Building high availability systems using Docker

Taivo Käsper
University of Tartu
Supervised by Eero Vainikko

Abstract

Shipping applications to different environments is always trivial when it involves copying the binary to another machine and running it. When the number of dependencies goes up and custom configuration is needed the complexity of the job raises immediately. As a result of this many servers run obsolete and unsafe software that has not been updated in a while - it is a risk to update because something might break and the rollback is not easy to conduct. What if reproducing the whole software environment from scratch took just a few commands and moving the setup from machine to machine was as easy?

I. Introduction

Very often the development configuration is done by the developer who is using the machine but the test environment and the production environment is created and managed by a dedicated person. It is inevitable that the setups are different and some things work on one environment and not on the other - humans cannot reproduce exact changes well, underlaying architectures differ and the creation time varies which makes different software versions available, over time modifications are done to separate environments etc.

In contrast to Java programming language which has for one reason become very popular because of it's *Write once, run anywhere* (WORA) capability there has not been a much progress regarding server setups which would enable the strengths of WORA in infrastructure deployments [1] [2].

A docker container system is created that takes as an input a Java Archive file (JAR), knows how to execute multiple instances of it and spread the traffic across all of them. The same is applied to a database that is automatically distributed across multiple instances and can self heal without losing data when few of them stop working. This helps to increase the availability of the application and integrity of the data without requiring extra work from the developer other than using the docker container system developed for deployments.

The project promises to make scaling of a microservices easier for developers by providing general purpose tools that can deploy any Java based web application into a scaled environment. It also aims to provide deployment with no downtime by directing traffic to instances that are up and updating them one-by-one.

II. Introduction to Docker

Docker is, in its own words, "an open platform for developers and sysadmins to build, ship, and run distributed applications". In other, simpler words, Docker is a container manager. Note that it's not a virtualization solution that abstracts away the underlying OS or hardware, like the well-known

VirtualBox, but an engine to package an individual application system to run in its own particular environment.

It means that Docker can be used as a virtualenv in Python, sandboxing the application dependencies so as not to let them to get into dependency management hell. [3]

Docker allows to create containers of software and it's configurations, after creation the containers they can be run on any environment without the need to make changes. The container will be a wrapper around the application with it's dependencies and all the side products created by it during it's lifecycle will remain in the container, unless specified differently. This allows to delete the container with all of it's installed dependencies, logs etc. what is left behind is a clean server environment with only Docker installed.

To describe the sandboxing effect a Docker container might be compared to a VirtualBox virtual machine which has some folders in sync with the host machine and some host machine internet socket ports (port) forwarded to a guest machine port but Docker is really something else. It can easily transfer containers across network, for container storage and sharing it has a web environment Docker Hub which is similar to what GitHub is for software developers. The containers have close to none overhead compared to virtual machines and it has a base container system where every container can be selected as a bases for another container meaning you can build on top of existing infrastructure. The container inheritance is like inheritance in an object oriented programming languages where each subclass add or change some functionality. User who need can also look up the diff as shown in Listing 1 which shows what changes have been made to a container where C means changed, A added and D deleted.

```
C /dev
A /dev/kmsg
C /etc
A /etc/mtab
A /go
A /go/src
A /go/src/github.com
A /go/src/github.com/docker
A /go/src/github.com/docker/docker
A /go/src/github.com/docker/docker/.git
```

Listing 1: "docker diff running-image-name" command example output. [4]

II.1 Fig container administration

Docker allows to easily build and run containers from the built images but it gets more complicated when the application as a whole consists of multiple containers. Docker has a builtin API that allows to easily control almost every aspect of the containers and thus there is a lot of different container management systems - Fig has the most clear focus, is easy to understand and was just at the time of writing this included in Docker itself. Almost immediately when there are linked containers with clear order of dependencies involved, the administration gets more complex. For example it is important that the container with the database is running before the container with the web application is started and tries to access the database through the link. This example with fig configuration file *fig.yml* is illustrated in Listing 2.

```
database:
        image: dockerfile/mongodb
        volumes:
        - ./mongo-data:/data/db

web:
    image: java:openjdk-8-jdk
```

```
8       ports:
9           - "80:80"
10      volumes:
11          - ./artifact/:/artifact/
12      links:
13          - database
14      command: java -jar /artifact/web.jar
            --spring.data.mongodb.host=
            database
```

Listing 2: Web application and database example with Fig

Listing 2 builds two containers from images which are automatically downloaded from https://hub.docker.com and uses volumes to sync host directories with the containers directories to provide the Java jar file and receive all the database files. The "link" argument creates a record to /etc/hosts of the web container which then directs all the traffic to right ip address of the database container.

