Expertise
and insight
for the future

Minh Cao

Implementation and Performance Analysis of Replicated Load-balanced Services Pattern in Distributed Systems

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

2 April 2019

Author Title Number of Pages Date	Minh Cao Implementation and Performance Analysis of Load-balanced Replicated Services Pattern in Distributed Systems 35 pages + 0 appendices 2 April 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Janne Salonen, Head of Department (ICT)

Distributed systems of multiple machines have been on the way to become the de-factor architecture in digital services that have a big number of user population because not a single machine could provide such scalability. However, it might be challenging to design a distributed system that could live up to its expectation. With the rise of virtualisation and containerisation technologies, tools and methods have been introduced to help ease the challenge. As a result, design patterns have been gaining attractions as standardised architectural solutions. The goal of this project is to demonstrate an implementation of the load-balanced replicated services pattern and analyse its performance under several scenarios.

The approach taken to carry on this project is to develop a simple web service in Scala and deploy it under a distributed system environment with the help of Kubernetes technology as the backbone of the design pattern. Detailed implementation is documented in chronological order of the project process.

Under different configurations and settings of the pattern, it may have different performance. Therefore, the research involves inspecting different components and tuning them for highest performance. As the testing is done in a experimental environment where the use case is fixed, the data is fake and the users are simulated, several practical factors in real-life scenarios that may impact the success of the pattern are also discussed.

The result of the project is theoretical evidence why load-balanced replicated services pattern is considered to be high-performant and reliable as well as analytical data to help with tuning different configurations to achieve the best outcome.

Keywords	Distributed system, web server, load balancer, cache, Kubernetes, performance

Contents

List of Abbreviations

API	Application programming interface. A protocol or method to interact with an application or service.
CPU	Central processing unit. The processor that carries out a program's instructions by performing arithmetic, logic, controlling and input/output operations.
DAO	Data access object. An object that has access to database layer to do operations on data of an application.
DNS	Domain name system. A naming system for machines connected to the Internet. Typically consists of different information associated with a website name, most importantly its IP address and other metadata.
HTTP	Hypertext Transfer Protocol. An application protocol for information systems. It is the foundation for communication on the Internet between a web client and a web server.
ID	Identifier. A symbol that uniquely identifies an object or record.
IP	Internet Protocol. Numeral label that associates to a device to identify itself on a network. It is also the address of the machine to reach to.
I/O	Input/output. Input/output devices are hardware devices that enable a computer to send data (output) or receive data (input) from users.
JVM	Java virtual machine. A virtual machine that enables a machine to run any programs that are compiled to Java bytecode.
OpenJDK	Open Java Development Kit. A set of tools that enables the development of a JVM application.
OS	Operating system. System software that manages a machine's hardware and software as well as its memory and processes.
RAM	Random-access memory. A form of storage that almost takes same amount of time for any read or write operations.

SSL	Secure Sockets Layer. Standard security technology that enables an encryption link between clients and web servers.
URI	Uniform Resource Header. A string of characters that identifies a particular resource, typically on the Internet.
VM	Virtual machine. An emulation of a computer system. For example, a computer can run multiple operating systems by creating multiple virtual machines.

1. Introduction

The Internet has transformed drastically in the past 30 years. The demand for high-availability online services with quick response times has grown exponentially during the course of time. As a result, programs that were designed to run on a single machine have now been deployed to systems of multiple nodes to meet users' requirements. This has been the standard design in the industry to make sure services scale up by consuming resources provided by not one but many computers.

But building a distributed system that scales effectively is definitely not trivial. Fortunately, recent technological advances such as virtualisation and the rising popularity of container-orchestration software have significantly reduced the challenges in building a distributed system. The concept of containerised software pieces undoubtedly helps establish a collection of reusable components and design patterns, which are a toolkit that can be used in order to implement a reliable distributed software.

One of such design patterns is load-balanced replicated services. Burns (2018) asserts that it is the simplest and most wide-known among all [1, 45]. However, there are different configurations or settings that one could choose for their implementation to optimise the effectiveness of every component. In general, coming up with a high-level architecture does not mean that the job is done as there can be unforeseeable shortcomings. Performing analysis as well as studying and making sure each component is configured with the right settings is necessary to have a healthy and maintainable system.

This paper aims to present how to implement load-balanced replicated services pattern to achieve a secured, high-availability and performant distributed system. Apart from the background knowledge of characteristics of a distributed system and detailed implementation of the architecture, it also displays different performance analysis conducted and their results to provide some insights on how to bring the best out of the pattern.

2. Theoretical background

2.1. Distributed systems

2.1.1. Model

A distributed system consists of multiple individual computers that are physically distributed within geographical area. These machines communicate by passing messages between their processors inside a computer network. This is enabled due to the widespread of high speed broadband Internet connections. Each machine has its own task and interacts with others to achieve a common goal. [2, 53.]

Instead of running a program on a single computer, an application executed in distributed environment could make use of the resources and the availability of all the nodes in the network. According to Burns (2018), when structured properly, distributed systems are inherently more reliable [1, 2]. As each system handles a specific task, in case one or more tasks are identified as bottleneck(s) of the whole system, it should be easy to scale the relevant subsystems without the need of handling the entire system. Not only load scalability is an advantage of distributed systems, one can also benefit from administrative scalability as the number of different teams that share responsibility of a single distributed system is flexible and easy to scale, usually divided by business purposes or just simply a team per subsystem [2, 54].

Unfortunately, these advantages come at a cost. It is more demanding in terms of engineering skills to build a reliable distributed system as it is significantly more complicated to design, build and debug when comparing to a single-machine application [1, 2]. Usually this means more extensive infrastructure which leads to extra labor costs [2, 54].

2.1.2. Virtualisation and containerisation

Virtualisation is the technology that enables running multiple operating systems and applications (virtual computers) within the same machine. This means all the hardware components like CPU, OS, I/O devices, etc are emulated to numerous VMs. As a result, one can simplify infrastructure by consolidating applications onto fewer number of physical computers which leads to cost reduction and potentially utilising resources better as there are monitoring methods to help allocate resources efficiently among

VMs [3, 38]. However, this method has one drawback that additional resources are required from a machine to run multiple operating systems for its VMs.

This leads to the idea of containerisation which essentially is very similar to virtualisation excluding the overhead of separate operating systems. A container acts as a VM that operates on a separate runtime using the same host operating system kernel as described in Figure 1.

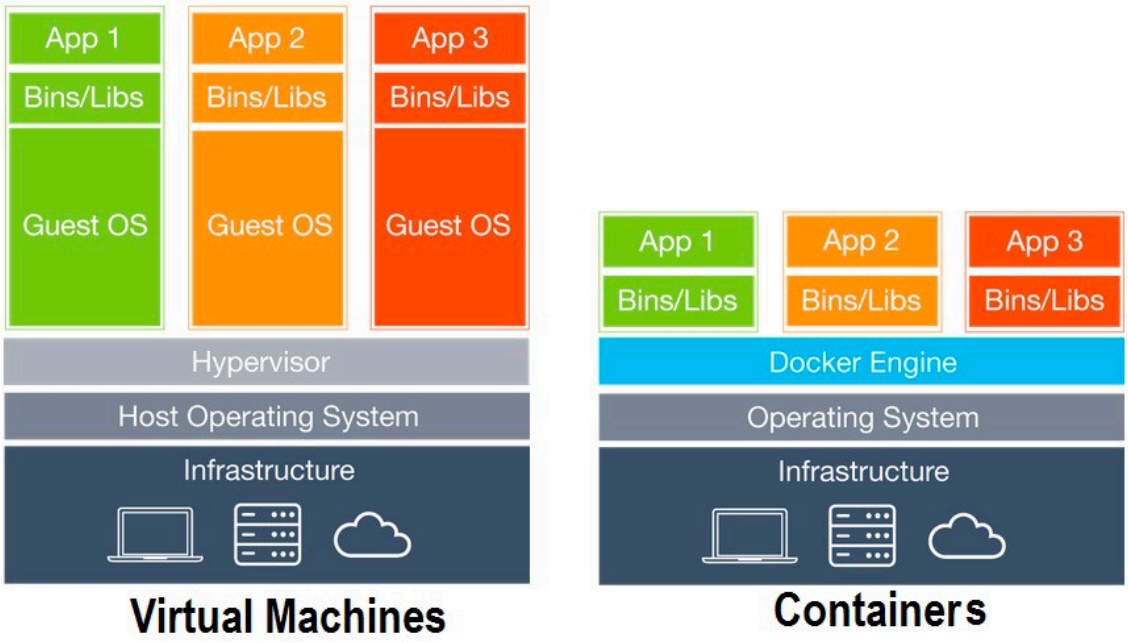


Figure 1: Comparison of virtualisation and containerisation

In order to manage containers easily, software acting as a orchestrator was developed and received much attention in recent years. It helps package applications into separate containers and automatically deploy them across a cluster of physical or virtual machines as well as provides auto-scaling, networking and availability managements.

