hochschule mannheim

Performance Comparison between Java and Node.js in terms of Scaling in Cloud

Zheng Zeng

Bachelor Thesis

for the acquisition of the academic degree Bachelor of Science (B.Sc.)

Course of Studies: Computer Science

Department of Computer Science

University of Applied Sciences Mannheim

15.02.2017

Tutors

Prof. Peter Knauber, Hochschule Mannheim

Jens Keller, SAP SE

Zeng, Zheng:
Performance Comparison between Java and Node.js in terms of Scaling in Cloud / Zheng Zeng. –
Bachelor Thesis, Mannheim: University of Applied Sciences Mannheim, 2017. 35 pages.

Zeng, Zheng:
Performance Comparison between Java and Node.js in terms of Scaling in Cloud / Zheng Zeng. –
Bachelor Thesis, Mannheim: University of Applied Sciences Mannheim, 2017. 35 pages.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, d. h. dass die Arbeit elektronisch gespeichert, in andere Formate konvertiert, auf den Servern der Hochschule Mannheim öffentlich zugänglich gemacht und über das Internet verbreitet werden darf.

Mannheim, 15.02.2017

Zheng Zeng

Abstract

Write English abstract

Contents

Chapter 1

Introduction

The promise of cloud has ushered the software engineering into a new era. Technology is to be used anytime and anywhere to untangle issues, provide solutions, bring people together and change their ways of living. This new era of net-centric web-based applications redefines how software is delivered to a customer and how the customer uses the delivered software. The digitization of economy impacts everyone. New businesses and leaders are emerging from nowhere. Companies appear and disappear much faster than ever before.

SAP, a company which has been renown for its on premise ERP solutions, also faces the pressure from customers to reduce Total Cost of Ownership (TCO) and increase agility. It has since long announced its cloud strategy to help customer master the digital economy. The reach of SAP systems is to be extended through cloud based access, ultimately reaching everybody everywhere in entirely new ways. Cloud computing allows on demand software provisioning with Zero-Installation and automatic configuration at low cost and immediate access in ultra-scalable data centers, which leads to the next generation networked solutions. In the process of leverage cloud infrastructure to increase the business agility and to lower the TCO of customers, ultimately new types of applications are enabled.

Cloud solutions come with new capabilities and technical challenges for cloud software development. Different cloud software development platforms and frameworks that applied with different programming languages are used inside SAP. JAVA EE and Node.JS are the most prominent ones. There have been on-going discussions about how to take a choice between them for SAP applications considering the SAP environment. The thesis aims to discuss this issue in terms of their respective performance and scalibility. Among all the major advantages brought by the cloud paradigm, scalability is the one that makes cloud computing different to an "advanced outsourcing" solution. It's no secret: high-performance website and web applications drive more traffic, engagement, and increase brand loyalty. In this age where customers are won and lost in a second, continual evaluation and optimization of web properties is essential. Performance affects customer satisfaction. Slow downs and/or downtime can cost companies thousands of dollars.

Write more about project introduction

Chapter 2

Related Work

Write Related Work

Chapter 3

Load Generating Tool

Performance and load testing has been conducted ever since applications came into been. There is a sea of different tools that can fulfill the task of generating virtual end users. In the case of this thesis, the server's capacity is going to be driven to its limit, after which instead of staying overloaded, it will try to scale and consume more clients. This requires the testing tool to be capable of generating a large enough amount of end users.

3.1 JMeter

One of the most popular testing tools is JMeter which is based on Java and cross-platform. It supports multiple protocol and has a very friendly UI to configure test plans. It even produces aggregated test results in different type of graphs. It seems JMeter has everything one asks for except scalibility. When the test requires a bigger scale, JMeter client machine quickly runs into issue that it is unable, performance-wise, to simulate enough users to stress the server or is limited at network level. An option to work around this performance limitation is to utilize remote/distributed testing [Apache, 2016]to control multiple remote JMeter engines from a single JMeter client. In this way, one can replicate a test across many low-end computers and thus simulate a larger load on the server. However, remote mode does use more resources than running the same number of non-GUI tests independently. If many server instances are used, the client JMeter can become overloaded, as can the client network connection. Another inconvenience of using JMeter is to find a ideal environment to host it. While one can execute the JMeter engine on the application server, one can not ignore the fact that this will be adding processing overhead on the application server, consequently the testing results will be somewhat tainted. All put together, it is decided not to use JMeter as load generating and testing tool in the thesis.

3.2 nGrinder

nGrinder [ngrinder, 2016] is a platform for stress tests that enables you to execute script creation, test execution, monitoring, and result report generator simultaneously. nGrinder consists of two major components. One is controller, a web application that enables the performance tester to create a test script and configure a test run. One is agent, a virtual user generator that creates loads. The performance team of Cloud Foundry in SAP has conducted their load test on router with nGrinder. The controller operates in a dedicated AWS c3.xlarge instance with 4 CPUs and 7.5 GB memory. Four agents are installed to create loads. Two of those agents are running on the host machine with controller and two others are distributed to another AWS c3.xlarge virtual machine. It manages to produce a load of 8500 requests per second to be handled by a router instance with 16 CPUs and 30 GB memory. nGrinder is considered too resource consuming and therefore not chosen in this thesis. Other comparatively more light-weight test frameworks like Locust also came into view but not decided for due to the complexity to scale the client.

3.3 hey

The Cloud Foundry Routing team has approached the load generating in a way similar to this thesis. They used a simple utility called *hey* [jbd@rakyll, 2016]. The tool is written in Go which accomplishes two tasks: generate a given number of requests and aggregate a simple report at the end of the test. It occupies next to nothing of the memory (less than 10m) and is intended to quickly smoke-test a web-application. Thus it is not a complex tool which can define a desired user behavior. It is actually quite fit for the use case of this thesis except that the original implementation of the tool persist no test results in database to facilitate its lightweight. To enhance the tool with desired database connection would be too time consuming because the existence of this tool comes into knowledge at a late state of thesis. At this point of time, another similar functioning tool in node is already maturely implemented and in use.

3.4 A self implemented scalable client

As mentioned above, most existing load generating tools requires comparatively powerful computing unit to host them. In this project, it is not realistic to host such a vigorous load generator because of the cost and complexity for extra virtual machine. Additionally, if client and server are running in the same cluster, unwelcome network overhead can be avoided in some extent. Although self implemented load generator is limited in its capacity, it can be made up through scaling the load generator instance. As figure 3.1 shows, enough load can be brought about by parallel running multiple instances.

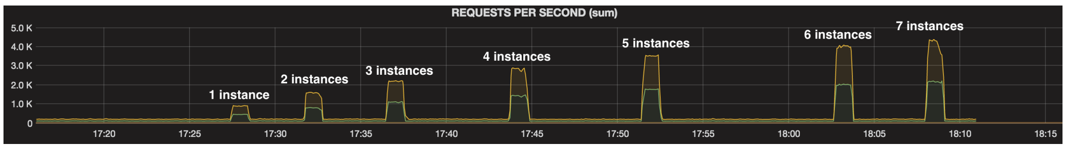


Figure 3.1: Load produced from different number of instances

Source: Grafana - Router Health

A simple load generator is implemented in this project as a node application. It sends requests with unique correlation id in the header to the server, records the response time and stores them in database at the end of load generating. The application is deployed to cloud foundry and uses PostgreSQL backing service. A timeout mechanism is added to make sure load is generated in the given time frame. Except for producing fancy graphic analysis results, it pretty much accomplishes every thing a load generating tool can do. It is more flexible to work with raw data anyway.

