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

11.11.2015

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, 2015. 20 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, 2015. 20 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, 11.11.2015

Zheng Zeng

Abstract

This project aims to evaluate the performance

Write English abstract

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

Einsatz eines Flux-Kompensators für Zeitreisen mit einer maximalen Höchstgeschwindigkeit von WARP 7

Write German 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 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. This paper aims to discuss this issue in terms of their respective performance and scalibility. The reason is simple. Among all the major

advantages brought by the cloud paradigm, scalability is the one that makes cloud computing different to an "advanced outsourcing" solution. Performance, which heavily affects customer satisfaction is a key metric.

Write more about project introduction

Chapter 2

Related Work

Write Related Work

Chapter 3

Technological Background

Performance and load testing has been conducted ever since applications came into been. There is a sea of different tools available in internet. In the case of this paper, the server's capacity is going to be driven to its limit, after which instead of staying overloaded, the server 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.

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 it 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 [1]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 JMeterEngine on the application server, one can not ignore the fact that this will be adding processing overhead on the application server 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 project.
nGrinder is a platform for stress tests that enables you to execute script creation,

[1]JMeter Remote Testing Apache [2016]

test execution, monitoring, and result report generator simultaneously [2]. 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 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 not applied in this project.
Other tools might be more light-weight, like Locust. It also supports distributed testing with master and slave.

Write about CPU Memory measuring tool

Write about other tools: Locust, Apache Bench, NGrinder, hey

[2] NGrinder Github ngrinder [2016]

Chapter 4

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 project, confined by existing infrastructure provider. However, as the project is conducted under the company context, the limitations have to be accepted and taken into account.

4.1 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. Host machine also inevitably runs other applications like outlook, skype office, which adds processing overhead and affect any performance potential of the application.

4.2 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 (http://www.cloudfoundry.org/) 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.)

4.2.1 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 containerization (a virtualization 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 that executes application start and stop actions locally, manages the VM's containers, and reports app status and other data to the **BBS!** (**BBS!**)and Loggregator. Instead of Warden, Garden 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.

4.2.2 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 project, 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.

4.2.3 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 [Foundation, 2016]. 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 shows the CPU allocation for 3 applications running in the same VM.

Table 4.1: 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 project 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 project, the docker container version of PostgreSQL is used owing to the fact that all other dedicated services cost. The database used in the project 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 this paper 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 project, 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.
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 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 dur-

ing 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 HA-Proxy by the time this paper is finished. HA-Proxy is never meant to be put in a productive Cloud Foundry landscape owing to its incapability of scaling. Every HA-Proxy 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 HA-Proxy 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 project which will be mentioned in **??**.

4.3 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.

4.3.1 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.

4.3.2 Client in Cloud Foundry

Running the client in Cloud Foundry poses less network overhead for the request. Although requests still have to go through HA-Proxy 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 in this project.

Chapter 5

Load Generating Tool

As mentioned in chapter 3, 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 from extra virtual machine to be installed. 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 5.1 shows, enough load can be brought about by parallel running multiple instances.

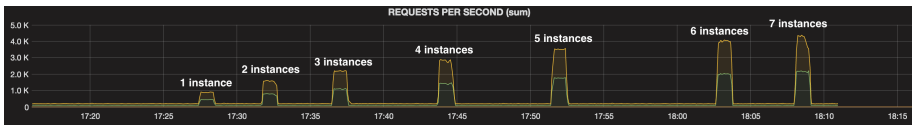
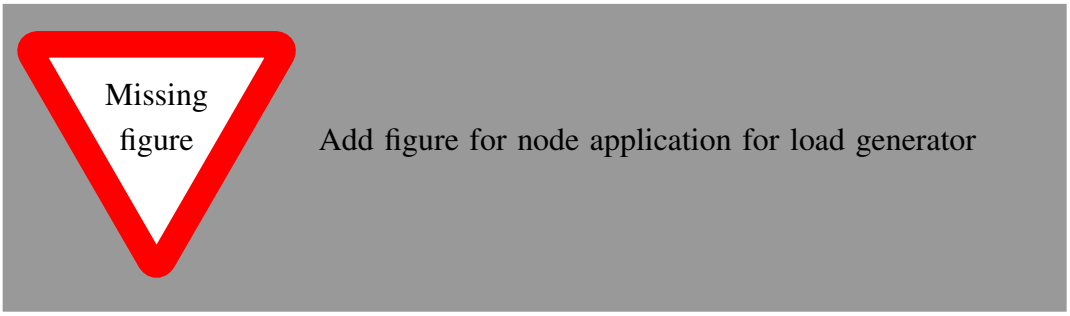


Figure 5.1: Load produced from different number of instances

5.1 Load generator - a self implemented scalable client

A simple load generator is implemented in this project as a node application. It sends requests with unique correlation id to the server, records the response time and stores them in database at the end of load generating. Application is deployed to cloud foundry and uses PostgreSQL backing service. A time-out 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.