2.2. Replicated load-balanced services pattern

A replicated load-balanced service consists of multiple identical instances that all are capable of handling users' requests. The reason for having duplicates of a server is twofold. Firstly, it helps achieve high availability as now the system could tolerate in case one or more, but not all, instances crash at the same time. Secondly, the system often increases in throughput since it could run parallel processes. In front of them, there is a component routing the traffic called load balancer. Thanks to the evolutions

of technologies discussed in previous sections, such a pattern can be implemented without significant complexity because each component can be viewed as a container.

2.2.1. Load balancer

Load balancer is the application that sits between client and server that helps distributing requests to one or more servers using a specified algorithm to make sure no server is heavily loaded while the others are being idle or doing little work. As a result, resources utilisation is optimised efficiently because all the nodes in the network are required to perform relatively equal amount of work. Furthermore, this affects user satisfaction directly as the request is given to an instance that has high likelihood of fulfilling that request faster. [4, 3376.]

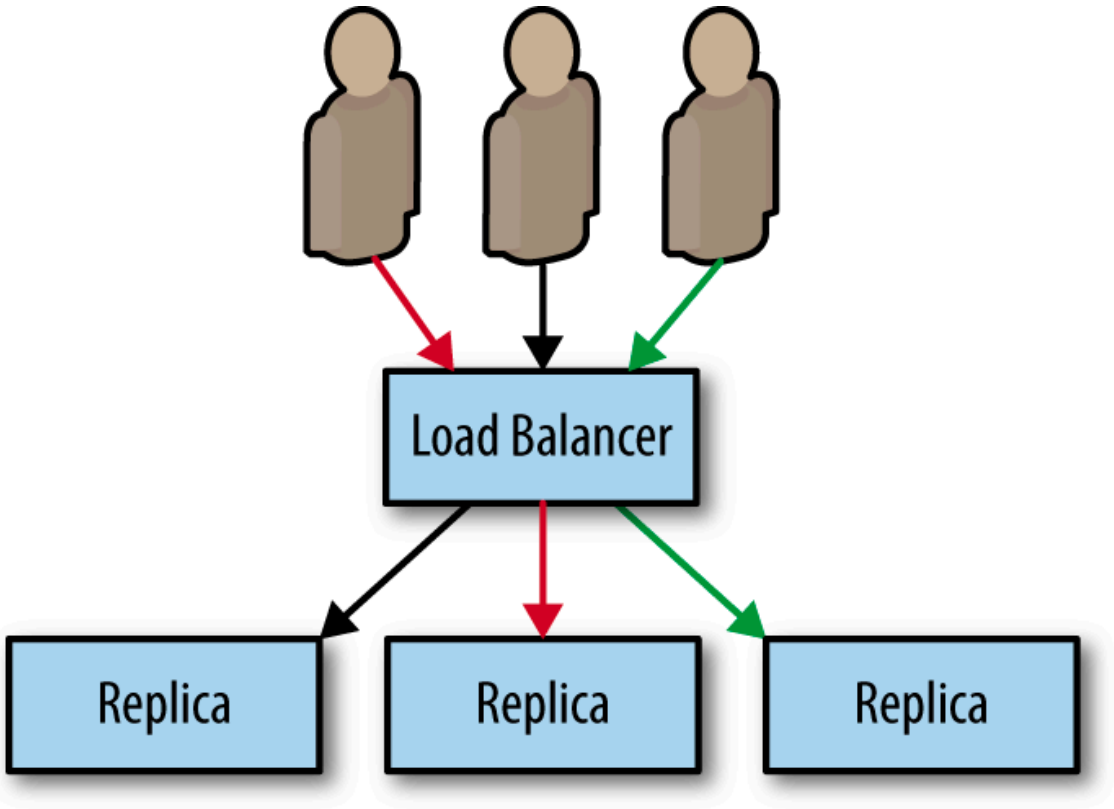


Figure 2: A load balancer that routes request to a single instance

Figure 2 demonstrates such architecture with a load balancer acting as traffic router. Whether a load balancer could make the right decision to route a request to the right server depends significantly on the algorithm it uses. There are two classifications of load balancing algorithms: static and dynamic. A static algorithm is considered as less complex as it does not require any information regarding the current state of the server [4, 3377]. For example, the round robin algorithm forwards a client request to each

server in turn. In contrary, a dynamic algorithm always yields the target server based on the current status of the system. For example, Ant Colony optimisation technique is an algorithm that every time a request in initiated, all the nodes are checked one by one to give out a result [4, 3377].

2.2.2. Caching layer

Often times the requests from users might still take much time even when the load balancer has given the task to the desired worker because of the internal processing happening inside the replication. In such a case, it is recommended to consider adding a caching layer, a component which stores the results of specific expensive requests [2, 49]. If two users conduct two requests asking for the same set of data, only the first one will go to the server then the result is stored in the cache to serve the other user. This is extremely helpful in scenarios that it is important to fulfil users' requests in a fast manner with little or no harm of possible outdated data. Figure 3 below shows the full design pattern with load balancer, caching layer and several application replicas.

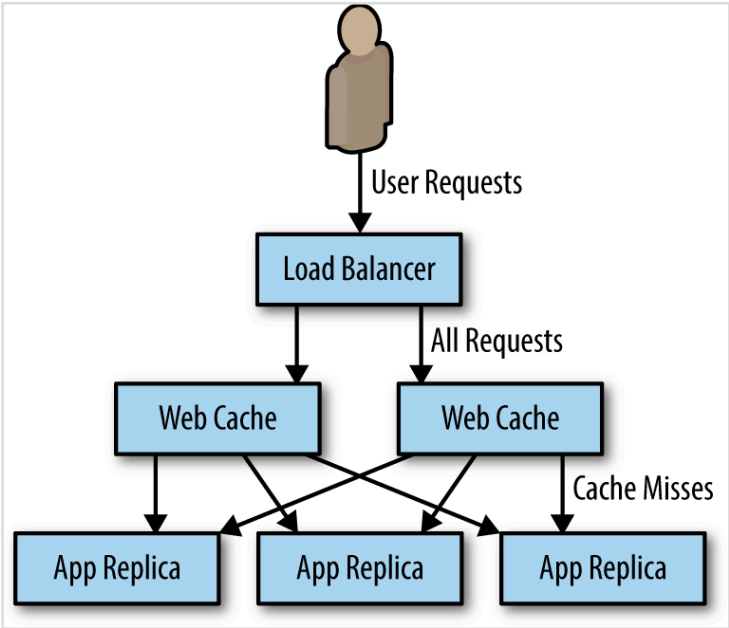


Figure 3: Caching layer added to the design pattern

Apart from helping with performance, the caching layer could enhance security if integrated with some defence layers. Limiting the rate at which a single user requests from the service is generally a good idea to avoid servers' overload or denial-of-service attack. Burns (2018) also suggests to do SSL termination in this layer to offload SSL processing from every backend service to improve performance. [2, 54.]

2.3. Performance analysis

The process of using an automated and controlled method to record the repeated implementation of real user's behaviour as on a tested system is called web performance testing. The test generator is responsible for simulating the high traffic user's behaviour. It can do so by running hundreds to thousands of Web client software. In a performance test, the number of virtual users can be set in the generator. Additionally, between the acts of two tests there can be delay time. [5, 328.]

There are generally three types of web performance testing: stress test, load test and strength test. Stress test is conducted by increasing the system load gradually and observe the system performance changes. The ultimate goal is to put load conditions in a failure state on the system and determine under which conditions the application performance will be unacceptable, therefore discovering bottlenecks in the system. Load test is generally to perform a specific type of stress test, which is to increase the number of users, or the load. Through the load test, the system is tested with a variety of work load, performance indicators are recorded when load is increasing to ultimately determine the maximum load the system can withstand. Finally, strength test is a load test or stress test in a longer interval, meaning the system has to endure the test for a few hours or even days to identify unexpected errors. [5, 329.] According to Huaji and Huarui (2017), load capability of a web server is a key factor in examining the performance of Web applications [6, 1175].