3.4.1 Features

The majority of the load testing frameworks choose to generate a fixed number of total requests with given amount of concurrencies. For example, a typical *JMeter* test plan configures parameter *Number of Threads* as concurrent virtual users. Through *Loop Count*, the same concurrency is repeated so to reach the total number of requests. *Hey* defines the total requests and concurrency degree then carry out the load generating accordingly. The same principle behind them is that the period of time during which the tests are carried out is not measured, only the response time of individual request. So many requests shall be sent either ended with successful transaction or landed in error because of timeout or other various reasons.
The load generator used in this project produces load in two ways. One implementation also generates the load with a fixed number of requests. Another implementation brings about parallel requests in a given fixed time frame, for example in one minute. The application will first pile up requests to a predefined parallel parameter. Whenever a request is successfully returned, a new one will be created to maintain the total parallel requests. Since the load testing is executed in an cloud environment where there are productive applications, define a relatively short time frame reduces influence on others. Therefore it is also used in the thesis most of the time. Figure 3.2 shows how load generator operates.
The proxy server limitation is discussed in 5.2.3. At the beginning the requests on

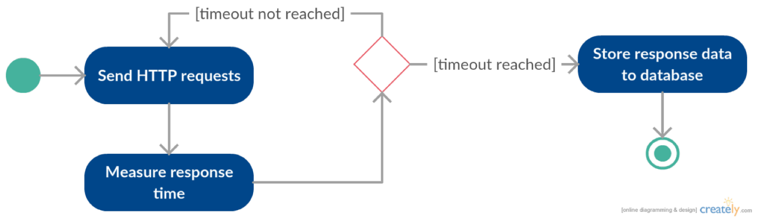


Figure 3.2: Activity diagram of load generator

the server was always brought about through establishing new connections. This has quickly hit the limit of what the HAProxy can handle. To workaround it, the started connections are kept alive and reused. Although HAProxy has a strange setting of cutting off the connection after a period of time, when the time expanse of test is not set to this limit, all things work perfectly.

3.4.2 Limitations

Since load generator is deployed as a worker in Cloud Foundry, it has little choice over database selection. In this project, a docker container version of database is used owing to the fact that other dedicated backing services are not free of charge. There are times container is simply overloaded and refuse to store any more data.
Secondly, load generator can be resource consuming as already discussed. In Cloud Foundry, some organization or space has only restricted size of memory to provide. It could be hard to scale the load generator to ideal capacity.In this thesis, for a period of time the load testing is conducted in a staging landscape of Cloud Foundry with a limited space quota of 10 G memory. Imagine all the applications and load generators scaling together, the space limit is easily exceeded.
Last but not the least, there is a chance the load generator is running on the same node as applications which will directly influence the CPU share the application could obtain and result in tainted test results. In the thesis, this unlucky scenario is avoided by deploying the load generator in DEA cells while the application is running in Diego cell.

Chapter 4

Measuring Tool

A critical part of performance testing is to ensure as few variables as possible change during each run of the experiment. This generates more consistent results, and allows easier comparison between test runs.
In a typical science experiment setup, the variables are separated into dependent, independent, and controlled variables. The dependent variables for this testing are of course the throughput and latency numbers that result from performance benchmarking, as well as the CPU usage of the applications during the test runs. For independent variables, the implementation of the application: in node.js or Java. The exact same tests are running against different implementations. Besides these variables, everything else is controlled and kept static.

4.1 Average Response time and Throughput

The Average Response Time takes into consideration every round trip.The resulting metric is a reflection of the speed of the web application being tested – the best indicator of how the server is performing from the users' perspective. The Average Response Time includes the delivery of all resource being used. Thus, the average will be significantly affected by any slow components. For example, geographic locations can have small impact on response times if the end user is thousands of miles away from the server.
In the thesis response time is measured in three different places as you can see from figure 4.1. The first measuring is done by load generator which records the true end-

to-end time and throughput. The second checkpoint is the response time retrieved from Logstash which present the round trip from router. Last measuring spot is inside server itself. It reflects the round trip of a query in database. The end-to-end measuring result is used in the final performance analysis while the others serve as tool to determine where is the bottleneck of the performance test is.

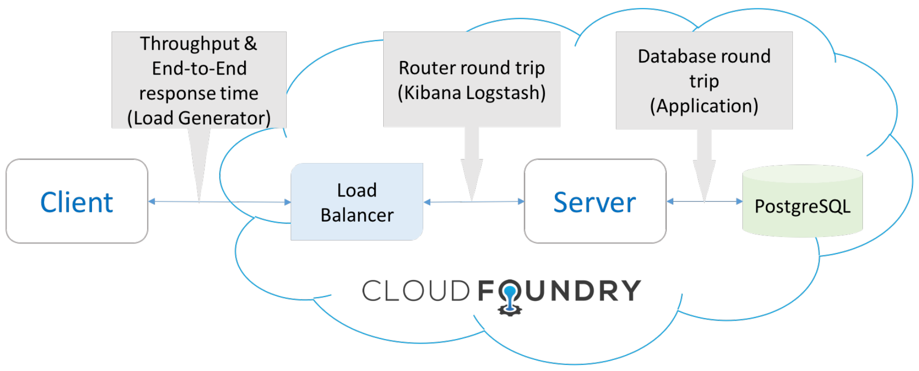

Throughput & End-to-End response time (Load Generator)
Router round trip (Kibana Logstash)
Database round trip (Application)
Client
Load Balancer
Server
PostgreSQL
CLOUD FOUNDRY

Figure 4.1: Measure response time three different places

Throughput is the measurement of bandwidth consumed during the test. It shows how much data is flowing back and forth from servers. The requests made to the server in the thesis do not consume data of any significant size, such as images. Therefore the throughput would directly reflect successfully handled amount of requests during the load testing.

4.1.1 Recording with Load Generator and its limitations

In chapter 3.4.1, it is discussed how the load generator drives loads. It carries another duty to record the end-to-end response time for each request. Together with response time it also stores the correlation id and start time for each request. The data is first saved in an array and insert into database at the end of load generating. However, this information is not enough. It would also be great to know the stored result is resulted from which test settings, like how many application instances or how many parallel requests are sent. Therefore these meta information is given to the load generator and saved along with response time. Since the load generator in this thesis is only a worker, it doesn't provide any API to call with given parameters. All the information external from the generator has to be given as start command. However, in cloud foundry, one application has only one start command. This means the generator should be pushed to the cloud foundry every time the parameter is changed. A small shell script is written to fulfill the task of pushing owing to the limited time frame of the thesis.

It is easy to scale load generator according to what one needs. On the other hand, it also means the generator has to stay stateless, which results in certain amount of manual work. For example, the same test settings may be carried out in several rounds. The result of different rounds should be saved to compare with each other. Making *testround* a automatically incremented data type is what first comes to mind. Yet it can't be defined like that when there are more than one instance of the generator running. This parameter is hence manually given for every new round of test.

4.1.2 Retrieving data from Logstash and its limitations

Application involves multiple components to accomplish one little task. Performance test can very easily hit limit of one of the components and converting the test of application into test of a router or database. On account of avoiding such cases, finding and ruling out the bottleneck of all the components are carried out after the application is implemented.
Network is one of the major causes of bottleneck. Is the router capable of taking in so many requests? The best way is to get the response time from router. Although router is not accessible as a component to the developers, Elasticsearch, Logstash, and Kibana (ELK) [Vaderzyden, 2015] is integrated into cloud foundry at SAP. Router response time is logged as default for every request. The logs produced in the application are saved in Logstash [1]and visualized in Kibana [elastic, 2017b].
However, in order to compare the router response time with end-to-end response time, just reading the results from visualization in Kibana is not enough. To investigate what causes an exceedingly long response time, one has to seek out the precise request and do the comparison. Kibana's visualization is pretty but more convenient would be directly comparing the figures from Logstash. To retrieve the results, a simple node application is written in the thesis. It sends a json query to the logstash

[1]see Logstash [elastic, 2017a]

and then parse the query results and sort out the response time logged by router. Following that the statistics are stored into a database.
Of course, the result from Logstash can be very large and takes some time. More strangely the logs appeared to be incomplete. Especially when the load is large. Only ten percent of the requests are retrieved from Logstash. After ruling out there is anything wrong with the application that retrieves the result, it is observed that the logs are always retrieved as bundles of 1000 requests which can be clearly seen from visualization in Kibana. After inquiring the colleagues responsible for ELK on Cloud Foundry, it turns out there is quota for logs. They don't come unlimited. One can only get a certain total of logs preserved per hour or per day.
In spite of incomplete information from Logstash, the comparison from the retrieved data shows the router has no performance bottleneck, at least not at the scale this thesis is aiming for. This leads to the realization of the HAProxy limitation discussed in chapter 5.2.3.