5.1.1 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 under great stress and stops running.
Secondly, load generator can be resource consuming as already discussed. In Cloud Foundry, some organization or space has only so much memory to provide. It could be hard to scale the load generator to ideal capacity. For example, 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 resulting in tainted test results.

Chapter 6

Measuring Tool

6.1 Response time

Write why care about response time

6.1.1 Recording with Load Generator and its limitations

refer to previous chapter

6.1.2 Retrieving data from ELK and its limitations

Write ELK node application and log quota

6.2 CPU and Memory consumption

Write why care about cpu and memory

6.2.1 Cloud Foundry CLI and its limitations

Write: cf statistics

6.2.2 A self-implemented Ruby application and its limitations

Do I need it?

Chapter 7

Test Scenario

7.1 IO Intensive

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.

Write statistcs to support sap business scinario

In the I/O 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.

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 influence on performance can be first taken off the table.

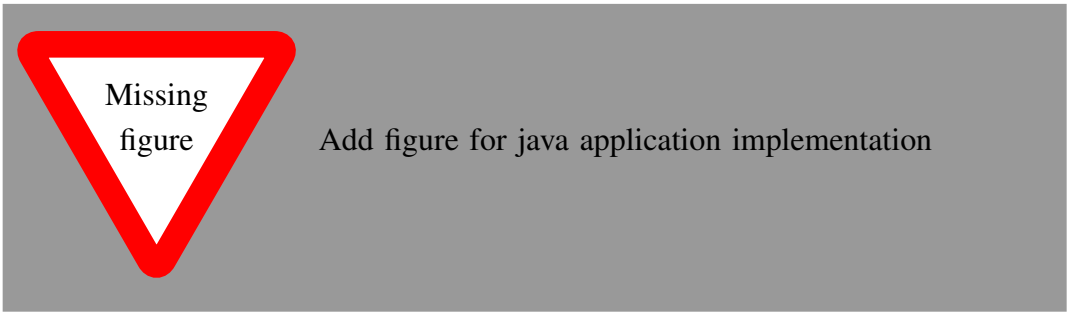


Figure X shows the implementation of the Java application. It uses embedded Jetty server to handle plain HTTP requests. No REST or any other kind of web services is implemented. The connection with database is plain JDBC. CRUD operations are implemented with query statements. No ORM is utilized. Since the data structure is intentionally kept simple: only one table and with no complex data types, the application without ORM does not produce significantly more boiler codes.

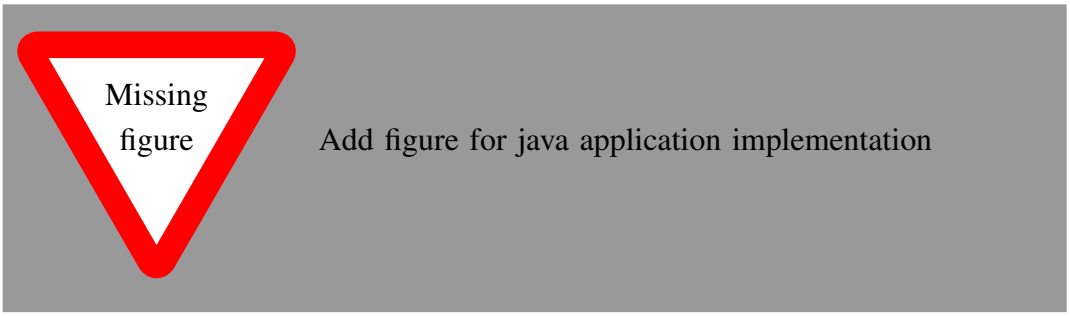


Figure X shows the implementation of the node application. It uses the standard node.js library to build the web platform. One other dependency it has is the database connection.

Another library both applications have used is the logging library. The logging library is configured so it can be filtered in elastic search. Because the convenience provided by the existing cloud foundry ELK platform, it is able to log information relevant to single requests and retrieve them through elastic search. This function is used later when comparing the measuring results and determining the bottleneck.

7.2 Compute Intensive

Do I still need it ?

Chapter 8

Test Results and Analysis

8.1 IO Intensive

Write results for local

Write results for CF

Write results for monsoon

8.2 Compute Intensive

Do I need it ?

Chapter 9

Conclusion

Write conclusion

Abbreviations

DEA Droplet Execution Agents

PaaS Platform as a Service

AWS Amazon Web Service

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]

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

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

Index

Todo list