In order to measure performance, a series of performance indicators are chosen. This paper mainly focuses on three: response time, throughput and availability. Response time is counted from when the client sends a request to the response from server. Typically once the user load increases, the response time will be gradually increased as well because some of the system's resources are getting exhausted or the number of concurrent requests exceeds the number of processes or threads the system could afford, leading to accumulated waiting time for upcoming requests to be processed [5, 329]. Throughput is referred as the amount of workload that the server is able to handle calculated on a period of time, typically requests per second. It is believed to be the most useful indicator as it resemblances how many concurrent clients or users a web service could manage in a second. In the initial stage of a load test, the system throughput should increase as more resources are reinforced until it reaches a peak for as all resources have been manoeuvred. Servers with slower response times generally are studied to have lower throughput. [5, 330.] Lastly, availability is the measurement of how reliable a service could serve to clients over time. It is often calculated as the percentage of time that a service is able to provide access over a period of time. Often

times, a server may reject user requests, leading to lower availability, when under heavy traffic of a big number of users. [6, 1175.]

3. Methods and tools

This section shows methodologies and tools that will be used in order to conduct the findings.

3.1. Project approach

O'Reilly (2019) suggested that the most common form of a web service is one that is able to use some type of communication to receive a message calling certain procedures running on a server and return a respectively result [7]. In this project, such a service is implemented and packaged into a distributed system environment following the replicated load-balanced services pattern. While the implementation is progressing, a performance analysis is carried out at specific steps in order to document, compare and give an idea of how different parts of the pattern affect its performance.

3.2. Technology

3.2.1. Scala

The web service is developed in Scala language (version 2.12.6) as it is a high-level language that enables quick transformation of data using its excellent collections. Scala is designed to implement programming patterns in a type-safe and concise manner. Both object-oriented and functional languages' features are seamlessly integrated in Scala. Also it is a JVM language so one could take advantage of the reliable ecosystem and open-source libraries. It is believed to have smaller code sizes when compared with a similar Java application. Scala is being adopted for many existing companies to boost their productivity. Twitter, for example, decided to migrate their core message queue from Ruby to Scala. [8, 2.]

3.2.2. REST

Representational State Transfer, or REST, is an architectural style to guide the definition of the HTTP and URIs in web development. All resources in a REST system have defined URIs. Clients access these resources by using GET, POST, DELTE and PUT as general HTTP interfaces. GET generally returns a representation of the resource. PUT lets a client replace the resource entirely with a new state while DELETE is used

to remove the resource. POST normally is used to create a new resource. The goal of REST is to achieve: (1) scalability (2) generality (3) independency (4) intermediary reduction of components and (5) reduction in latency, security enforcement and legacy systems encapsulated. [9, 1.]

An architecture that applies REST principles is called RESTful.

3.2.3. Play framework

Play framework (version 2.6.17) contributes to providing built-in libraries and backbone to implement a web server. Play is a Java and Scala web application framework designed for high productivity that has integrated components and APIs to develop a modern web application. It has several built-in modules that help with the development of REST services or Web Application such as HTTP server, routing mechanism, JSON parsing handling, etc. Akka and Akka Streams technologies run under the covers to manage resource consumption for minimal usage. [10.]

Play implements Model-View-Controller (MVC) architecture which is a concept of a system divided in three components: model that implies the domain knowledge, view that refers to the interface that user faces and controller that handles view's updates [11,1]. Each component is specialised in its own task. In more details, the model holds the data of application. The view is responsible for visual display of this data or the model. The controller and the view are always linked. It notifies the view to determine which resources are being changed by the user and to propagate this information to model methods that updates new states to these objects. The model handles these changes and it sends updates to the view. A view is usually associated with a unique controller while the model can have multiple view-controller pairs. [11, 2.]

Based on the relationship of the three components, Play is believed to have advantages in less coupling and higher cohesion architecture, flexibility in the views, clarity in design, greater scalability and helps with maintenance facilitation [11, 1].

3.2.4. PostgreSQL

Postgres is chosen as a relational database software that persists data of the service. Throughout the years, by consistently delivering performant and innovative solutions with its proven architecture, reliability, data integrity, many robust feature sets, extensi-

bility and the active contribution of open source community, Postgres has gained tremendous reputation [12].

3.2.5. Kubernetes

Kubernetes is used to simulate a distributed system environment within one single machine. Kubernetes also provides a considerable amount of components that enable replications, load-balancing, auto scaling as well as orchestrates the whole system throughout its lifecycle from deployment to destruction. A Kubernetes cluster consists of Kubernetes Master which is the controller panel, responsible for managing the cluster state and Kubernetes Nodes, which are the machines (physical servers or VMs, etc) belonged to the cluster, responsible for running the deployed applications. Typically, the cluster administrator communicates with Kubernetes Master using command-line interface “kubectl” when interacting with the cluster.

A Pod is the most fundamental unit in a Kubernetes cluster, representing an encapsulation of application container(s) and their storages, an associated unique IP address and settings on how the container(s) may run. When having multiple containers in a Pod, the containers are ensured to be located and scheduled to be on the same physical or virtual server. Resources and dependencies are shared among these containers and they could communicate with each other. However, Pods do not heal by themselves, meaning if a Pod is scheduled in a Node that is under maintenance or in lack of resources, it will be deleted. Therefore, Kubernetes uses controllers to manage Pods. [13.] ReplicaSet is such a controller that is responsible for maintaining a stable number of replicas of a Pod at any time [14].

A Kubernetes Deployment is a controller that manages state changes for Pods and ReplicaSet. It has many states in a lifecycle: progressing, complete or failed. A Deployment is in progressing phase if a new ReplicaSet is created, a ReplicaSet is scaling up/down or new Pods become available. A Deployment is marked complete if it could make sure: all updates to the replicas have finished, all replicas are available and no old replicas are running. On the other hand, any of the errors that prevent the Deployment from deploying the newest ReplicaSet would cause the Deployment to get stuck, or failed, such as: insufficient resources, readiness probe failure, errors in pulling Docker images, etc [15.]

In Kubernetes, it is necessary to understand the concept of a Service, which is an abstraction that targets Pods based on their labels and defines a policy to access those

Pods. To address a particular Service, it is possible to use its record in Kubernetes DNS server, which is a storage of mappings between a Service's name and its IP address. Therefore, it is sufficient to remember the name of a Service when trying to communicate with it. For example, a Service with name "my-app" in Namespace "my-ns" would have a DNS record of "my-app.my-ns" which is then discoverable inside the cluster. The default mode of proxying the request to a Service's Pods is called "iptables", which essentially chooses a Pod randomly when routing the traffic. The level of exposure of a Service is determined by its "ServiceType", which could be one of the followings: ClusterIP, NodePort, LoadBalancer and ExternalName. ClusterIP is the default type and only allows the Service to be reachable inside the cluster. NodePort type proxies the configured port on all Nodes into the Service which allows clients from outside the cluster to reach via <NodeIP>:<NodePort>. LoadBalancer type enables a cloud load balancer to route traffic to the Pods inside the cluster. And finally, ExternalName creates a DNS record that maps that Service's name to a specified "externalName". [16.]

Horizontal Pod Autoscaler (HPA) is a Kubernetes component that observes CPU utilisation of a set of Pods and ensures that the number of Pods is scaled up or down to match a configure CPU target. Every 15 seconds (default), HPA queries for each targeted Pod's CPU utilisation from the resource metrics API and calculates a mean value then compare with the target CPU to define a ratio used to calculate the number of desired replicas. Readiness of a pod and missing metrics are taken into consideration when computing. [17.]

ConfigMap is a storage object used inside Kubernetes to bind configuration artefacts to a Pod's containers. In this way, configuration data is decoupled from the Pod specifications, which prevents hardcoded values and at the same time shares this configuration information to any other parties interested in the cluster. [18.]

3.2.6. Varnish

Varnish is a web application accelerator that typically is install in front of a server that speaks HTTP and is configured to cache specific contents. It is believed to be able to speed up delivery 300-1000 times faster. A key feature of Varnish is the flexibility in configurations using its own domain language, VCL. Policies written in VCL help configure what content to be served, where to get the content and modifications to the request or response. [19.]