4.1.3 Recording data inside the application and its limitations

This would be a very simple task if there is no log limit in Logstash: just print out the response time recorded in application out and retrieve it from Logstash. With the restriction present, the data has to be saved in database. To elevate the performance of saving such large amount of entries, measures have to be taken: using batch in Java and manipulating array to realize a bulk insert in node.js. In this thesis, the response time and correlation id are saved in an array for all the requests. Another endpoint is defined inside the application to transfer the statistics in array into database.
Looking into the response time from database also helps to discover where the bottleneck of database starts. Up to a certain number of parallel requests from load generator, the throughput of the application no longer increases while the average response time goes up. Comparing the response time from database, there is an exact analog from peak of end-to-end response time to the peak in the response time from database.The slower requests also require more time to get database connection. This leads to the discovery that the database used in this thesis only provides 100 concurrent connections. After all the connections are used up, concurrent requests have to wait for a free connection to query into database.
Another thing to be taken into consideration is the memory leak which can be

caused by not transferring recorded data to database. As said above, the information obtained from the application help to identify the efficiency of database. After the determination of limitation, one no longer needs to analyze the data consequently forgets to call the endpoint and transfer the data to database. As a result, the array will accumulate, grow into a considerable size and cause memory leak. For example, correlation id is saved as 36-letter *string* which consumes 36 * 2 byte memory. Response time is saved as *long* which takes 16 byte. Putting them together, a request needs 88 byte memory to store the response information. One round of load test generates 20000 to 100000 requests. which will result in occupying up to 8 MB memory. A node application as a whole only needs maximum 100 MB to run. Therefore a switch should be built inside the application to turn off recording the data or there should be a central operational endpoint to clean up the array before each load testing.

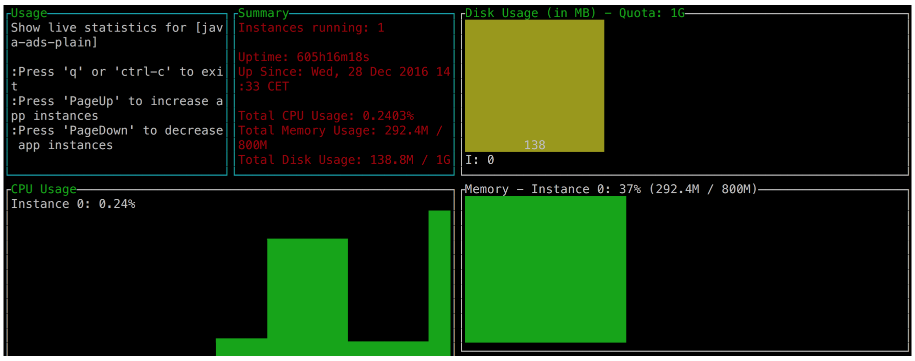
4.2 CPU and Memory consumption

In chapter 5.2.3, it is described how Cloud Foundry allocate its computing resource. In this thesis, tests are conducted on different days and sometimes it is noticed some out of ordinary fluctuations which can be accounted for by a reallocation of computing resource. Nevertheless, CPU and memory consumption are gathered as key metrics since they are directly related to todays cloud pricing mechanism.

4.2.1 Cloud Foundry CLI and its limitations

Since the thesis has no access to the direct virtual machine in Cloud Foundry, it can not investigate the node. However, the Cloud Foundry command line interface provides simple command *cf app <app-name>* to check the metrics. However, it is extremely annoying and inconvenient to require for the information from command line. Luckily there is a plugin, *cf statistics* [JamesClonk, 2017], which deliver real-time metrics and statistics data. This plugin displays a terminal dashboard showing current app usage metrics/statistics. It can also stream these metrics as JSON formatted output to stdout. It is decided to use the plugin in the thesis.

Although this tool is better than the command line, it still requires manual operation. The data gathered can not be stored into database along with other information.



```
Usage
Show live statistics for [jav
a-ads-plain]

:Press 'q' or 'ctrl-c' to exi
t
:Press 'PageUp' to increase a
pp instances
:Press 'PageDown' to decrease
 app instances

Summary
Instances running: 1

Uptime: 605h16m18s
Up Since: Wed, 28 Dec 2016 14
:33 CET

Total CPU Usage: 0.2403%
Total Memory Usage: 292.4M /
800M
Total Disk Usage: 138.8M / 1G

Disk Usage (in MB) - Quota: 1G
138
I: 0

CPU Usage
Instance 0: 0.24%

Memory - Instance 0: 37% (292.4M / 800M)
```

Figure 4.2: cf statistics display

After all what the plugin does is only request the API of Cloud Controller [Pivotal Software, 2017]from cloud foundry and gather them together. In the long run it would be advised to write one's own application which fetches the information from cloud controller's rest API.

4.2.2 Memory consumption

The memory has double meaning in cloud foundry. One can assign a certain amount of heap memory to the application which also decides the computing resource an application can get. However, the application doesn't necessarily consume that much. How much memory does an application need to run? It is known, at some point, when the heap reaches its maximum capacity, a full garbage collection will occur, which will bring down the size of the heap memory. Then it will start to grow again and the cycle should continue as long as the application is running. An application with no memory leaks should continue this cycle until the application is stopped.
For apps deployed on Cloud Foundry, one way to find out the native memory to run the application is to let it run under load until the first full garbage collection occurs. The total used memory of the container will continue to grow until this first garbage collection occurs. After that, the memory utilization of the container will stabilize and will not grow any more with sustained load. Figure 4.3 shows how the load test is conducted to find out how much memory the application needs to run.

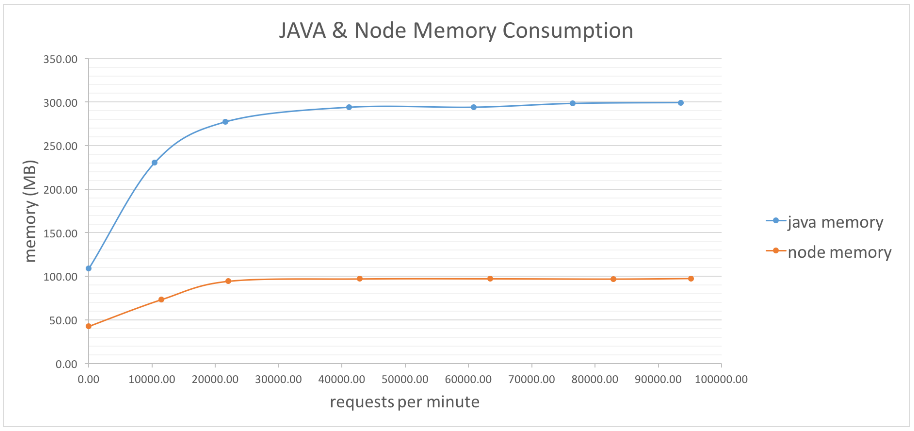


Figure 4.3: Find out the memory Java and node.js applications need

4.3 Access data from backing service

In the thesis, raw data instead of aggregated ones are collected and saved in backing service in cloud foundry. However, as all the other things in cloud, they are not that simple to access.
Toad extension for eclipse [2] is used in the thesis to connect to database and query data. It has no problem connecting to local PostgreSQL, as long as information such as hostname, user, password etc. are provided. Run the command *cf env <application-name>* in Cloud Foundry, one can find a detailed information regarding backing service. For PostgreSQL, exactly the information one needs to establish connection locally are all listed. It seems to be only straightforward to get into the database. However, TCP connection to the postgresql back-end service in Cloud Foundry is not allowed.
Cloud Foundry has provided a solution. It has enabled SSH access to the services in Diego cell [Cloud Foundry Foundation, 2017]. However, if one is behind a co-operate proxy, for example, SAP proxy, the SSH connection cannot be established. Finally, Chisel [Morikawa, 2017] comes to rescue. It is an HTTP client and server which acts as a TCP proxy, written in Go. It is an application to be deployed to Cloud Foundry and binded to the target backing service. By starting the application, it maps TCP endpoits of backing services to local workstation and ready to be connected from the toad extension.