III. Load balanced web application

Horizontal scaling Scaling horizontally (or scaling out) means adding more nodes to a system, such as adding a new computer or an instance of the application to a distributed software application. For example scaling out from one application server instance to three. [5]

It is expected that the web application that is wanted to scale horizontally is totally stateless or is aware of the complications of state regarding scaling. Example applications could potentially be blogs, newspapers etc that usually allow the user to browse. To be aware of the complications means that the state is stored in one place where every instance of the application can store and query it. In the scope of the project a totally transparent solution is wanted enabling the scaling to be done automatically.

Linked container Docker allows to link containers together. This can be thought of as another unidirectional method for access. Simplest example would be a the first container running the application and the second container running a database. If the application container has a link to the database container then it can be accessed through the link without making the database server port available to the host machine.

To route the traffic to one of the application server instances a load balancer needs to be used as shown on Figure 1. Because docker has a lot of existing containers on https://hub.docker.com an existing container for load balancer is used called tutum/haproxy which balances traffic to every linked container. For simplicity the default configuration with Roundrobin algorithm is used - every request is forwarded to the next instance in circular order. The load balancer is not able to reconfigure itself when a node leaves or joins unless they are executed through Tutum's webservice. In the current project the reconfiguring is not needed because stopping and starting with a new Java jar file does not require configuring the load balancer.

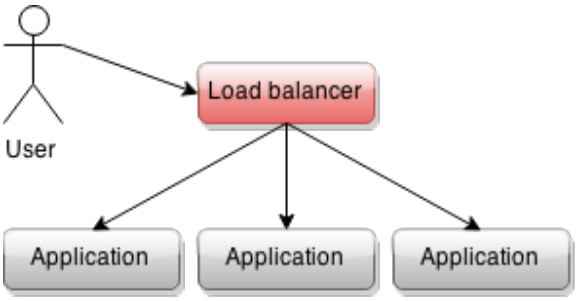

Figure 1: Resource request path with load balancer.

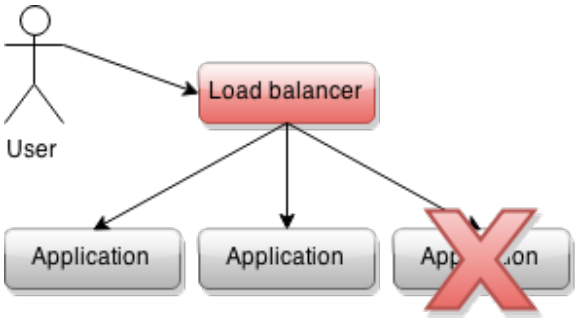

Figure 2: Replacing the application version one container at a time.

The infrastructure shown in Figure 1 is created using the fig.yml file from appendix *Fig administration file for load balanced web application* . If each request from the user takes more than returning certain values straight from the memory, like for example querying data from the database, then creating more instances of the application will not decrease the response time, but having multiple instances will increase the availability during new version deployment if the containers are stopped one at a time like shown in Figure 2.

If an application can instantaneously respond to a request because it has the requested value in cache or the data is stored in memory then the performance gets better when adding containers. The application framework cannot properly use threads for each request because the processing that the thread has to do is to return an immediate value so the bottleneck forms to accepting requests and passing to threads not at business logic executed in threads. This means that the optimal number of web application instances behind a load balancer is equal to the number of processor cores available on the machine which is illustrated in Figure 3. The results were collected with late 2013 Macbook Pro 15" model with 2.0GHz quad-core Intel Core i7 processor and 8GB of 1600MHz DDR3L memory.

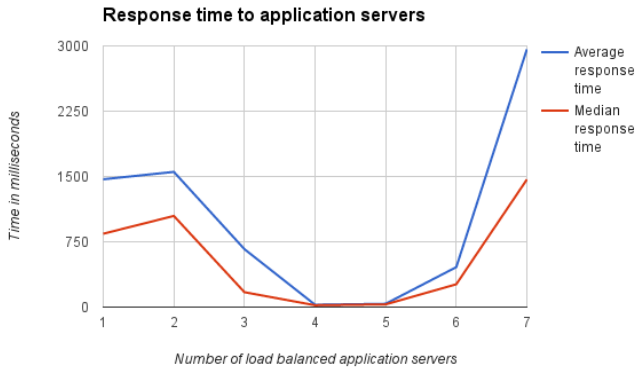

Figure 3: Response time vs application servers with fast response lookup.