3.2.7. Jmeter

Apache Jmeter is an open source software, designed to do load testing and measure performance. Static or dynamic resources and Web Application could be performance tested using Jmeter. Different load types can be simulated on a server or group of servers to test its strength and analyse overall performance. [20.]

Jmeter acts as the main tool responsible for conducting performance testing. Different configurations are written to adapt with different scenarios.

3.3. Tools

MacOS is the chosen operating system with built-in Terminal used as an interface to instruct different commands.

Codebase of the web server and deployment files to Kubernetes are maintained in version control Github. IntelliJ and Vim serve as main IDE and lightweight text editor respectively.

4. Implementation

4.1. The application

4.1.1. Design

As presented in 3.1, the application is subjected to be a common web service that provides an interface via RESTful architecture for interested parties to retrieve and save information. For easy understanding, the application's context is taken as financial operations in a bank such that it enables retrieving bank account details and depositing as well as withdrawing.

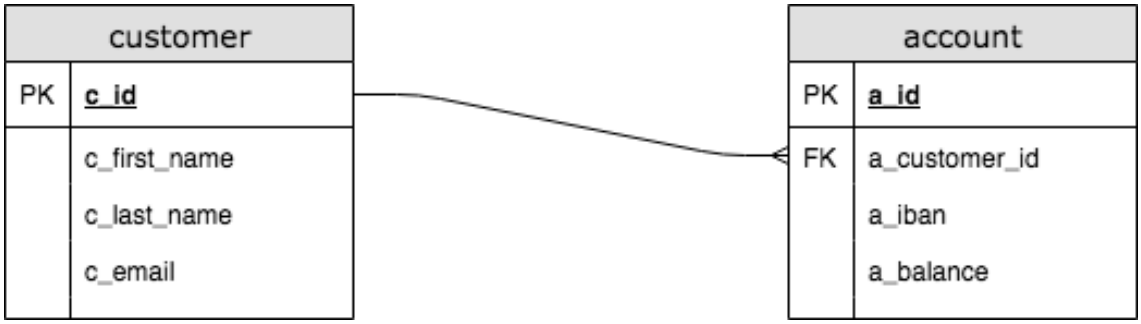


Figure 4: Relational database model

As shown in Figure 4, a simple model consisted of a customer with multiple bank accounts is implemented in Postgres database.

The interface to talk to the service is in RESTful style as described in the following table.

HTTP Verb	Path	Usage
GET	/accounts/:accountId	Get information of the bank account given accountId
POST	/accounts/:accountId/balance	Withdraw or deposit money to a bank account balance
GET	/exchanges/eur_usd	Get information of EUR_USD exchange rate

Figure 5: RESTful endpoints of application

Per each request, the application tries to interact with its database and response back.

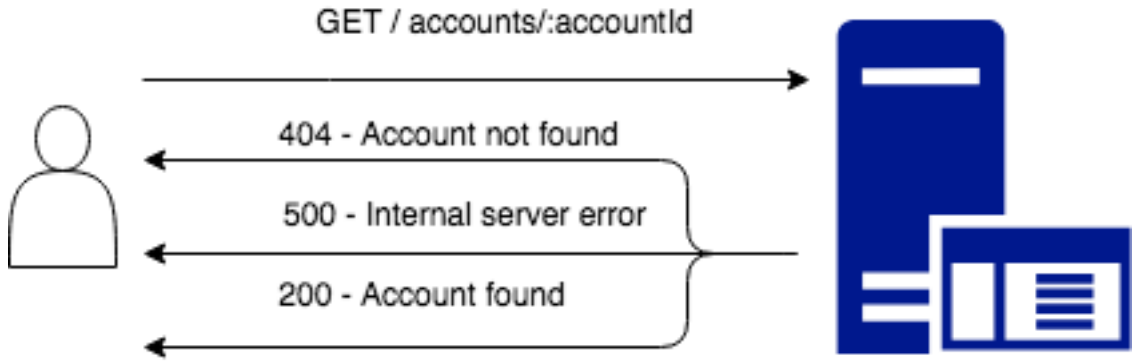


Figure 6: Communication with user in GET /accounts/:accountId endpoint

As seen from Figure 6, given a GET request to /accounts/:accountId, the application looks up for the respective account from the database and returns response with status code 200 and account details in Json format if the account is found, status code 404 if not found and 500 with detailed error for other scenarios.

On the other hand, the POST endpoint is slightly more complicated as it is responsible for both withdrawing and depositing money to account balance.

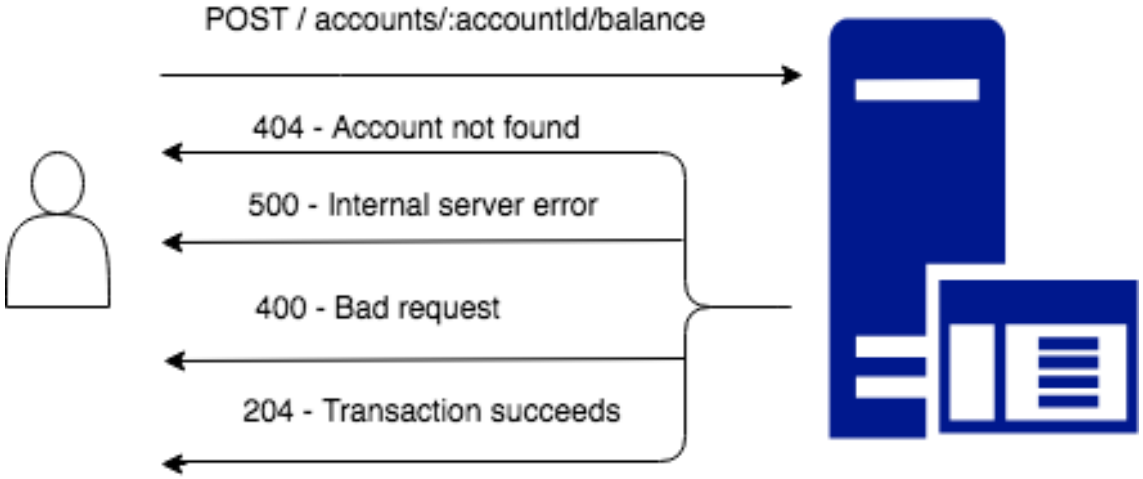


Figure 7: Communication with user in POST /accounts/:accountId/balance

Figure 7 shows an addition response 400 Bad request compared to the presented GET request as the information of the request body can be considered improper if the application could not understand or it is logically incorrect to fulfil such request. An example would be when the balance is insufficient to process a withdraw.

The other GET /exchanges/eur_usd endpoint is a proxy to a public API that provides exchange rates with continuous updates. Therefore, all the responses are mirrored via the application. More details about this endpoint will be presented in the following section.

4.1.2. Play framework integration

The application is structured according to Play's guidelines as shown in Figure 8.

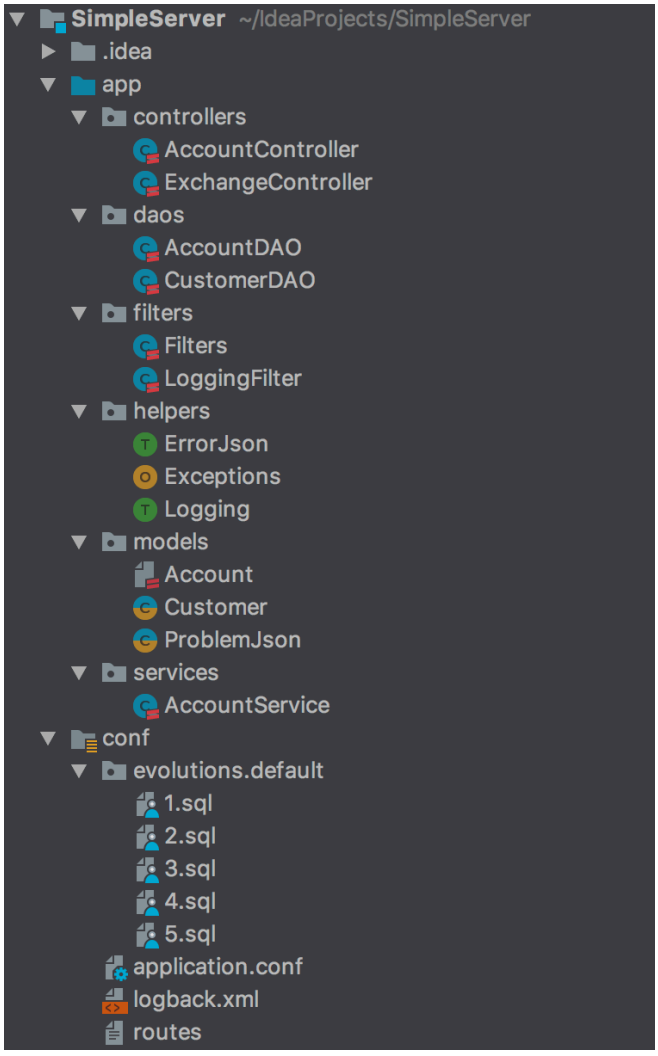


Figure 8: Anatomy of the application

Controllers are the modules that call Services and handle requests from users as well as giving back respective responses. Services interact with DAO layer and apply business logic or data transformation before giving back to Controllers. The last layer in the service communication flow being DAO is responsible for talking with application's

database. Models are object representations needed by the application to pass around different layers, most important ones being the data persisted in Postgres. Additionally, Filters act as a middleware between users and Controllers while Helpers are useful methods that are used across the application. Database model changes are implemented and version-controlled using Play's native evolutions.