[2]see Toad extension for eclipse [The Eclipse Foundation, 2017b]

Chapter 5

Experimental Environments

The experimenting environment is of most importance for technological evaluation. It is also impossible to find a configuration which can do both technology justice. One can easily end up trapped by some technological limitation of a framework that one used or in the case of this thesis, confined by existing infrastructure provider. However, as the thesis is conducted under the company context, the limitations have to be accepted and taken into account.

5.1 Server in local machine

Locally a virtual box running in Ubuntu 14.04 with 4 CPUs and 12G memory is installed on host machine which is a Macbook Pro with 4 CPUs and 16G Memory. However, the application has to share the resource with database and load generator which is not at all a neat configuration.

5.2 Server running environment - Cloud Foundry

Cloud Foundry is an open source software bundle that allows you to run a polyglot Cloud Computing Platform as a Service (PaaS). Initially it was developed as a Java PaaS for Amazon EC2 by Chris Richardson, in 2009 acquired by SpringSource, which was then acquired by VMWare, then handed over to Pivotal. The Cloud Foundry Foundation [Cloud Foundry Foundation, 2016a] is now the maintainer of Cloud Foundry. More than 50 companies are members of this foundation, such as

Pivotal, EMC, HP, IBM, Rackspace, SAP, or VMWare. The Cloud Foundry PaaS is a multi-component automation platform for application deployment. It provides a scalable runtime environment that can support most frameworks and languages that run on Linux. It also contains many components that simplify deployment and release of microservices applications (for instance, Router, Loggregator, Elastic Runtime (Droplet Execution Agents (DEA)), a message bus (NATS), Health Manager, Cloud Controller, etc.)

5.2.1 Cloud Foundry at SAP

Compared with other PaaS offerings, Cloud Foundry has some unique features: it has no limitation on language and framework support and is does not restrict deployment to a single cloud. It is an open source platform that one can deploy to run his apps on his own computing infrastructure, or deploy on an IaaS like Amazon Web Service (AWS), vSphere, or OpenStack. In SAP, it is first integrated with SAP Monsoon, then later shifted to Openstack and now hosted in AWS.

There are two different landscape of Cloud Foundry in SAP. One is called "AWS live". In this landscape, applications run inside containers managed by Warden [Cloud Foundry Foundation, 2016g] containerization (a visualization technique providing isolation on operating system level, which is more efficient than virtual machines). The Warden containers including the applications running inside are managed by DEA that also monitor the application health and provide the management interface to the Cloud Foundry platform.

The other landscape is called "AWS Canary". Instead of DEA, each application VM has a Diego Cell [Cloud Foundry Foundation, 2016e] that executes application start and stop actions locally, manages the VM's containers, and reports app status and other data to the Bulletin Board System (BBS) [Cloud Foundry Foundation, 2016f] and Loggregator. Instead of Warden, Garden [Cloud Foundry Foundation, 2016b] is used as the containers that Cloud Foundry uses to create and manage isolated environments. Diego architecture improves the overall operation and performance, for example supporting Docker containers.

5.2.2 Backing services , Service plans,and Scaling

Services such as database, authentication form the core of a PaaS offering. Cloud Foundry emphasizes on its flexibility in terms of backing services. It distinguishes managed services that obey the Cloud Foundry management APIs, and User-provided Services that serve as adapters to external services. In context of SAP, featured service like Hana database is offered along with other popular SQL and NoSQL databases. These services come with different plans, which define the size and capacity they are capable of offering. For example, PostgreSQL database has a service plan which is delivered in a docker container with predefined maximal number of connections. There are also other plans which provide PostgreSQL in dedicated VMs with different database sizes from extra small to extra large.
Service binding in Cloud Foundry is a simple way to tell the application to use a certain backing service. With frameworks like Spring cloud, database connector makes it extremely simple to enable the connection within application. In this thesis, as no extra frameworks is used, application parses the database information from the environment variable *VCAP_SERVICES* .

Since applications and its essential external components are loosely coupled in the context of Cloud Foundry Deploy, it leads to easy scalibility which consists the ultimate goals of cloud. In short, everything is a service: The application, framework functionality (such as persistency), and of course external services to use or integrate with. A service has its own data store, can be implemented and deployed independently, and communicates with other services using lightweight mechanisms. In this way, horizontal scaling of the system is supported. One can scale services up and down by adding or deleting service instances.

5.2.3 Limitations

As mentioned at the beginning of this chapter, it is impossible to have a perfect environment to conduct a evaluation of the performance of given technology. It is even harder in Cloud Foundry.

Limitation of CPU resource

The first difficulty is to ensure that both applications obtain the equal computing resource. Easiest way is to have JAVA and node applications running on the same VM. However, in Cloud Foundry, applications are not confined to a single cloud. There is no way to secure a dedicated VM on which only the given applications are running. So if one doesn't know if the same node is running the application, one tries to at least make sure both applications have the same CPU allocated. As mentioned above, Cloud Foundry uses containerization which provides operating-system-level virtualization instead of simulating bare computing hardware. This means multiple instances running in their respective containers share the same operating system. CPU shares instead of direct CPU time are first distributed. Each container is placed in its own cgroup. Cgroups make each container use a *fair share* of CPU relative to the other containers [Cloud Foundry Foundation, 2016d].This includes two layers of meanings. First, CPU shares are set aside for application according to the memory. Applications with larger amount of memory allocation receive more CPU shares. Does the applications have the same CPU resource when they occupy the same amount of memory? Not necessarily. The second step to designate CPU time is *relative to the other containers*. The following table 5.1 shows the CPU allocation for 3 applications running in the same VM.

Table 5.1: Example: allocate CPU time in Cloud Foundry

Application Name	Memory	CPU Shares	CPU Percentage	State
A	100 m	10	20%	running
B	200 m	20	40%	running
C	200 m	20	40%	running

Nevertheless, this schema of CPU allocation only shows when all the applications are running. If application A now start idling, B and C will both get 50% of the CPU resources. If only C is running, it gains whole 100% of the computing power. Now you see why securing identical CPU resource for two applications deployed in Cloud Foundry is so hard. Therefore, the thesis is carried out in a Cloud Foundry landscape called Canary Staging, where no heavy CPU consuming productive applications are running.

Limitation of database service

Cloud Foundry backing services are very convenient to use but is not free of charge when one need a bit more performance. In this thesis, the docker container version of PostgreSQL is used owing to the fact that all other dedicated services cost. The database used in the thesis has a limited connections of 100 per service instance. Luckily, with connection pooling one application barely needs more that 20 connections. Nevertheless, it does set a threshold on how many instances the application can scale. Large number of scaling is not discussed in the thesis therefore the database is still chosen as default for the performance testing.

Limitation of proxy server

Scalibilty certainly doesn't come for free. The minute system is under stress, the whole landscape of Cloud Foundry is facing challenge. During the thesis, a strange scenario has occurred: the total number of throughput in a minute lingers around 21,000 to 23,000 when the parallel requests reach a certain number. There must be a bottleneck somewhere. Is it because of how the application is implemented? Then the throughput should rise when more application instances are there to handle requests. However, no matter how many application instances are running to relieve the stress, the result stays the same. The CPU usage of the application doesn't go beyond 20%. The load is no more than a scratch on the application.
Is it because the database simply no longer has enough connection? To scale the database is somehow more complicated than scaling the application. Instead of adding one instance to the existing application, another identical application is deployed to Cloud Foundry with its own database instance. The same route is mapped to the new application. Load balancer distributes the requests from client between two applications. As there are two database instance now, the database capacity is doubled. However, this also changes nothing. To absolutely rule out the limitation is in database, the application is pushed to cloud without database at all as a simplest "Hello World" application. The limit of throughput persists.
Finally, after some searching into existing issues reported from other users, it turns out the bottleneck is in network. Cloud Foundry has a router written in Go, called gorouter [Cloud Foundry Foundation, 2016c] which does routing and load balancing. The router performs perfectly and can handle 5,000 requests per second without

a hiccup. Nonetheless, Gorouter is not the only thing standing between client and application. The requests are secured HTTPS calls to keep sensible information from hackers while routes are not encrypted during load balancing. This means before the request is routed to the destination, another server has to terminate SSL or TLS and change HTTPS calls to HTTP. This task is accomplished by High Availability Proxy (HAProxy) [Cloud Foundry Foundation, 2016h] by the time the thesis is finished. HAProxy is never meant to be put in a productive Cloud Foundry landscape owing to its incapability of automatic scaling. Every HAProxy can handle maximum 100 requests per second till its CPU consumption hits the celling. That is when it stops handling any more requests. In the current landscape, there are 4 HAProxy servers. Since they can not scale, it results in a network bottleneck where no more that 400 requests per second can be consumed. This is exactly why the strange scenario come into being. To overcome this restriction, either the proxy server should be changed into scalable ones, which is on the agenda of first quarter 2017 or keeping the connection to Gorouter alive and sending more requests in one decrypted connection. The latter choice is carried out in this thesis which will be mentioned in 3.4.1.

