NodeJS application would need to be deployed using more pods (with lower CPU). However, the Kotlin application would be deployed using less pods, but with each pod having higher a CPU limit.

Summary

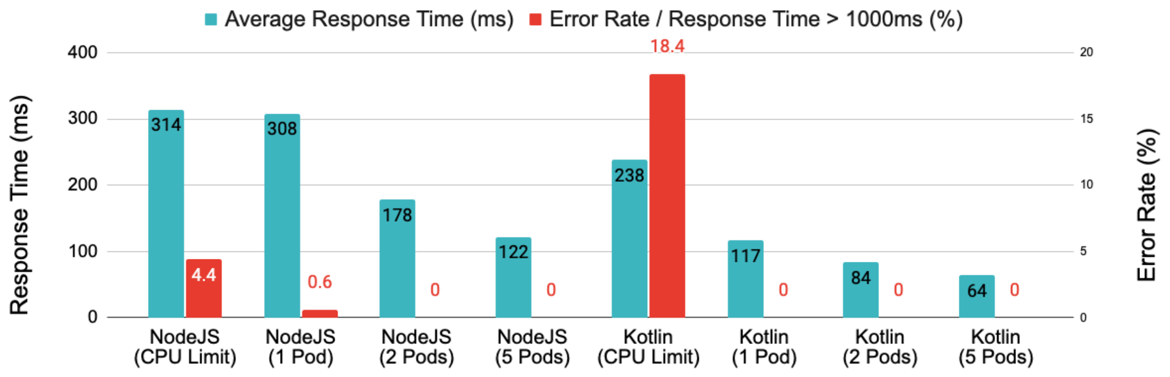


Figure 2.5: Summary of all results

Carrying out these performance tests was a very nice experience for me as I had never done these kinds of investigations on real applications before. I realise that we aren't really testing the performance of the language here, but are actually testing the performance of a lot of things around that, including the application frameworks (express with NodeJS, spring with Kotlin).

I liked that we were able to change different variables and see the impact these variables had on performance. These variables were the language framework of the application, how many instances of the application was running & how much CPU the application was able to consume.

It was very satisfying to see the performance improve as we scaled the applications up horizontally & this could be done very easily & dynamically in a cloud computing environment.

I really enjoyed the experience of using JMeter to perform the load testing because this is not something I've needed to do before. I hope to use these skills in the future. I might do some further research into more modern performance testing tools. The user interface & experience is quite dated. I also imagine that newer tools might also include some more advanced features. However, for the purpose of this assignment, the tool worked very nicely & gave me the insight & results that I wanted.

Appendices

NodeJS/Express Results

Route	# Samples	Average (ms)	% Over 1000ms	Throughput
*	500	308	0.6%	15.9/sec
[GET] /cpu-intensive?num=8	100	487	1%	
[POST] /books	100	590	2%	
[PATCH] /books/1	100	427	0%	
[GET] /books/1	100	16	0%	
[GET] /books	100	6	0%	

Table 3.1: NodeJS/Express 1 pod (no cpu limit).

Route	# Samples	Average (ms)	% Over 1000ms	Throughput
*	500	178	0%	26.1/sec
[GET] /cpu-intensive?num=8	100	357	0%	
[POST] /books	100	313	0%	
[PATCH] /books/1	100	214	0%	
[GET] /books/1	100	3	0%	
[GET] /books	100	3	0%	

Table 3.2: NodeJS/Express 2 pods (no cpu limit).

Route	# Samples	Average (ms)	% Over 1000ms	Throughput
*	500	122	0%	37.0/sec
[GET] /cpu-intensive?num=8	100	348	0%	
[POST] /books	100	139	0%	
[PATCH] /books/1	100	115	0%	
[GET] /books/1	100	3	0%	
[GET] /books	100	4	0%	

Table 3.3: NodeJS/Express 5 pods (no cpu limit).

Route	# Samples	Average (ms)	% Over 1000ms	Throughput
*	500	314	4.4%	15.6/sec
[GET] /cpu-intensive?num=8	100	563	10%	
[POST] /books	100	622	12%	
[PATCH] /books/1	100	368	0%	
[GET] /books/1	100	6	0%	
[GET] /books	100	5	0%	

Table 3.4: NodeJS/Express 1 pod (1000m/1 core CPU limit).

Kotlin/Spring Results

Route	# Samples	Average (ms)	% Over 1000ms	Throughput
*	500	117	0%	41.7/sec
[GET] /cpu-intensive?num=8	100	481	0%	
[POST] /books	100	67	0%	
[PATCH] /books/1	100	16	0%	
[GET] /books/1	100	11	0%	
[GET] /books	100	12	0%	

Table 3.5: Kotlin/Spring 1 pod (no cpu limit).

Route	# Samples	Average (ms)	% Over 1000ms	Throughput
*	500	84	0%	57.3/sec
[GET] /cpu-intensive?num=8	100	327	0%	
[POST] /books	100	58	0%	
[PATCH] /books/1	100	16	0%	
[GET] /books/1	100	10	0%	
[GET] /books	100	10	0%	

Table 3.6: Kotlin/Spring 2 pods (no cpu limit).

Route	# Samples	Average (ms)	% Over 1000ms	Throughput
*	500	64	0%	82.31/sec
[GET] /cpu-intensive?num=8	100	245	0%	
[POST] /books	100	48	0%	
[PATCH] /books/1	100	10	0%	
[GET] /books/1	100	7	0%	
[GET] /books	100	9	0%	

Table 3.7: Kotlin/Spring 5 pods (no cpu limit).

Route	# Samples	Average (ms)	% Over 1000ms	Throughput
*	500	238	18.4%	20.8/sec
[GET] /cpu-intensive?num=8	100	994	92%	
[POST] /books	100	127	0%	
[PATCH] /books/1	100	20	0%	
[GET] /books/1	100	17	0%	
[GET] /books	100	31	0%	

Table 3.8: Kotlin/Spring 1 pod (1000m/1 core CPU limit).