Given an input, a controller calls services to get respective outputs and ultimately wraps a response that should meet user's expectation under defined format.

```
def getAccount(id: Int) = Action.async { implicit request =>
  accountService.get(id) map {
    case Right(accountOption) =>
      if (accountOption.isDefined) {
        Ok(Json.toJson(accountOption.get))
      } else {
        respondWithError("Not found",
          NOT_FOUND,
          "The account with specified id is not found.")
      }

    case Left(e) =>
      logger.error(s"Error getting account", e)
      respondWithError(
        "Internal Server Error occurred",
        INTERNAL_SERVER_ERROR,
        e.msg)
  } recover {
    case NonFatal(e) =>
      logger.error("Error getting account", e)
      respondWithError(
        "Internal Server Error occurred",
        INTERNAL_SERVER_ERROR,
        e.getMessage)
  }
}
```

Listing 1. Excerpt of getting an account in Controller

As shown in listing 1, "accountService" is called with an ID to get the respective account. Controller returns Ok - 200 with the account in Json body if it is found or Not found - 404 with an explanation. For other reasons that the service fails to complete the procedure, an Internal server error - 500 is returned with its error message.

```
def withdraw(id: Int, amount: BigDecimal): Future[Either[RuntimeException,
Unit]] = {
  accountDAO.withdraw(id, amount) map {
    case Right(_) => Right(())
    case Left(e) =>
      if (e.msg.contains("balance_nonnegative")) {
        Left(InsufficientAmountException)
      } else {
        Left(e)
      }
  }
}
```

Listing 2. Excerpt of withdrawing money in Service

Listing 2 describes some code lines that involve business logic, in this case determining whether balance has sufficient money to conduct a withdraw, in a Service.

The last layer in the flow of fulfilling a user request is the DAO methods. Below shows an example when trying to communicate with database to withdraw money of a bank account.

```
def withdraw(id: Int, amount: BigDecimal): Future[Either[DatabaseException,
Int]] =
  Future {
    Try {
      db.withConnection { implicit connection =>
        SQL("UPDATE account SET a_balance = a_balance - {amount} WHERE a_cus-
tomer_id = {id}")
          .on(
            "id" -> id,
            "amount" -> amount
          )
          .executeUpdate()
      }
    }.toEither.left.map { e =>
      DatabaseException(e.getMessage)
    }
  }
```

Listing 3. Excerpt of withdrawing money in the DAO

```
def get(id: Int): Future[Either[DatabaseException, Option[Account]]] =
  Future {
    Try {
      db.withConnection { implicit connection =>
        SQL("SELECT * FROM account WHERE a_customer_id = {id}")
          .on(
            "id" -> id
          )
          .as(parser.*)
          .headOption
      }
    }.toEither.left.map { e =>
      DatabaseException(e.getMessage)
    }
  }
```

Listing 4. Excerpt of getting bank account in the DAO

Anorm, Play's database library, enables giving instructions to the database using plain SQL language and transforms results into Scala classes as seen in listing 3 and 4.

4.2. Infrastructure and environment

4.2.1. Architectural design

Design of how components communicate with each other is demonstrated in Figure 9 below.

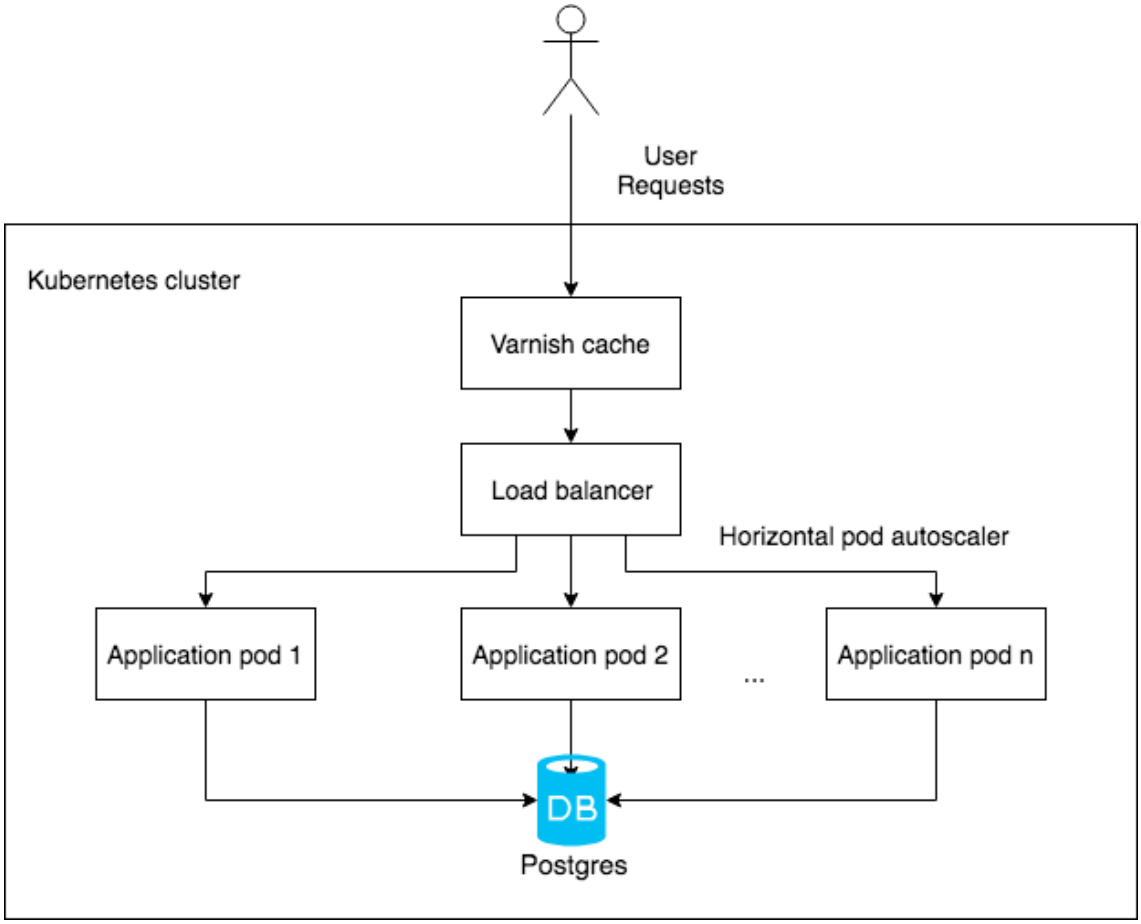


Figure 9: High-level architecture

User request typically comes outside of Kubernetes and firstly passes by Varnish which then checks if this request is configured to be cached and gives back already stored response if possible. If not, the request then will directly try to reach the web application which is routed randomly by the load balancer, which essentially is Kubernetes Service. The number of application pods is determined by the mechanism of Kubernetes horizontal pod autoscaler that tries to ensure not any single pod has CPU utilisation over a configured target. Finally, the application processes the request and communicates with Postgres database to give back an HTTP response.

4.2.2. Minikube setup

The process of installing Minikube is easiest using Homebrew on macOS with some dependencies setup such as Virtualbox and kubectl [21].

```
minikube start --cpus 2 --memory 4048
```

Listing 5. Starting minikube

Listing 5 shows a simple command line to issue a local Kubernetes cluster - minikube.