5.3 Client running environments

Where the client is running can influence greatly on the performance of the server. Do they share the same database? How great is the network latency? Do they share the same computing resource? To minimize all the influence to zero is next to impossible if the application is running in cloud. Different approaches are discussed in the following.

5.3.1 Client in local machine

Running server and client in the same physical machine kind of limited the network overhead. However, they consume the same computing resource and inevitably affected by other processes simultaneously operating in the same machine. One client instance won’t be enough to generate load to stress application. Regardless, the moment one tries to scale the total of client applications, then it is competing resource with the actual application. This leads to tainted test results.

5.3.2 Client in Cloud Foundry

Running the client in Cloud Foundry poses less network overhead for the request. Although requests still have to go through HAProxy to arrive at Gorouter despite the fact that client and server are under same domain or even running on the same node. Another convenience to run client on Cloud Foundry is that it is simple to scale. One client instance will have its limitation in sending parallel requests, but one can have as many instances as one needed. This is powerful and essential for this thesis.

5.4 Server and Client Configuration in the Thesis

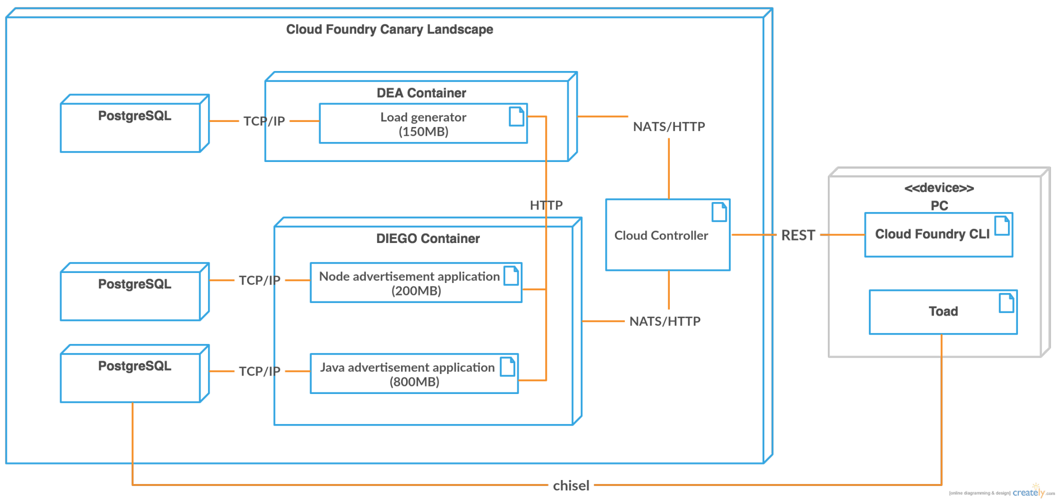


Figure 5.1: Deployment diagram for test environment

With all the matters considered, figure 5.1 shows the final setting of testing environment. First, both client and server are deployed in the same landscape in Cloud Foundry in order to minimize possible network overhead. Second, to avoid the scenario that load generator is running in the same node as the applications, the client and server are deployed into different containers. Clients are in DEA while applications are deployed to Diego. In addition to that, separate PostgreSQL database services are assigned to applications so that the load on the database is distributed. To retrieve data from database, chisel is used to bring TCP tunnel over HTTP, so that Toad extension on local machine can access the test results. Finally, through Cloud Foundy CLI, requests on application health is made and CPU and memory consumption information is gathered.

Another thing to pay attention to is how much memory the applications are allocated. For Java application, 800MB memory is set aside. As mentioned earlier in 4.3, it turns out Java needs about 300MB. However, since Java relies heavily on CPU shares allocated to it, more memory is alloted to ensure it has enough CPU even in a crowded node. On the contrary, Node.js application is given 200MB for the reason that it is single-threaded and more memory doesn't help it gain more computing resource. For the sake of not running into situation of not enough memory due to operational overhead in application such as garbage collector, memory is given quite generously to both applications.

Chapter 6

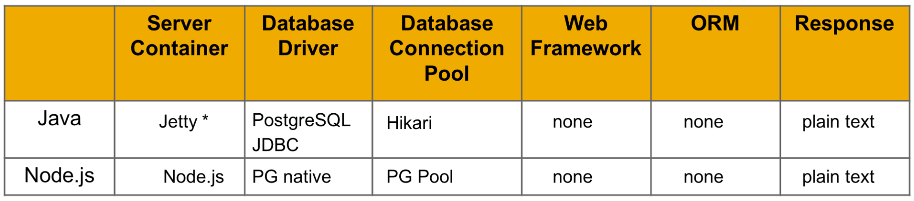
Implementation of Applications

Store the current order of product and its meta information, find the which shelf stores the current requested product, assign an idling logistic unit to pick up the itinerary ... I/O operations make up the majority of SAP business scenarios. In the load test conducted in this paper, applications are built to realize a scenario: advertisements are published in a bulletin board and clients can browse through the items.

The PostgreSQL backing service from Cloud Foundry is used as database. As the goal is to test how the application handles large amount of concurrency instead of the database efficiency, data will not be queried in high quantity or complicated joined actions in the test.

6.1 Implementation Configuration

To bring Java and node.js to a comparable level, the applications are implemented with minimum use of frameworks so that the overhead or any other influential factors on performance can be first taken off the table. As table 6.1 shows, the imple-



	Server Container	Database Driver	Database Connection Pool	Web Framework	ORM	Response
Java	Jetty *	PostgreSQL JDBC	Hikari	none	none	plain text
Node.js	Node.js	PG native	PG Pool	none	none	plain text

Figure 6.1: Implementation configuration

mentation both applications utilize no REST or any other kind of web framework.

No ORM is applied. Response is sent as plain text to the client to avoid overhead brought by JSON serialization.
Java application uses embedded Jetty server to handle plain HTTP requests. Node.js application used its embedded web platform. The connection with database is plain JDBC for Java while Node.js uses a popular PostgreSQL library: PG. Since the data structure is intentionally kept simple: only one table and with no complex data types. The application without ORM doesn't bring about a lot of boiler code.