IV. MongoDB cluster

For high availability and data integrity the database has to be fault tolerant, able to self heal and the data has to be stored on multiple nodes - this means that small number of stopped nodes will not affect the behavior and results.

For the database MongoDB was chosen which has a flexible document data model and is built with high scalability in mind. Multiple nodes of MongoDB working in collaboration form a MongoDB cluster. A working and easy to deploy example was found from GitHub [6]

and small modifications were done to better suit the needs.

The MongoDB cluster consists of 12 docker containers:

Replica sets - 6 containers running Mongo daemon called *mongod* with 3 replicas of the data.

Configuration servers - 3 containers of Mongo configuration servers which store the metadata for the cluster. These servers are responsible for creating replicas and configuring *mongod* daemons so that data is always stored on multiple nodes.

Routing service - 1 routing container which is used as the endpoint for the web application. This container is responsible for querying and inserting data from and to correct nodes provided by the configuration servers.

Internal DNS - 1 SkyDNS container used for intern DNS.

Service discovery - 1 SkyDock container responsible for discovering new nodes in the cluster and removing not responsive nodes. Inserts the DNS data of the nodes to SkyDNS.

V. Results

A reusable set of containers are created which allows to horizontally scale any stateless Java application that uses MongoDB as the database. The project, it's source code and further execution instructions are available from `https://github.com/taivokasper/high-availability-java-application-base`.

VI. Summary

With the trend of using microservice architecture pattern and organizations often rolling out updates even multiple times a day the need for high availability is more and more relevant. This project focuses on applications or narrowly focused, independently deployable services that have to be available at all times. For this a reusable Docker container set and a concept was introduced that can be reused and requires no extra effort from the developer.

As a future work the created solution needs validation with high performance solutions under heavy load. It also needs to be tested in production environments for an extended period of time to confirm the potential benefits received.

References

[1] Write once, run anywhere? http://www.computerweekly.com/feature/Write-once-run-anywhere. Accessed May 13, 2015.

[2] Write Once and REALLY Run Anywhere with OpenStack and Docker https://www.openstack.org/summit/openstack-summit-hong-kong-2013/session-videos/presentation/write-once-and-really-run-anywhere-with-openstack-and-docker. Accessed May 13, 2015.

[3] Docker for Java Developers: How to sandbox your app in a clean environment http://zeroturnaround.com/rebellabs/docker-for-java-developers-how-to-sandbox-your-app-in-a-clean-environment. Accessed May 13, 2015.

[4] Docker Command Line https://docs.docker.com/reference/commandline/cli/#diff. Accessed May 13, 2015.

[5] Hesham El-Rewini and Mostafa Abd-El-Barr, published Apr 2005, Advanced Computer Architecture and Parallel Processing p. 63-66. Accessed May 13, 2015.

[6] Instant MongoDB sharded cluster https://github.com/jacksoncage/mongo-docker. Accessed May 13, 2015.

A. Fig administration file for load balanced web application

Place a Java jar file next to fig.yml at ./artifact/CoinbaseTraders.jar.

```
web1:
    image: java:openjdk-8-jdk
    volumes:
        - ./artifact/:/artifact/
    command: java -jar /artifact/CoinbaseTraders.jar --server.port=80 --spring.data.
        mongodb.host=192.168.0.100 --spring.data.mongodb.port=27017 --spring.data.mongodb
        .database=production

web2:
    image: java:openjdk-8-jdk
    volumes:
        - ./artifact/:/artifact/
    command: java -jar /artifact/CoinbaseTraders.jar --server.port=80 --spring.data.
        mongodb.host=192.168.0.100 --spring.data.mongodb.port=27017 --spring.data.mongodb
        .database=production

web3:
    image: java:openjdk-8-jdk
    volumes:
        - ./artifact/:/artifact/
    command: java -jar /artifact/CoinbaseTraders.jar --server.port=80 --spring.data.
        mongodb.host=192.168.0.100 --spring.data.mongodb.port=27017 --spring.data.mongodb
        .database=production

lb:
    image: tutum/haproxy-http
    ports:
        - "80:80"
    links:
        - web1
        - web2
        - web3
```

Listing 3: fig.yml