To check whether minikube is running without any problems, a command shown in listing 6 is necessary.

```
minikube status
```

Listing 6. Checking minikube status

4.2.3. Application deployment

The application is packaged into a Docker image with the help of external library called “sbt-native-packager”. Once installed properly, configurations of the image are defined in sbt configuration file as shown in listing 7.

```
lazy val dockerSettings = Seq(
  dockerBaseImage := "openjdk:8-slim",
  dockerExposedPorts := Seq(9000)
)


lazy val root = (project in file("."))
  .settings(commonSettings:_*)
  .settings(dockerSettings:_*)
  .settings(libraryDependencies ++= libraries)
  .enablePlugins(PlayScala, JavaAppPackaging)
```

Listing 7. Docker settings in build.sbt

The most important setting being “dockerBaseImage” which is set to a light OpenJDK image.

Given this setup, a simple sbt command could compile Scala classes to byte codes and link the Docker process to JVM execution that runs the application. That Docker image is then ready to be deployed into Kubernetes pods without much difficulty. De-

clarative object configuration is chosen as the sole method to create resources in Kubernetes cluster because changes to these configuration files could be stored in source control system like Git. Such a configuration file is constructed following a Kubernetes template which depends on the kind of resource and is possible to be customised with different arguments and variables.

```
kubectl apply -f deploy/
```

Listing 8. Applying the resource configurations to cluster

Once the files are ready, the management of their resources are applied to minikube using kubectl tool like shown in listing 8.

The application needs a Deployment template, which most importantly defines the Docker image needed to be deployed for each pod and its pod configs as shown in listing 9. Number of replicas and resources request/limit are modified extensively during performance analysis.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    run: http-server
  name: http-server
spec:
  replicas: 1
  selector:
    matchLabels:
      run: http-server
  template:
    metadata:
      labels:
        run: http-server
    spec:
      containers:
      - image: simpleserver:0.8
        imagePullPolicy: Never
        name: http-server
        ports:
        - containerPort: 9000
          protocol: TCP
        resources:
          requests:
            memory: "800Mi"
            cpu: "0.2"
          limits:
            memory: "1200Mi"
            cpu: "0.4"
```

Listing 9. Application deployment file

Each of the pods then is associated with a Kubernetes service which is meant to enable communication with other pods inside the cluster.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    run: http-server
  name: http-server
spec:
  type: NodePort
  ports:
  - port: 9000
    protocol: TCP
    targetPort: 9000
  selector:
    run: http-server
```

Listing 10. Application service file

Listing 10 demonstrates a service configuration file where the associated pods are matched using their labels, pod's port inside container and service pod inside cluster is linked up. This service is then exposed to the outside world at a static port on its Node's IP, meaning one is able to access this service via <NodeIP>:<NodePort>, which is necessary in order to conduct load tests on the application using a software.

4.2.4. Postgres deployment

Similarly, Postgres' configurations are managed in the same Deployment template with minor differences as can be seen in listing 11.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
  labels:
    run: postgres
  name: postgres
spec:
  replicas: 1
  selector:
    matchLabels:
      run: postgres
  template:
    metadata:
      labels:
        run: postgres
    spec:
      containers:
      - image: postgres:10.0
        name: postgres
        ports:
        - containerPort: 5432
          protocol: TCP
        resources:
          requests:
            memory: "800Mi"
            cpu: "0.2"
          limits:
```

```
            memory: "1200Mi"
            cpu: "0.4"
```

Listing 11. Postgres deployment file

And its companion Kubernetes service is required to make it accessible within the cluster as in listing 12.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    run: postgres
  name: postgres
spec:
  type: NodePort
  ports:
  - port: 5432
    protocol: TCP
    targetPort: 5432
  selector:
    run: postgres
```

Listing 12. Postgres service file

Pods inside the cluster now could communicate with Postgres by requesting to postgres:5432

4.2.5. Horizontal pod autoscaler

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: http-server
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

Listing 13. HorizontalPodAutoscaler configuration file

The configuration of auto scaling of an application as described in listing 13 simply contains minimum and maximum number of pods as well as a target CPU utilisation where Kubernetes tries to ensure that the average of CPU metric across all pods reaches the target by scaling up or down.

4.2.6. Varnish

As the cache is placed in front of the application, certain configurations written in VCL language have to instruct Varnish to follow the request to the correct service as well as its cache behaviour as shown below in listing 14 where the backend to be sending traffic to is “http-server” and associated port 9000, which is the port of Service http-server inside the cluster. Upon receiving a request, Varnish will assess in subroutine “vcl_recv” whether to pass directly to the backend (pass) or to modify and process later (hash). Here a request with method GET and header X-cache_enabled would suffice to be a candidate for caching. From there, Varnish will lookup for a cached response; if successful, it will return the cached response right away in subroutine “vcl_deliver”; if not, the request will be forwarded to the respective backend for a response and arrives at “vcl_backend_response” which creates a cached data with time-to-live of 30 minutes.

```
vcl 4.0;
backend default {
  .host = "http-server";
  .port = "9000";
}

sub vcl_recv {
    if (req.method == "GET" && req.http.X-cached-enabled) {
     return (hash);
    }
    return (pass);
}

sub vcl_backend_response {
    if (beresp.ttl <= 0s ||
      beresp.http.Set-Cookie ||
      beresp.http.Surrogate-control ~ "no-store" ||
      (!beresp.http.Surrogate-Control &&
        beresp.http.Cache-Control ~ "no-cache|no-store|private") ||
      beresp.http.Vary == "*") {
        /*
        * Mark as "Hit-For-Pass" for the next 0.5 minutes
        */
        set beresp.ttl = 30m;
        set beresp.uncacheable = true;
    }
    return (deliver);
}

sub vcl_deliver {
    if (obj.hits > 0) {
        set resp.http.X-Cache = "HIT";
    } else {
        set resp.http.X-Cache = "MISS";
    }
    return (deliver);
}
```

Listing 14. Varnish configuration in VCL language

As the endpoint to be cached itself is refreshed every hour, it is necessary that each cached response is set to be expired less than that to make sure application could get access to newest data. In this case, it is set to be 30 minutes as the maximum affordable delay.

In order to have these configurations accessible inside Kubernetes cluster, a Configmap is deployed alongside with Varnish as shown in listing 15.

```
apiVersion: v1
data:
default.vcl: "vcl 4.0;\nbackend default {\n  .host = \"http-server\";\n  .port
=\"9000\";\n}\n\nsub vcl_recv {\n if (req.method == \"GET\" && req.http.X-
cached-enabled){\n\treturn (hash);\n}\n return (pass);\n}\n\nsub vcl_back-
end_response{\n if (beresp.ttl <= 0s ||\n beresp.http.Set-Cookie ||\n
beresp.http.Surrogate-control~ \"no-store\" ||\n(!beresp.http.Surrogate-Con-
trol &&\n beresp.http.Cache-Control ~ \"no-cache|no-store|private\") ||\n
beresp.http.Vary == \"*\") {\n/*\n\* Mark as \"Hit-For-Pass\" for the next 2
minutes\n */\n set beresp.ttl = 30m;\n set beresp.uncacheable = true;\n    }\n
return (deliver);\n}\n\nsub vcl_deliver {\n    if (obj.hits > 0) {\n
set resp.http.X-Cache = \"HIT\";\n\} else {\n set resp.http.X-Cache =
\"MISS\";\n}\n return (deliver);\n}\n\n"
kind: ConfigMap
metadata:
  creationTimestamp: null
  name: varnish-config
  selfLink: /api/v1/namespaces/default/configmaps/varnish-config
```

Listing 15. Varnish configuration in Configmap

This Configmap is mounted as a volume in Varnish pod.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: varnish-cache
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: varnish-cache
    spec:
      containers:
      - name: cache
        resources:
          requests:
            # We'll use two gigabytes for each varnish cache
            memory: 1Gi
        image: brendanburns/varnish
        command:
        - varnishd
        - -F
        - -f
        - /etc/varnish-config/default.vcl
        - -a
        - 0.0.0.0:8080
        - -s
        # This memory allocation should match the memory request above
        - malloc,1G
```