6.2 optimize the Java implementation

All possible attempts are made to bring about every potential performance of the application. Since there is no complex logic, the focus of optimization lays on the interaction between applications and database. In case of Java implementations, to set a optimal thread pool configuration is also investigated.
The first checkpoint is database connection which greatly affect performance since it is the most expensive operation in the application without a complicated computing logic. Opening a connection and closing it with every request would gigantically slow down the application. Therefore connection pool is a key component in the implementation. It turns out there are quite a few libraries which handles connection pooling. In the thesis, three different libraries are tried out. *PGPoolingDataSource* [The PostgreSQL Global Development Group, 2017] comes with default PostgreSQL JDBC driver. *commons-dbcp2* [The Apache Software Foundation, 2017] package from Apache Software Foundation provides an opportunity to coordinate the efforts required to create and maintain an efficient, feature-rich package under the Apache Software Foundation (ASF) license. *HikariCP* is a "zero-overhead" production ready connection pool. It turns out *HikariCP* [Wooldridge, 2017] has outperformed the other two.
The next thing is to find an ideal configuration for the connection pool size. In an article from Brett Wooldridge [Wooldridge, 2017], it is pointed out larger connection pool size configuration doesn't necessarily lead to a better performance. Single core can only execute one thread at a time; then the OS switches contexts and that core executes code for another thread, and so on. It is a basic Law of Computing that given a single CPU resource, executing A and B sequentially will always be faster than executing A and B "simultaneously" through time-slicing. However, there are

a few other factors at play. For example, databases typically store data on a disk, which traditionally is comprised of spinning plates of metal with read/write heads mounted on a stepper-motor driven arm. So there is a time cost for disk "I/O wait". During this time that the OS could put that CPU resource to better use by executing some more code for another thread. So, because threads become blocked on I/O, more work can be done by having a number of connections/threads that is greater than the number of physical computing cores. In the case of thesis, database is run-

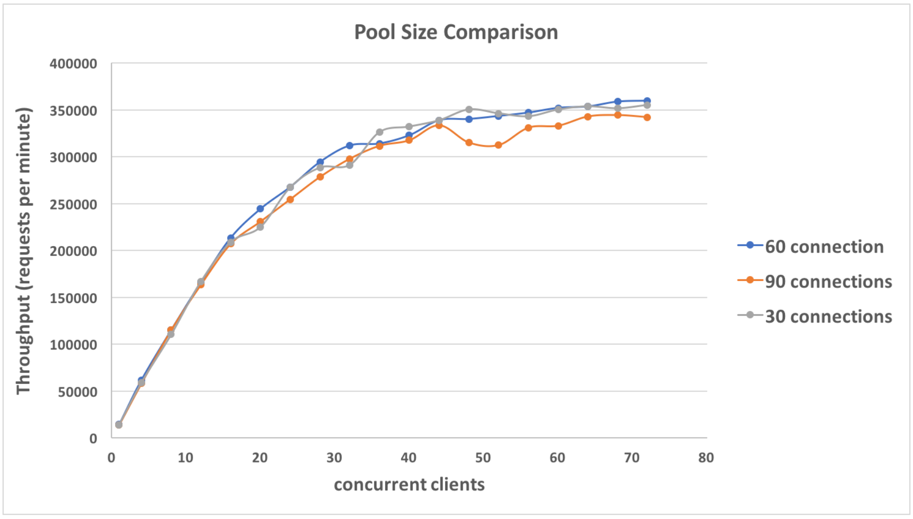


Figure 6.2: Database connection pool size comparison

ning in Diego cell which contains 4 CPUs. In order to verify and find the best fit for the load the thesis intend to generate, an experiment is conducted with connection pool size of 30, 60, and 90. Figure 6.2, shows a slight difference can be deducted that the pool size of 90 reaches the upper limit of total transaction slightly earlier than the other configurations. A little bit better is the performance from a connection pool size of 60 than that of 30. Then we compared the end-to-end response time with the data base response time. As figure 6.3 shows, when the response time peaks, the database response time stays stable and can hardly be considered responsible for the peak. Since the database pool size doesn't pose a conspicuous difference, we can draw the conclusion that the database is working at a high speed with no noticeable delay therefore the pool size setting is not a deciding bottleneck at all. It could be because the database is a docker container version and has no dedicated virtual machine. Thus it lies likely quite near the application and has little network overhead.

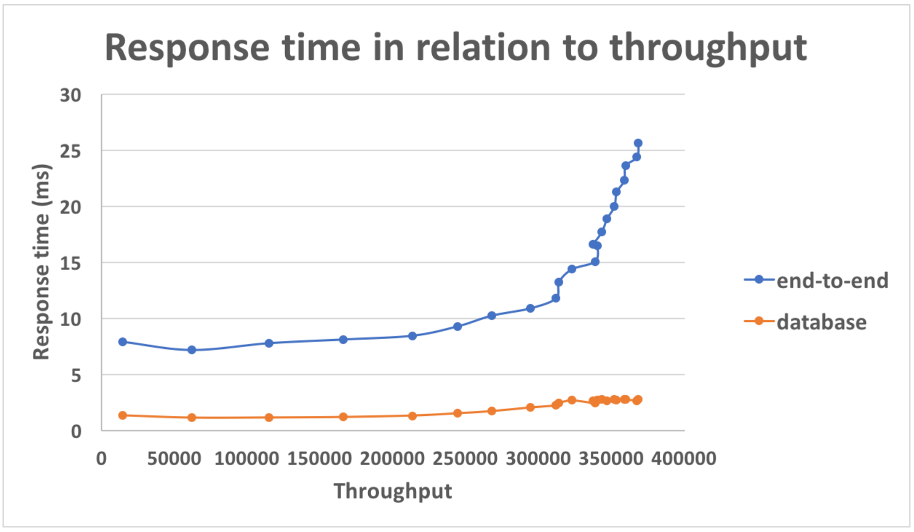


Figure 6.3: Comparing end-to-end response time with database response time

Thread pool size is also scrutinized and following the recommendation made by Jetty [The Eclipse Foundation, 2017a], the size is set from 10 to 400.

6.3 optimize the node.js implementation

Node.js application basically faces the same configuration of connection pool in database. In the thesis, also three different libraries are tried out. Unlike Java libraries, there is some overlapping in regard to the node libraries. In npm, one can find a number of PostgreSQL drivers. However, majority of them are built on the basis of one library: "node-postgres/pg" [C, 2017]. They are either wrappers or additional implementation with "promise" or "async/wait". For example, a great difference can not be derived from using of "node-postgres/pg" and "pg-promise" in the scenario in this thesis. The comprehensive research on framework benchmarking [TechEmpower, 2017] uses "Sequelize", which is also tried out in the thesis. However, it yields even a worse performance result because the object mapping costs indisputably computing time. In the end, the suggestion from "node-postgres/pg" writer is adopted to use "pg-native" which can boost a 20-30% increase in parsing speed.

Chapter 7

Test Strategy, Results and Analysis

7.1 Test strategy

At the beginning of the testing phase of the thesis, both applications are given the same amount of memory to ensure they have the same CPU share. It turns out, Java has nearly twice much throughput as the Node.js application. It seems one is comparing something which is totally not comparable at all. It is also quite far away from expectation. Things are tried and investigations are made to identify the cause.
The monitoring on CPU usage shows that the Java application can utilize up to 250%, in contrast which the Node.js application never quite exceeds the limit of 100%. This has everything to do with the single-threaded principle of Node.js applications. While Java application is through adding CPU and memory vertical scalable, Node.js is single-threaded and scales by creating multiple-node processes. It is only fair to test both application when they utilize the same amount of CPUs. However, the Cloud Foundry specific way of designating computing resources ensures there is no neat cut of a piece of CPU unless one is completely alone in the land scape. Therefore, multiple configuration of tests are conducted and described in this chapter so one can analyze the results from different aspects.

7.2 Test with optimal response time

In this test, fixed variable is average end-to-end response time which is kept under 10 ms as a criteria for optimal performance. An additional bar is set on the CPU consumption which can not exceed 100%.
In the context of Cloud Foundry, it is impossible to assure an application using only one core. Setting the memory of application can limit the CPU shares it can get. Nevertheless, if other applications on the same node are idling, even the tiniest application can get all the 4 CPUs of the node. In terms of Java application, it is already much more memory consuming than node as shown in 4.3 hence it is highly unlikely only one CPU is given to the application. What one witnesses as a 70% CPU is also very likely distributed in several CPUs.
Figure7.1 shows the result of the test described above. It can be deducted from the graph, that Java has a better performance in comparison with Node.js. As the CPU distribution is unclear, it can be accounted for that Java application is likely running with 4 or 2 very relaxed CPUs while Node.js is grabbing every bit of computing resource from that single CPU. This also leads to the reflection on the results of benchmarking from TechPower [TechEmpower, 2017]. In terms of applications on cloud, Java is twice more efficient than Node.js. It is possible they haven't taken the CPU distribution into consideration.

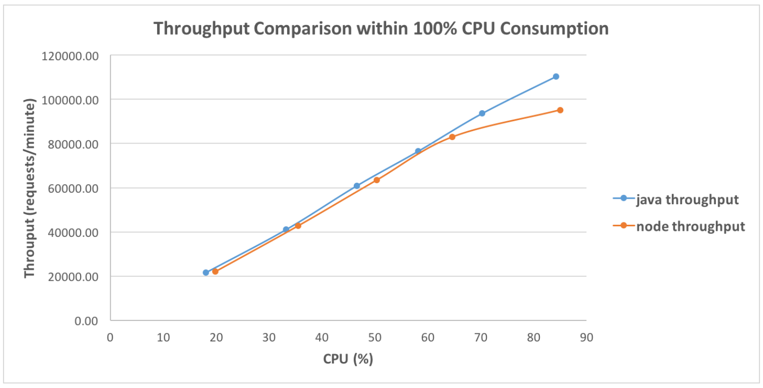


Figure 7.1: Compare throughput under 100 CPU consumption

7.3 Test with real load

7.3.1 Test configuration

In this round of test, applications are going to scale. Then how should it be contemplated whether two applications are equally scaled? As found out earlier, one Java instance utilizes more CPU while one Node.js employs only one, which leads the test to comparing vertical scaling in Java application with horizontal scaling in Node.js. This is still feasible if one can be sure Java always obtains the same CPUs. However, figure 7.2 shows, that Java application never get to use all of its 4 CPUs. The database connection and router load are checked and they are not under stress at all. An abnormal increase in memory is observed which indicates something suspicious in the application. The implementation is scrutinized, nevertheless, in the thesis no obvious cause is found. This leads to the realization that even though one Java instance gains 4 CPUs, it doesn't equal to four one-CPU Node instances because somehow not all 4 CPUs are put in use.

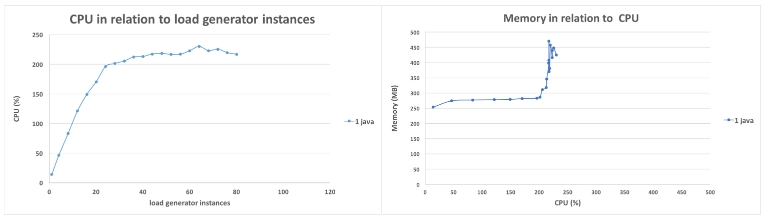


Figure 7.2: Limited CPU usage in Java application

Therefore the comparison in the thesis would not restricted to one Java instance against some multiple instances from Node.js. Load will be continuously added on to the application. Then the applications will scale when the response time shows a clear tendency to climb. At the end, an aggregated comparison will be made in terms of scalability in respect to CPU and instance.

Load is brought about through sending one request on application per instance of load generator.It is decided not to send parallel requests so as to guarantee each instance of the generator is simulating one end user. With each new round of test, 4 instances will be added to the existing running load generator. The maximum load generator instances will be 96 so that the limitation of database connection pool size will not be exceeded.

Java application will stop scaling when router has reached its limit (70% load). For Node.js applications, the scaling stops when the CPU usage reaches the maximum CPU usage of Java applications tested previously.

7.3.2 Test result

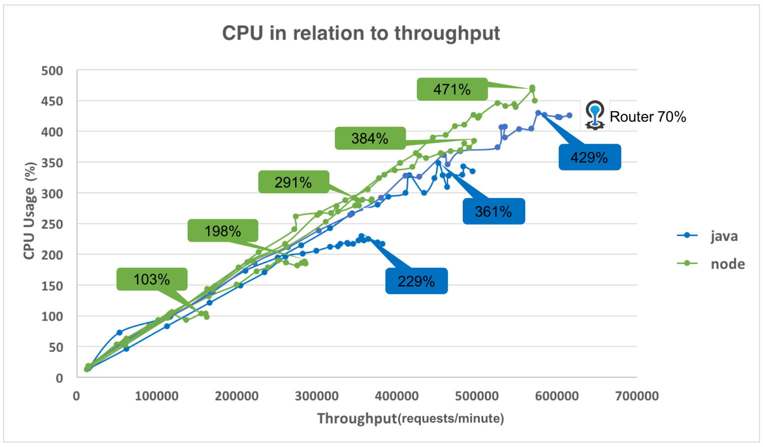


Figure 7.3: Throughput in relation to CPU

In figure 7.3, it is illustrated the throughput in relation to the CPU usage of Java application in running instance numbers of 1, 2, and 3. The xx axis is the throughput , the yy axis is the CPU consumption. When 3 Java applications are running, the router reaches its maximum capacity and becomes bottleneck because no more requests can be handled without losing time spent waiting for the load balancing in router. As summarized in table 7.1, with every increase of instance, the CPU usage from total available CPU decreases, which indicates the application doesn't scale horizontally by adding more instances.

Table 7.1: Percentage of utilized CPU

Application type	Instance Number	Available CPU %	Maximum CPU%	Utilzed CPU
Java	1	229	400	57%
Java	2	361	800	45%
Java	3	429	1200	35%
Node.js	1	103	100	103%
Node.js	2	198	200	99%
Node.js	3	291	300	97%
Node.js	4	384	400	96%
Node.js	5	471	500	94%

The same performance information of Node.js application is also depicted in figure 7.3. The test ends with 5 instances of Node.js application , which is when the CPU usage is in the same range of Java one. As summarized in table 7.1, application shows a quite steady utilization of more than 90% of the available CPU resources. It implies a horizontal scaling towards adding instances. If we draw a line down to the xx axis to see the correspondent throughput like 7.4, we can see the throughput scales also with adding instances. Since every instance make full use of the CPU, the throughput scales also linear per CPU.

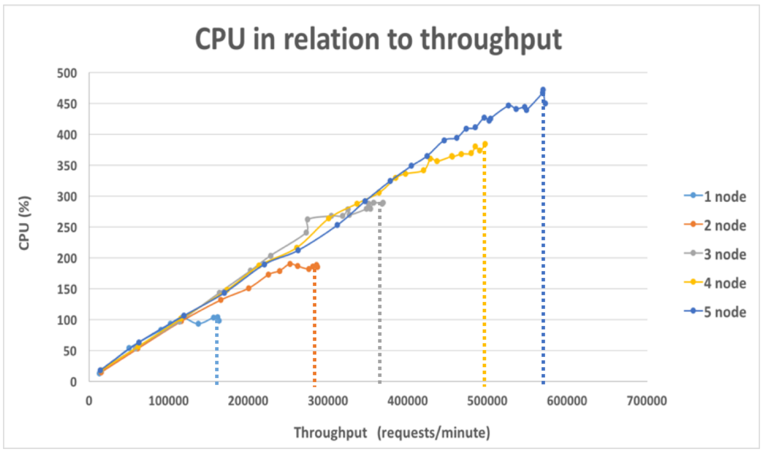


Figure 7.4: cpu efficiency of Node.js application

Although we observe an inefficient utilization of available CPU from Java application, we still want to know whether the applied computing resource is efficiently used. Figure 7.5 presents the same relation between CPU and throughput from a Java application with 3 running instances. Here we can have a look how the throughput scales when more CPU is given to a Java application. We can see the throughput scales almost linear with more CPUs.

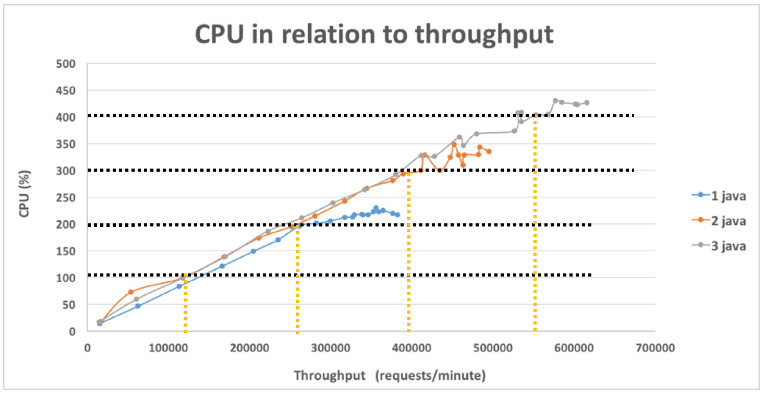


Figure 7.5: cpu efficiency of 3 Java instances

The last metric to check is memory consumption. As displayed in figure 7.6, Node.js shows excellent memory utilization for the whole test range while Java is quite memory demanding. Although it offers some performance gains in terms of handling more requests than the Node.js, it is not in proportion with the memory consumption.