```
    ports:
    - containerPort: 8080
    volumeMounts:
    - name: varnish
      mountPath: /etc/varnish-config
  volumes:
  - name: varnish
    configMap:
      name: varnish-config
```

Listing 16. Varnish deployment file

Listing 16 demonstrate a similar deployment file for Varnish where daemon “varnishd” is passed a path to get VCL configuration.

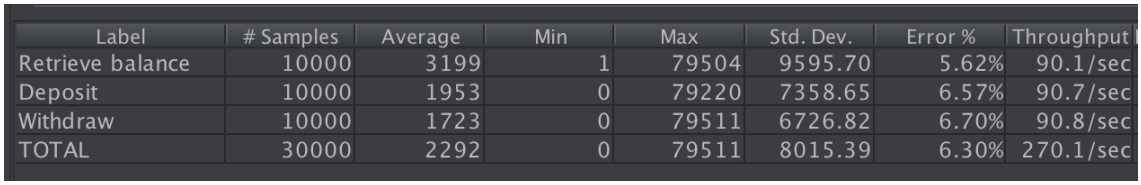
5. Performance analysis

5.1. Test cases

In order to achieve a high-performant design, it is vital to conduct testing with changing configurations, to define what is the most effective arrangement of components and their settings. Two main test cases are presented: (1) Number of application pods and (2) The use of caching. All tests are done using the same machine with the following specifications: MacBook Pro 2017, macOS High Sierra, 2.3 GHz Intel Core i5, 16 GB 2133 MHz LPDDR3 of memory and 256 GB of SSD. Fake test data (account information and their bank details) are generated with 1000 records each. Jmeter is used as performance testing software that directly sends out HTTP requests to the web service then collects results and produces a consistent summary report.

5.2. The number of application pods

This first test is about how significant a performance improvement it is when increasing number of application pods. Jmeter simulates a heavy traffic with 1000 concurrent users retrieve random bank account balances then deposit and withdraw money from their accounts. To increase reliability, each round is looped 10 times. All traffics are sent to the application's REST API with respectively GET "/accounts/:accountId" to get bank details then POST "/accounts/:accountId/balance" to change account's balance (or in other words deposit/withdraw).



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput
Retrieve balance	10000	3199	1	79504	9595.70	5.62%	90.1/sec
Deposit	10000	1953	0	79220	7358.65	6.57%	90.7/sec
Withdraw	10000	1723	0	79511	6726.82	6.70%	90.8/sec
TOTAL	30000	2292	0	79511	8015.39	6.30%	270.1/sec

Figure 10: Result of test with 1 pod

As shown in Figure 10, with only 1 pod, the average elapsed time to retrieve, deposit then withdraw per user was 2292 ms. Most importantly, the web service processed 270 requests per second with 6.3 % error rate, which resulted from timed out operations because the application could not response back in time.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput
Retrieve balance	10000	2094	2	65760	6047.51	3.13%	122.3/sec
Deposit	10000	1552	0	66014	5060.26	3.53%	123.3/sec
Withdraw	10000	1438	0	63282	4599.90	3.99%	123.5/sec
TOTAL	30000	1695	0	66014	5278.35	3.55%	366.8/sec

Figure 11: Result of test with 2 pods

Observed from Figure 11, when switching to having 2 pods at the same time, the web service performance favoured from a great 35% increase in throughput (366.8/sec compared to 270.1/sec) and a 43.65% decrease in error rate (3.55% compared to 6.3%). This improvement was expected as there were twice as many resources to fulfil the requests.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput
Retrieve balance	10000	1357	2	43324	3910.58	1.00%	172.4/sec
Deposit	10000	1040	0	36175	3303.99	1.10%	174.0/sec
Withdraw	10000	1043	0	43964	3275.81	1.48%	174.1/sec
TOTAL	30000	1147	0	43964	3512.19	1.19%	517.2/sec

Figure 12: Result of test with 3 pods

Similarly in Figure 12, the throughput tremendously increased to a 517.2 requests per second when applying 3 pods, which was roughly 41% improvement in comparison with 2 pods' result while error rate dropped down to 1.19%.

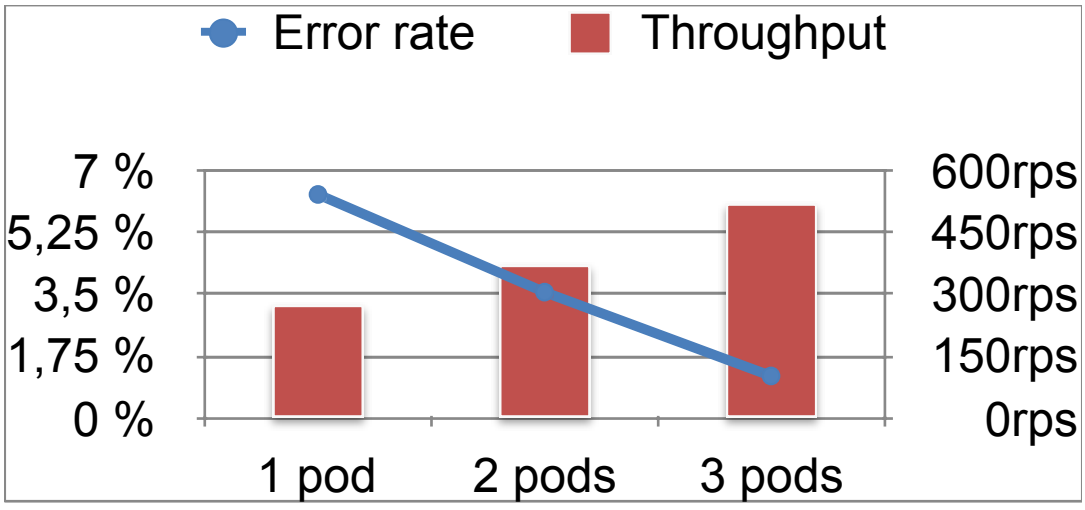


Figure 13: Comparison of results in graph

From Figure 13, it can be seen that the performance of the web service is linear to the number of replicas of the application. The more resources available to scale up, the more replicas could be spawn which means as user traffic increases in a day, it is possible to use autoscaling mechanism, like HPA provided by Kubernetes, to adapt based on the change. This means the amount of resources being consumed by the web service would change according to how the demand requires and directly leads to efficient resources allocation in the cluster.

Given a target CPU utilisation of 50%, minimum 1 pod and maximum 3 pods configured in HPA, running the same test again, the number of pods automatically increased to 3 to satisfy heavy traffic then reduced back to 1 after the test as shown in Figure 14.

```
~/IdeaProjects/SimpleServer  master ●+  kubectl describe hpa hpa
Name:                                                  hpa
Namespace:                                             default
Labels:                                                <none>
Annotations:                                           kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"autoscaling/v1","kind":"H
orizontalPodAutoscaler","metadata":{"annotations":{},"name":"hpa","namespace":"default"},"spec":{"maxReplicas":...
CreationTimestamp:                                     Sat, 02 Feb 2019 18:03:01 +0700
Reference:                                             Deployment/http-server
Metrics:                                               ( current / target )
  resource cpu on pods  (as a percentage of request):  2% (4m) / 50%
Min replicas:                                          1
Max replicas:                                          3
Deployment pods:                                       1 current / 1 desired
Conditions:
  Type            Status  Reason              Message
  ----            ------  ------              -------
  AbleToScale     True    ReadyForNewScale    recommended size matches current size
  ScalingActive   True    ValidMetricFound    the HPA was able to successfully calculate a replica count from cpu resource utilization (percenta
ge of request)
  ScalingLimited  False   DesiredWithinRange  the desired count is within the acceptable range
Events:
  Type    Reason             Age             From                       Message
  ----    ------             ----            ----                       -------
  Normal  SuccessfulRescale  7m (x2 over 7h)  horizontal-pod-autoscaler  New size: 3; reason: cpu resource utilization (percentage of request) a
bove target
  Normal  SuccessfulRescale  1m (x2 over 7h)  horizontal-pod-autoscaler  New size: 1; reason: All metrics below target
```

Figure 14: Pods autoscaling logs from HPA

Under HPA configuration, the performance was a bit less than having consistent 3 pods (448.9 RPS in Figure 15) but resources are free after traffic is reduced. This essentially means that not much administration effort has to be spent in order to cope with fluctuating web traffic throughout the day and still ensure the resources are allocated efficiently within the overall system.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput
Retrieve balance	10000	1577	1	53035	4389.25	1.38%	149.7/sec
Deposit	10000	1319	0	51371	3876.97	1.65%	151.1/sec
Withdraw	10000	1294	0	52680	3859.12	1.86%	151.4/sec
TOTAL	30000	1396	0	53035	4051.27	1.63%	448.9/sec

Figure 15: Result of test with HPA configuration

5.3. The use of caching

As discussed in Theoretical background section, responses are often cached to help with performance if they are consistent and do not change often. The API to get currency exchange rate “/exchanges/eur_usd” is such one as the source of the data only updates its information every 60 minutes. Varnish is configured to save the response and keep it alive for 30 minutes, meaning in the worst case it takes 30 minutes to get the latest update from the source but the performance should be greatly significant as an advantageous tradeoff. In this scenario, it is assumed that a possible delay of 30 minutes is acceptable to the business.

In this test, Jmeter proceeded to simulate 100 users for 10 rounds and tried to get exchange rate from the web server.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput
Get rate	1000	372	251	1369	156.08	23.20%	42.5/sec
TOTAL	1000	372	251	1369	156.08	23.20%	42.5/sec

Figure 16: Result of test without cache

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput
Get rate	1000	1	0	68	2.66	0.00%	50.5/sec
TOTAL	1000	1	0	68	2.66	0.00%	50.5/sec

Figure 17: Result of test with cache

As observed from both Figure 16 and Figure 17, with the cache in place, the performance was considerably better with 0% error rate, 1 ms response time in average and 50.5 RPS.

6. Discussion

Though not mentioned in the performance analysis, it is probably fundamentally clear that given an environment with fixed amounts of power (CPU, RAM, etc), having more replications (more pods, each pod has smaller resources) might decrease performance since the machine has some overheads of using its power to run abstraction processes created by the applications. At the essence of a computer, the number of cores of a processor enables parallelism processing. However, it might be that a single application already understands how to take advantages of multi-core processor, hence achieving maximum potentials. Therefore, having more applications may not mean better parallelism but create overheads for the machine. In order to have a better understanding in regards to this theory, it might make sense to perform load tests and analyse at the low-level processes/threads of the machine to see the impact.

In practice, the fact that having more replications, each having the same resources, increases performance already assumes that the system has additional resources to rescue under heavy load. In that sense, autoscaling the number of pods is a powerful method to ensure high performance but minimised cost in a distributed system, assuming that the whole system is similar to a virtual machine that could agilely scale up/scale down, which is usually a feature that cloud providers like Amazon Web Service supports. Although the HPA configuration used in this research takes average CPU utilisation as a method to calculate the number of replicas, it is also possible to have custom metrics that may be more relevant in a practical context. It may take metrics that actually impact the business. For example: a service has SLI (Service Level Indicator), which is the measurement of service to reach its business goal, of 95% of user requests have response time less than 200ms may be taken as a variable in autoscaling algorithm.

Generally, it is recommended to have a minimum number of replicas larger than one without consideration of performance. Because it may increase the availability or fault-tolerance of the system. For example, an application with 50% failure rate may get up to 25% failure rate when having another replica. Or when rolling out a new version of the application, it is undoubtedly necessary to make sure there are some replicas running to still serve the clients while some are getting updated to the newer version. Another good reason to have multiple replicas is to place them in different geographical locations in order to have better chances of having a server that is closer to a client, meaning the request may reach a server faster and the client gets faster response. Coming up with a balanced minimum number of replications that provides best out-

come of both throughput and reliability probably requires extensive testing (load test, strength test, etc) under different conditions.

As mentioned when discussing load balancers in section Two, different algorithms of choosing where to route traffic to could also have an impact in performance. Since Kubernetes 1.9, a feature in beta testing called “ipvs” is a new proxy-mode for its Service. It is promised to be faster and more importantly, provides different options for load balancing algorithms. The analysis conducted in this paper used a random strategy, which may not provide the best outcome. Testing with different algorithm, especially ones that take into account the states of the applications before making a decision where to go to may have a promising outcome.

7. Conclusion

All in all, the replicated load-balanced services pattern is considered reliable and high-performant for three reasons. Firstly, it ensures high availability through replications of the service. Secondly, loads among these instances are distributed equally by load balancers. Thirdly, heavy computations possibly do not have to happen in the services frequently thanks to the caching layer. The implementation of such design pattern is facilitated by the rapid development of virtualisation and containerisation technologies.

In order to achieve the optimal performance using this pattern, it is essential to acknowledge that the more application pods, the more throughput they provide. However, it is probably wise to apply some autoscaling mechanism to both ensure a high throughput and at the same time cut down unnecessary resources during low-traffic time.

The use of caching is a great tool to offload heavy processing in the backend and provides high performance but it comes with a cost of potentially losing latest data. In that case, configuring a sensible time-to-live while understanding the business impact if in worst case scenarios would bring out the best values of caching and mitigate possible risks.

References

1. Brendan Burns. Designing distributed systems [online]. California, USA: O'Reilly Media, Inc; 2018 Available from: https://www.cbronline.com/wp-content/uploads/dlm_uploads/2018/04/Designing_Distributed_Systems.pdf [Accessed 3 October 2018].

2. Kamble A.L, Shinde T.K, Kothiwale N., Khot S.S. Real Time And Distributed Computing Systems. Second International Conference on Emerging Trends in Engineering (SICETE). 2013; Vol.3: pp.53-56.

3. Anu Kaul, Meena Gupta, Khushwant Kaur. Using virtualisation for distributed computing. International Journal of Advances in Electronics and Computer Science. July, 2015; Vol.2; Issue 7: pp.37-39.

4. T. Deepa, Dhanaraj Cheelu. A comparative study of static and dynamic load balancing algorithms in cloud computing. International Conference on Energy, Communication, Data Analytics and Soft Computing . Aug 2017; pp.3375-3378

5. Kunhua Zhu, Junhui Fu, & Yancui Li. (2010). Research the performance testing and performance improvement strategy in web application. 2010 2nd International Conference On Education Technology And Computer. Shanghai, 2010, pp. V2-328-V2-332.

6. Z. Huaji and W. Huarui, Research on web application load testing model. 2017 IEEE 2nd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC), Chengdu, 2017, pp. 1175-1178.

7. O'Reilly (2019). *Programming Web Services with SOAP*. [online] Available at: https://learning.oreilly.com/library/view/programming-web-services/0596000952/ch01.html [Accessed 11 Feb. 2019].

8. M. S. Bhat, D. G. Nair, D. Bansal and J. Vaishnavi, "Data structure based performance evaluation of emerging technologies — A comparison of Scala, Ruby, Groovy, and Python," *2012 CSI Sixth International Conference on Software Engineering (CONSEG)*, Indore, 2012, pp. 1-5

9. H. Li, "RESTful Web service frameworks in Java," 2011 IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC), Xi'an, 2011, pp. 1-4.

10. Play framework documentation (2019). *What is Play?* [online] Available at: https://www.playframework.com/documentation/2.7.x/Introduction [Accessed 30 March 2019].

11. D. M. Selfa, M. Carrillo and M. Del Rocio Boone, "A Database and Web Application Based on MVC Architecture," *16th International Conference on Electronics, Communications and Computers (CONIELECOMP'06)*, Puebla, Mexico, 2006, pp. 48-48.

12. ProgreSQL documentation (2019). About PostgreSQL. [online]. Available at: https://www.postgresql.org/about/ [Accessed 31 March 2019].

13. Kubernetes.io. (2019). *Kubernetes pods*. [online] Available at: https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/ [Accessed 31 March 2019].

14. Kubernetes.io. (2019). *Kubernetes ReplicasSet*. [online] Available at: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/ [Accessed 31 March 2019].

15. Kubernetes.io. (2019). *Kubernetes deployments*. [online] Available at: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ [Accessed 31 March 2019].

16. Kubernetes.io. (2019). *Kubernetes services*. [online] Available at: https://kubernetes.io/docs/concepts/services-networking/service/ [Accessed 1 April 2019].

17. Kubernetes.io. (2019). *Kubernetes Horizontal Pod Autoscaler*. [online] Available at: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ [Accessed 1 April 2019].

18. Google Cloud. (2019). *Kubernetes Configmap* [online] Available at: https://cloud.google.com/kubernetes-engine/docs/concepts/configmap [Accessed 1 April 2019].

19. Varnish (2017). Varnish introduction. [online] Available at: https://varnish-cache.org/intro/index.html#intro [Accessed 31 March 2019].

20. Jmeter (2019) [online] Available at: https://jmeter.apache.org/ [Accessed 31 March 2019].

21. Kubernetes.io. (2019). *Install Minikube*. [online] Available at: https://kubernetes.io/docs/tasks/tools/install-minikube/ [Accessed 25 Feb. 2019].