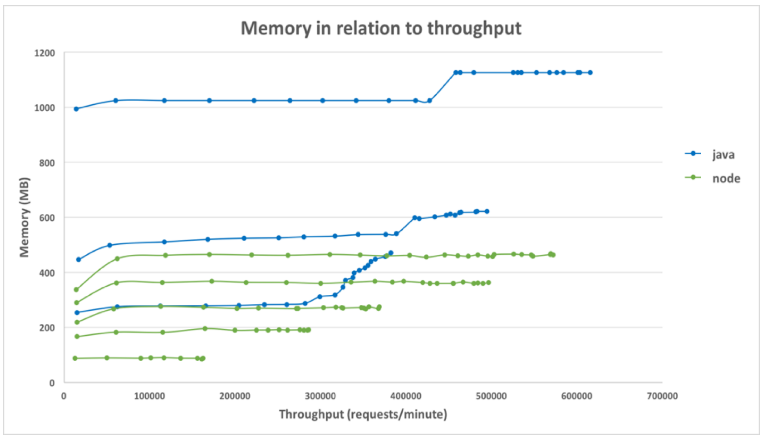


Figure 7.6: Memory in relation to throughput

However, another abnormality in the sudden surge in the case of one Java instance should acquire our attention. This might reveal some cause for the incapability of using all the CPU power assigned to the application. If the memory consumption fluctuation and inefficient CPU utilization can be resolved, Java would demonstrate a great vertical scalability. Although Node.js needs less memory for a single instance, the horizontal scaling requires the memory consumption to multiply itself. In the end, one Java instance which make full use of 12 CPUs would be much more memory efficient that 12 small Node.js instances.

Chapter 8

Conclusion

Write conclusion

Chapter 9

Future Work

Write Future Work

Abbreviations

DEA	Droplet Execution Agents
PaaS	Platform as a Service
AWS	Amazon Web Service
BBS	Bulletin Board System
ELK	Elasticsearch, Logstash, and Kibana
TCO	Total Cost of Ownership
ASF	Apache Software Foundation
HAProxy	High Availability Proxy

List of Tables

List of Figures

Bibliography

[Apache 2016] APACHE: *Apache JMeter - User's Manual: Remote (Distributed) Testing*. 2016. – URL http://jmeter.apache.org/usermanual/remote-test.html. – [Online; accessed 27-December-2016]

[C 2017] C, Brian: *node-postgres*. 2017. – URL https://github.com/brianc/node-postgres. – [Online; accessed 29-January-2017]

[Cloud Foundry Foundation 2016a] CLOUD FOUNDRY FOUNDATION: *Cloud Foundry*. 2016. – URL http://www.cloudfoundry.org/. – [Online; accessed 17-January-2017]

[Cloud Foundry Foundation 2016b] CLOUD FOUNDRY FOUNDATION: *Cloud Foundry Garden*. 2016. – URL https://github.com/cloudfoundry/garden. – [Online; accessed 17-January-2017]

[Cloud Foundry Foundation 2016c] CLOUD FOUNDRY FOUNDATION: *Cloud Foundry Garden*. 2016. – URL https://docs.cloudfoundry.org/concepts/architecture/router.html. – [Online; accessed 17-January-2017]

[Cloud Foundry Foundation 2016d] CLOUD FOUNDRY FOUNDATION: *CPU | Warden | Cloud Foundry Docs*. 2016. – URL https://docs.cloudfoundry.org/concepts/architecture/warden.html#cpu. – [Online; accessed 29-December-2016]

[Cloud Foundry Foundation 2016e] CLOUD FOUNDRY FOUNDATION: *Diego Architecture*. 2016. – URL https://docs.cloudfoundry.org/concepts/diego/diego-architecture.html. – [Online; accessed 17-January-2017]

[Cloud Foundry Foundation 2016f] CLOUD FOUNDRY FOUNDATION: *Diego Bulletin Board System*. 2016. – URL https://docs.cloudfoundry.org/concepts/diego/diego-architecture.html#bbs. – [Online; accessed 17-January-2017]

[Cloud Foundry Foundation 2016g] CLOUD FOUNDRY FOUNDATION: *Linux Implementation | Warden | Cloud Foundry Docs*. 2016. – URL https://docs.cloudfoundry.org/concepts/architecture/warden.html#linux. – [Online; accessed 17-January-2017]

[Cloud Foundry Foundation 2016h] CLOUD FOUNDRY FOUNDATION: *Scalable Components | High Availability in Cloud Foundry*. 2016. – URL https://docs.cloudfoundry.org/concepts/high-availability.html#processes. – [Online; accessed 17-January-2017]

[Cloud Foundry Foundation 2017] CLOUD FOUNDRY FOUNDATION: *Accessing service with SSH*. 2017. – URL https://docs.cloudfoundry.org/devguide/deploy-apps/ssh-services.html. – [Online; accessed 19-January-2017]

[elastic 2017a] ELASTIC: *Centralize, Transform Stash Your Data | Elastic*. July 2017. – URL https://www.elastic.co/products/logstash. – [Online; accessed 18-January-2017]

[elastic 2017b] ELASTIC: *Your Window into the Elastic Stack | Elastic*. July 2017. – URL https://www.elastic.co/products/kibana. – [Online; accessed 18-January-2017]

[JamesClonk 2017] JAMESCLONK: *Cloud Foundry CLI Statistics Plugin*. 2017. – URL https://github.com/swisscom/cf-statistics-plugin. – [Online; accessed 19-January-2017]

[jbd@rakyll 2016] JBD@RAKYLL: *hey*. 2016. – URL https://github.com/rakyll/hey. – [Online; accessed 04-January-2016]

[Morikawa 2017] MORIKAWA, Takeshi: *chisel*. 2017. – URL https://github.com/morikat/chisel. – [Online; accessed 19-January-2017]

[ngrinder 2016] NGRINDER: *nGrinder*. 2016. – URL https://github.com/naver/ngrinder. – [Online; accessed 02-January-2016]

[Pivotal Software 2017] PIVOTAL SOFTWARE: *Cloud Controller*. 2017. – URL https://docs.pivotal.io/pivotalcf/1-7/concepts/architecture/cloud-controller.html. – [Online; accessed 19-January-2017]

[TechEmpower 2017] TECHEMPOWER: *Web Framework Benchmarks*. 2017. – URL https://www.techempower.com/benchmarks/#section=data-r13&hw=ph&test=fortune. – [Online; accessed 29-January-2017]

[The Apache Software Foundation 2017] THE APACHE SOFTWARE FOUNDATION: *The DBCP Component*. 2017. – URL https://commons.apache.org/proper/commons-dbcp/. – [Online; accessed 25-January-2017]

[The Eclipse Foundation 2017a] THE ECLIPSE FOUNDATION: *High Load | Chapter 20. Optimizing Jetty*. 2017. – URL http://www.eclipse.org/jetty/documentation/current/high-load.html#_operating_system_tuning. – [Online; accessed 25-January-2017]

[The Eclipse Foundation 2017b] THE ECLIPSE FOUNDATION: *Toad Extension for Eclipse*. 2017. – URL https://marketplace.eclipse.org/content/toad-extension-eclipse. – [Online; accessed 19-January-2017]

[The PostgreSQL Global Development Group 2017] THE POSTGRESQL GLOBAL DEVELOPMENT GROUP: *Applications DataSource*. 2017. – URL https://jdbc.postgresql.org/documentation/94/ds-ds.html. – [Online; accessed 25-January-2017]

[Vaderzyden 2015] VADERZYDEN, John: *Welcome to the ELK stack: Elasticsearch, Logstash and Kibana*. July 2015. – URL https://qbox.io/blog/welcome-to-the-elk-stack-elasticsearch-logstash-kibana. – [Online; accessed 18-January-2017]

[Wooldridge 2017] WOOLDRIDGE, Brett: *HikariCP*. 2017. – URL https://github.com/brettwooldridge/HikariCP. – [Online; accessed 25-January-2017]

Todo list